POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

RandShaper: A Library for the Secure Shaping of Pseudo-Random Objects for Post-Quantum Cryptography

Tesi di Laurea Magistrale in
Ingegneria Informatica
Computer Science and Engineering

Author: **Marco Pozzi**

Student ID: 252375
Advisor: Prof. Alessandro Barenghi
Co-advisor: Marco Gianvecchio
Academic Year: 2025-26

Abstract

Today's cryptographic standards are threatened by the rapid advancement in the research and development of quantum computers. To address this problem, the National Institute of Standards and Technologies (NIST) is working for the development and implementation of cryptographic schemes that are resistant to quantum computing. For this reason, NIST has launched multiple competitions, aiming to standardize post-quantum cryptography (PQC), by gathering and evaluating schemes developed by industry, governments and academic institutions.

A fundamental requirement for all proposed digital signature algorithms is the generation of secure pseudo-random objects. This thesis presents the RandShaper library, a framework designed to support these schemes in the generation and shaping of such objects, including modular numbers, fixed-weight strings and permutations. RandShaper starts from binary strings generated by various PRNGs, and implements efficient and, when possible, side-channel-resistant (constant-time) techniques to obtain the target pseudo-random objects.

Finally, performance assessments are conducted by benchmarking all the methods implemented within the library, using the parameters of the NIST digital signature candidates to evaluate both computational efficiency and entropy consumption.

Keywords: Post-Quantum Cryptography, NIST PQC, Digital Signature, Pseudo-Random Number Generation, Constant-Time Algorithms

Abstract in lingua italiana

Gli attuali standard crittografici sono minacciati dai rapidi progressi nella ricerca e nello sviluppo dei computer quantistici. Per far fronte a questo problema, il National Institute of Standards and Technologies (NIST) sta lavorando allo sviluppo e all'implementazione di schemi crittografici resistenti al calcolo quantistico. Per questa ragione, il NIST ha indetto diverse competizioni con l'obiettivo di standardizzare la crittografia post-quantum (PQC), raccogliendo e valutando algoritmi proposti dall'industria, dai governi e dalle istituzioni accademiche.

Un requisito fondamentale per tutti gli algoritmi di firma digitale proposti è la generazione sicura di oggetti pseudo-casuali. Questa tesi presenta la libreria RandShaper, un framework progettato per supportare tali schemi nella generazione e strutturazione (shaping) di questi oggetti, tra cui numeri modulari, stringhe a peso fisso e permutazioni. RandShaper parte da stringhe binarie generate da vari PRNG e implementa tecniche efficienti e, quando possibile, resistenti agli attacchi side-channel (a tempo costante) per ottenere gli oggetti pseudo-casuali desiderati.

Infine, sono state condotte delle valutazioni prestazionali testando tutti i metodi implementati all'interno della libreria, utilizzando i parametri dei candidati per le firme digitali del NIST per valutarne sia l'efficienza computazionale sia il consumo di entropia.

Parole chiave: Crittografia Post-Quantum, NIST PQC, Firme Digitali, Generazione di Numeri Pseudo-Casuali, Algoritmi a Tempo Costante

Contents

Introduction

Cryptography is the process that makes data protection and secure communication possible. One of its main characteristic is to securely hide and code information such that only the intended receiver is able to read it. Nowadays, it is employed everywhere: from the communication and exchange of information between computers to bank cards and password storage.

Modern cryptography takes advantage of computationally hard to break codes and schemes. Within this field, symmetric and asymmetric cryptography rely on different principles. Asymmetric cryptography, in particular, is mainly based on the difficulty of solving two mathematical problems. The first one is the Integer Factorization Problem, which is the challenge of finding the prime factors p, q of a large composite number $n = p \cdot q$. This problem is the backbone of public-key encryption schemes like RSA. The second one is the Discrete Logarithm Problem, which consists in the challenge of finding the secret exponent x, given the base g, the result h and the modulus p, in the modular equation $g^x \equiv h \pmod{p}$. It is the foundation for cryptographic schemes like Diffie-Hellman and ElGamal. The common characteristic of these two problems is that they are easy and fast to compute in one direction, but practically impossible to reverse without an immense computational power.

The level of security of a cryptographic scheme mainly depends on the employed algorithm and its generated keys, secret parameters used to compute the output of a cryptographic function. The security level of a cryptographic algorithm depends on the size of the employed keys, determining the computational requirement in order to guess them correctly.

Real-world protocols, such as TLS and HTTPS, typically employ a hybrid approach: they start with the usage of an asymmetric cryptography scheme (e.g., RSA, DSA, ECDSA, Diffie-Hellman) that relies on a pair of public and private keys; while mathematically robust, they are computationally expensive. For this reason, it is usually employed only at the start of a communication session between two parties to establish in a secure way a shared secret key, which is then used for symmetric schemes (e.g., AES) to encrypt data, guaranteeing fast and efficient communication.

The security of the above described schemes is threatened by the rise of quantum computers. Nowadays, classical computers process information using classical bits, while quantum computers utilize qubits, which can exist in superpositions of states. Particularly, two quantum algorithms are a significant threat to modern cryptographic systems: the Shor's algorithm, which efficiently factors large numbers and computes discrete logarithms, and Grover's algorithm, which accelerates brute-force search processes, undermining the security of symmetric-key encryption by effectively halving key strength.

Today's quantum computers are still primitive; however, if more advanced and robust versions are developed, they have the potential to rapidly solve the mathematical problems on which modern cryptography relies. For this reason, governments and companies are already preparing for this transition by researching and implementing post-quantum cryptography (PQC) schemes, which relies on mathematical problems resistant to quantum attacks.

This thesis presents the RandShaper library, a framework designed to support existing post-quantum schemes, specifically in the generation and shaping of pseudo-random objects, such as binary strings and modular numbers, for which they are fundamental building blocks in the key-generation, signing, and verification processes of PQC algorithms. Pseudo-random object generation usually requires a trade-off between execution efficiency, optimal entropy consumption, and security: in particular, this thesis focuses on the constant-time property, a crucial defense mechanism against side-channel timing attacks that existing libraries often lack. This library collects and efficiently implements already existing methods, as well as proposes new methods and variants for generating the target pseudo-random objects.

The following document is organized in four chapters:

- **Chapter 1**: *Background and State of the Art* introduces the necessary concepts required to understand the current PQC panorama. It provides an overview of the National Institute of Standards and Technology (NIST) program for the standardization of PQC algorithms and reviews current primitives for generating pseudo-random numbers. The chapter concludes by defining the constant-time property and highlighting its critical importance in modern cryptographic schemes.
- **Chapter 2**: *The RandShaper Framework* describes the main contribution of this thesis. It presents the architectural design of the library and explains the proposed methods for generating each pseudo-random object. Additionally, it provides a theoretical analysis of both the computational complexity and the security properties of each implemented approach.

- **Chapter 3**: *Experimental Evaluation* presents the benchmark results obtained, evaluating the computational performance and entropy consumption of the proposed implementations. It describes the tests setup and the virtual machine used for execution the benchmarks. The chapter concludes with a report of the collected data and with a comparative analysis of the algorithms.
- **Chapter 4**: *Conclusions* summarizes the obtained results and discusses how the library can be employed in today's PQC scemes.

1 | Background and state of the art

This chapter introduces the necessary background to understand the reasons behind the development of the library described in Chapter 2. First, it presents the NIST standardization program for Post-Quantum cryptographic schemes, focusing on the ongoing competition for the selection of additional digital signatures. It also provides an overview of the current primitives for generating pseudo-random numbers. Finally, it explains the constant-time property of an algorithm and why it is a fundamental requirement for current cryptographic primitives and proposals.

1.1. NIST Post-Quantum Cryptography Standardization Program

The National Institute of Standards and Technology (NIST) is the national measurement institute (NMI) for the United States. One of NIST's principal focuses is developing trustworthy cryptographic techniques in an open, collaborative process with industry, government, and academic institutions.

NIST is currently working on meeting modern cryptographic requirements, including standards and technologies that should be implemented to face the PQC problem.

For this reason, in December 2016, following the "Call for Proposals for Post-Quantum Cryptography Standardization" announced in the Federal Register [39], NIST acknowledged the necessity of designing and implementing standards for countering the danger that quantum computers pose to the security of digital information.

A gathering stage was opened, where cryptographers could propose algorithms until the end of 2017.

Through a multi-year international competition involving industry, academia, and governments, NIST released three PQC Federal Information Processing Standards (FIPS) in

2024: the Key-Encapsulation Mechanism Standard FIPS 203 (ML-KEM) [33], and the two Digital Signature Standards FIPS 204 (ML-DSA) [34] and FIPS 205 (SLH-DSA) [35].

In addition to the three algorithms specified in the initial FIPS standards, the Falcon digital signature algorithm [24] and the HQC key encapsulation mechanism [25] have been selected for ongoing standardization.

In September 2022, NIST called for additional digital signature proposals to be considered in the PQC standardization process to diversify its post-quantum signature portfolio. Since the primary general-purpose signature standards (ML-DSA and Falcon) are both based on the computational hardness of structured lattices, NIST expressed particular interest in additional signature schemes based on different security assumptions, as well as signature schemes with short signatures and fast verification. Consequently, another gathering stage was opened.

Between the opening stage and June 2023, 40 signature schemes were accepted in the Round 1 Additional Signatures. In October 2024, 14 candidates were announced for the Second Round.

These 14 schemes were divided by NIST into 6 categories, based on the underlying problem: Code-based (CROSS [8], LESS [9]), Isogeny (SQIsign [2]), Lattice-based (HAWK [20]), MPC-in-the-Head (Mirath [3], MQOM [12], PERK [1], RYDE [6], SDitH [4]), Multivariate (MAYO [18], QR-UOV [5], SNOVA [43], UOV [19]), and Symmetric-based (FAEST [11]).

Recently, NIST announced the Round 3 Additional Digital Signature candidates, which saw the elimination of CROSS, LESS, and Mirath. However, for the scope of this thesis, the 14 candidates of the Second Round are maintained as the algorithms tested in the experimental chapter (Chapter 3).

For all of these schemes, the Key Generation, Signature, and Verification processes, as well as other auxiliary functions, strictly require the generation of pseudo-random objects, such as binary strings (Section 2.2), modular numbers (Section 2.3), fixed-weight vectors (Section 2.4) and permutation arrays (Section 2.5).

1.2. Cryptographic Primitives for Pseudo-Random Number Generation

A Pseudo-Random Number Generator (PRNG) is an algorithm used to produce pseudo-random sequences, i.e., a stream of zeros and ones that may be divided into sub-streams

or blocks of random numbers [40].

A PRNG uses one or more inputs, called seeds, and generates multiple pseudo-random digits. The outputs of a PRNG are deterministic functions of the seed; i.e., all true randomness is confined to the seed generation.

We will analyze three cryptographic primitives that are implemented in our contribution (Chapter 2) to act as PRNGs: the Advanced Encryption Standard (AES), SHAKE, and ChaCha.

1.2.1. Advanced Encryption Standard (AES) and Counter Mode

The Advanced Encryption Standard (AES) specifies a FIPS-approved cryptographic algorithm [31] that can be used to protect digital data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information.

The input and output for the AES algorithm each consist of sequences of 128 bits. These bits are divided into a 4×4-byte matrix called the state. The Cipher Key for the AES algorithm is a sequence of 128, 192, or 256 bits.

The AES scheme is divided into rounds (10, 12, or 14 rounds based on the length of the key), where in each round four operations are applied to the state: `SubBytes` (a non-linear function), `ShiftRows` and `MixColumns` (a permutation layer), and `AddRoundKey` (a XOR operation with the round key). Figure 1.1 shows the encryption scheme of AES.

Since AES is a block cipher, it produces fixed 128-bit blocks. To use it as a PRNG, the Counter (CTR) mode of operation is employed. In this mode, the seed acts as the AES Cipher Key, and a sequentially incremented counter, often initialized with an Initialization Vector (IV), is encrypted for each block. The concatenated output of these encrypted blocks forms the target pseudo-random sequence. Figure 1.2 illustrates the CTR mode of operation.

To meet modern cryptographic standards for pseudo-random number generation, the CTR mode is embedded within the Deterministic Random Bit Generator (DRBG) framework, specifically `CTR_DRBG`, as defined by the NIST SP 800-90A Recommendation [10]. The internal state of `CTR_DRBG` is represented by two components: a key K and a value V. The management of this state is carried out by three functions:

- `Instantiate`: Initializes the internal state, starting from the entropy input and a nonce, safely constructing the initial K and V values.
- `Generate`: Generates pseudo-random bits after instantiation while simultaneously

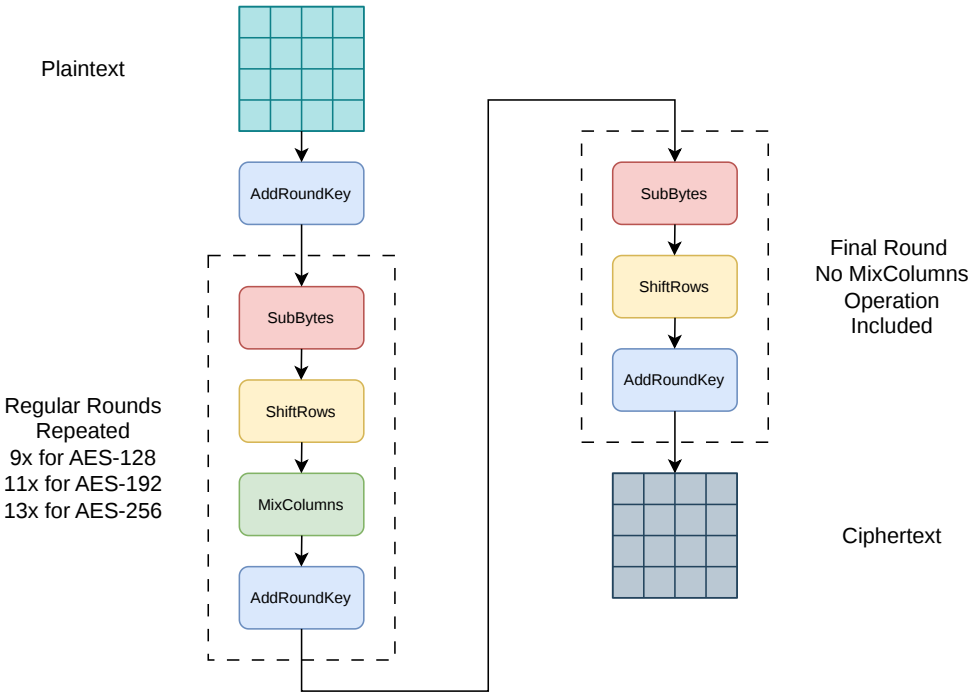

Figure 1.1: The AES Encryption Scheme

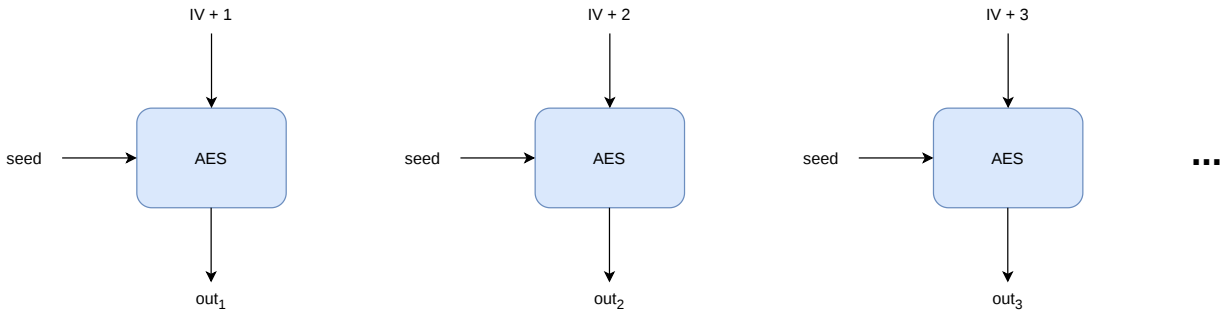

Figure 1.2: The Counter (CTR) mode of operation utilized as a PRNG.

updating the internal state.

- `Update`: Updates the internal state of the `CTR_DRBG` using the provided data, completely replacing the values of K and V.

1.2.2. SHAKE

SHAKE128 and SHAKE256 are extendable-output functions (XOFs) and are part of the Secure Hash Algorithm-3 (SHA-3) family, a FIPS-approved cryptographic standard [32].

The primary difference between the SHA-3 hash functions (SHA3-224, SHA3-256, SHA3-384, and SHA3-512) and the SHA-3 XOFs (SHAKE128 and SHAKE256) is that the former produce a fixed-length output, whereas the latter allow the output length to be chosen arbitrarily to meet the requirements of individual applications. Furthermore, the suffixes in SHAKE ("-128" and "-256") indicate the security strengths that these functions can generally support, in contrast to the suffixes of the hash functions, which indicate their respective digest lengths.

Each SHA-3 function is based on an instance of the KECCAK algorithm, which NIST selected as the winner of the SHA-3 Cryptographic Hash Algorithm Competition. The core of the Keccak family is the Keccak-p permutation. The Keccak-p permutation operates on a state of b bits, where $b \in \{25, 50, 100, 200, 400, 800, 1600\}$, and applies a series of elementary step mappings (denoted by $\theta, \rho, \pi, \chi, \iota$). Together, these operations mix and diffuse the bits across the b-bit state, resulting in a highly non-linear permutation.

Keccak is built upon a unique and flexible cryptographic structure known as the *sponge construction*. This framework is designed to specify functions on binary data with an arbitrary output length. The construction employs three main components: the Keccak-p function (denoted by f), a parameter called *rate* (denoted by r), and a padding rule (denoted by *pad*). The function produced from these components is called a sponge function. A sponge function takes two inputs: an initial bit string, denoted by N (the seed), and the desired bit length of the output string, denoted by d. The analogy to a sponge is that an arbitrary number of input bits are "absorbed" into the internal state of the function, after which an arbitrary number of pseudo-random output bits are "squeezed" out of it.

Because of this ability to "squeeze" an infinite stream of bits from a single initial seed, SHAKE is widely adopted as a highly efficient and secure Pseudo-Random Number Generator (PRNG) in modern cryptographic schemes. The sponge construction is illustrated in Figure 1.3.

Keccak can be efficiently implemented in a parallelized manner to enhance performance. This method seeks to exploit the SIMD (Single Instruction, Multiple Data) capabilities of modern CPUs, such as AVX2 (Advanced Vector Extensions) instructions, which can process multiple data streams simultaneously.

In our contribution (Chapter 2), we use a parallelized version that exploits the SIMD (Single Instruction, Multiple Data) capabilities of modern CPUs, such as AVX2 instructions, to process four Keccak operations simultaneously [26].

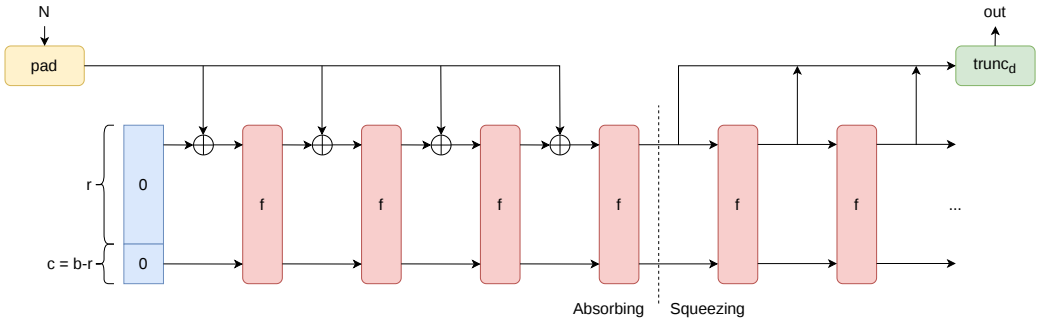

Figure 1.3: The Keccak Sponge Construction

"expa"	"nd 3"	"2-by"	"te k"
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Counter	Nonce	Nonce

Figure 1.4: The ChaCha initial internal state

1.2.3. ChaCha

ChaCha [38] is a stream cipher and a variant of the Salsa20 family. It is built on a pseudo-random function that utilizes ARX (Addition, Rotation, XOR) operations. The core function maps a 256-bit key, a 64-bit counter, a 64-bit nonce, and a 128-bit constant, arranged as a 4×4 matrix of 32-bit words (called the *internal state*), to produce a 512-bit block of the keystream.

Internally, the cipher uses 32-bit addition modulo 2^{32}, bitwise XOR, and constant-distance rotation operations on the internal state. This state is made up of eight words of key, two words of counter, two words of nonce, and four fixed words representing the ASCII string `expand 32-byte k`. Figure 1.4 shows the initial internal state of ChaCha.

The core operation in ChaCha is the *quarter-round*, denoted as $\mathrm{QR}(a, b, c, d)$, which takes a four-word input and produces a four-word output. Figure 1.5 shows the quarter-round operation applied to the four inputs.

Odd-numbered rounds apply the $\mathrm{QR}(a, b, c, d)$ function to each of the four columns in the

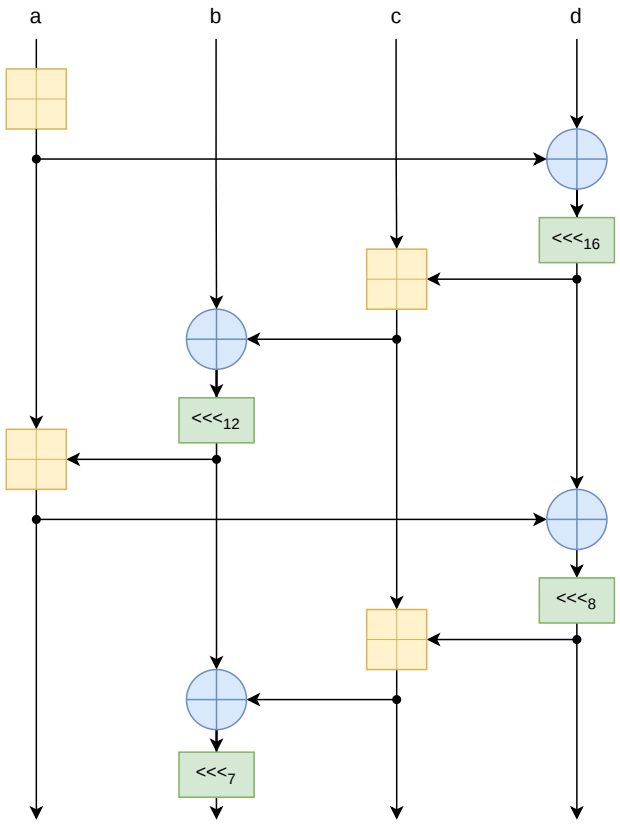

Figure 1.5: The ChaCha Quarter-Round Operation

4×4 matrix, while even-numbered rounds apply it to each of the four diagonals. In total, 20 rounds are executed for the standard ChaCha20 variant.

At the end of the 20 rounds, the original input words are added to the corresponding output words, and the result is serialized by sequencing the words one-by-one in little-endian order.

1.3. Arithmetic Coding

Arithmetic coding [45] is a data compression technique that encodes data (the data string) by creating a code string which represents a fractional value on the interval between 0 and 1. The coding algorithm is symbol-wise recursive; i.e., it operates upon and encodes (decodes) one data symbol per iteration or recursion.

According to Shannon's information theory, an optimal coding method generates code val-

ues that are uniformly distributed across the interval $[0, 1)$. Arithmetic coding achieves this uniform distribution, removing statistical dependence and redundancy. As the message length grows, it is asymptotically optimal, achieving compression performance equal to the source entropy.

Fundamentally, arithmetic encoding works by creating a sequence of "nested intervals": the process starts with a large range and progressively narrows it down based on the data symbols encountered. On the other end, arithmetic decoding starts from the narrowed interval, and step-by-step it enlarges the range until the original one is obtained, retrieving a data symbol at each step.

In theory, arithmetic coding requires infinite precision, but it is not possible to use infinite precision in real hardware. To handle this fixed-precision problem, practical implementations use fixed-size registers to store only the significant digits, or mantissa, of the interval. Instead of using floating-point numbers in $[0, 1)$, the implementation maps the interval to large integers.

The logic behind the decoding phase of arithmetic coding can be used for pseudo-random generation contexts. Starting from a pseudo-random large integer x, a division operation by a base n is applied, obtaining a remainder r, where $0 \leq r < n$. Iteratively, the division operation is applied to the resulting quotient, obtaining a new pseudo-random number each time.

1.4. The Constant-Time Implementation Paradigm

Constant-time requires that timing variations cannot be correlated with secret data like cryptographic keys. It ensures that cryptographic code execution time remains independent of secret values, preventing attackers from extracting sensitive information through timing measurements.

It follows a fundamental principle, often referred to as a golden rule: secret information may only be used as input to operations where that input has no impact on resource usage or execution time. In simple words, resource usage must be independent of secrets.

Timing attacks are a subset of a more general class of attacks known as side-channel attacks. A computer system runs operations in a conceptual abstract machine that takes some inputs and provides some outputs; side-channel attacks are all about exploiting the difference between that abstract model and the real system.

The best way to mitigate attacks from leaking out secret information is to ensure that

the implementations are constant time, meaning that the execution time of cryptographic operations should remain independent of the input.

Constant-time implementations are pieces of code that do not leak secret information through timing analysis. This means that the execution time of the implementation does not depend on secret elements.

Non-constant-time operations can occur in various scenarios, in particular:

1. *Conditional Jumps*: When a conditional jump operation is performed, the chosen branch often depends on the value of a variable. Since different branches execute different blocks of code, the overall execution time varies. This delay can be measured by an attacker to infer whether the jump was taken or not, thus leaking information about the underlying data, which is often a secret. An example is a comparator that checks a password character by character and exits the loop early at the first mismatch: depending on the execution time (i.e., how many iterations were performed), an attacker can deduce the length of the guessed correct prefix.
2. *Memory accesses*: When the program has to read or write data in memory, the execution time depends on where the target data is stored. Specifically, if the data is in the cache, it can be retrieved faster. These differences can be measured by an attacker.
3. *Integer divisions*: A division operation's execution time may depend on the dimension of the numbers involved; in particular, smaller numbers are handled faster than large numbers.

Constant-time is an additional security property desirable by NIST [37].

The generation of pseudo-random objects in PQC schemes often depends on secret data, so here arises the necessity of using algorithms that are constant-time, to prevent timing attacks. In our contribution (Chapter 2) we will see both constant-time algorithms and non-constant-time approaches.

2 | The RandShaper Framework

This chapter presents the methods implemented in the RandShaper library. For each method, we describe the approach, eventually provide its pseudo-code, discuss its security analysis, and evaluate its computational complexity to identify its optimal parameters. First, we describe the method for managing the bits coming out from the PRNG; after this, we analyze the procedures for generating arrays of numbers modulo n; then, we introduce the methodologies for creating fixed-weight binary vectors; finally, we present the algorithms used to generate permutations.

2.1. Software Architecture

The RandShaper library is designed with a three-tiered architecture to guarantee flexibility and ease of integration. Figure 2.1 illustrates the layers of the framework, which are structured as follows:

- **Cryptographic Backend Layer**: it contains the raw implementations of the cryptographic primitives discussed in Section 1.2, which are responsible for the generation of pseudo-random numbers (i.e., AES, SHAKE, ChaCha).
- **PRNG Abstraction Layer**: this intermediate layer is responsible for the internal buffering and the management of the pseudo-random bits obtained by invoking the primitives from the layer below.
- **Object Shapers Layer**: starting from the pseudo-random bits provided by the PRNG Abstraction Layer, this highest level is responsible for the secure generation of more complex pseudo-random objects (i.e., Modular Arrays, Fixed-Weight Vectors, and Permutations).

One of the main advantages of this architecture is that each layer depends on the one below only for its output. For example, the `PRNG Abstraction Layer` works independently of the selected cryptographic PRNG primitive.

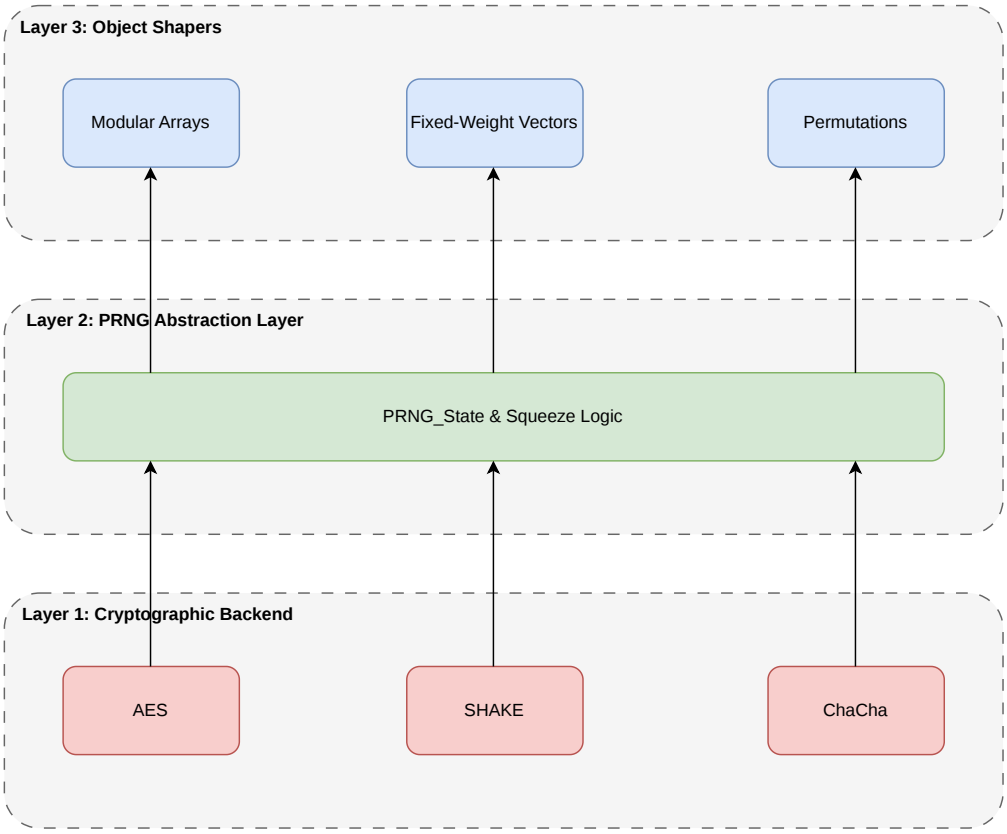

Figure 2.1: High-level software architecture of the RandShaper framework.

2.2. Management of the bits generated by the PRNG

2.2.1. The PRNG Abstraction Layer

The RandShaper framework's basic component is the generation and management of pseudo-random binary strings. Although the library employs existing cryptographic primitives, precisely AES [31], SHAKE [32], and ChaCha [38], to generate these bits, the algorithms responsible for the generation of more complex pseudo-random objects, such as Fixed Weight Strings (Section 2.4), arrays of integers modulo n (Section 2.3), and Permutations (Section 2.5), do not need to keep track of which specific PRNG is selected.

For this reason, an abstraction layer is required for the management of the extraction of pseudo-random bits, regardless of the employed cryptographic algorithm. The core design relies on a generic data structure called `PRNG_State`, which is used as a context passed as an argument to all sampling methods. Whenever randomness is required, a generic function called `prng_squeeze` is invoked to extract the required pseudo-random bits.

```
typedef struct {
    // The buffer where to save the unused bits
    uint8_t buffer[PRNG_BLOCK_SIZE_BYTES];

    // Counters to track the state of the buffer
    size_t bits_in_buffer;    // Number of valid bits left
    size_t current_bit_idx;   // Index of the next valid bit

    uint64_t total_generated_bits;  // Security parameter

    // Memory space to save the internal state of the PRNG
    ALIGN(32) uint64_t internal_state[128];
} PRNG_State;
```

Figure 2.2: The `PRNG_State` data structure.

Figure 2.2 describes the `PRNG_State` data structure. The fields are defined as follows:

- `buffer`: a byte array containing the generated block of pseudo-random bits.
- `bits_in_buffer`: the number of generated bits that are still unused.
- `current_bit_idx`: the starting index of the first unused bit.
- `total_generated_bits`: a counter tracking the overall number of generated bits. This acts as a security boundary for specific engines like AES (ensuring no more than 2^{56} bits are generated using the same key state).
- `internal_state`: the memory space allocated to hold the specific internal state of the chosen PRNG algorithm.

2.2.2. The PRNG Initialization

To initialize the PRNG, three sequential functions must be invoked:

- `prng_init`: Clears the `PRNG_State` structure, setting all its internal variables and buffers to zero.
- `prng_absorb` or `prng_absorb_iv`: Initializes the cryptographic engine using the provided seed as the key and, optionally, a provided Initialization Vector (IV). For Sponge-based constructs, such as SHAKE, this function can be invoked multiple times to absorb longer seeds.
- `prng_complete_init`: For Sponge-based constructs (like SHAKE), this applies the domain separation padding to the input and switches the internal state from the ‘Absorb‘ phase to the ‘Squeeze‘ phase. For block or stream ciphers (like AES or

ChaCha), this function does nothing.

2.2.3. The PRNG Squeeze Operation

The `prng_squeeze` function is responsible for managing the internal state buffer, particularly by refilling the buffer with a new pseudo-random block when empty and outputting the requested pseudo-random binary string.

Unlike many traditional PRNG APIs, the `prng_squeeze` operation in the RandShaper framework accepts the exact target number of pseudo-random bits, denoted as N_{bits}, as an argument. This eliminates the standard constraint of requiring byte-aligned requests (i.e., strictly multiples of 8 bits).

To optimize performance based on specific algorithmic requirements, the RandShaper library implements two versions of the squeeze operation: the Standard version and the Fast-Aligned version.

Standard Version The main advantage of this version is the lack of discarded source randomness. The algorithm uses, when possible, the `memcpy` intrinsic to copy bytes from the buffer array to the output array. The main problem with the `memcpy` operation is that it performs only byte-aligned copy operations. Since the RandShaper library also manages bit-level requests, if a previous squeeze operation was executed with a requested number of bits N_{bits} that is not a multiple of 8, the `current_bit_idx` will not be byte-aligned, and therefore the `memcpy` intrinsic cannot be used directly. The solution is to copy one byte at a time by using a sliding window between two consecutive bytes of the buffer. Finally, if the remaining bits to copy are fewer than 8, a bit-by-bit copy is performed.

Fast-Aligned Version The approach presented in the Standard version to solve the non-byte-aligned problem requires a computational overhead, since the sliding-window approach can be expensive. A faster solution is the one implemented in the Fast-Aligned version. At the end of each squeeze operation, if `current_bit_idx` is not a multiple of 8, it is incremented to the next multiple of 8. This allows the next squeeze function to always use the `memcpy` intrinsic, since the buffer is necessarily byte-aligned. This approach introduces an entropy waste of up to 7 bits per squeeze operation, but it is overall computationally faster.

2.2.4. Parallel Vectorized Generation (AVX2)

The RandShaper library also offers the `prng_init_x4`, `prng_absorb_x4`, `prng_complete_init_x4`, and `prng_squeeze_x4` functions to leverage vectorized cryptographic primitives, such as SHAKE-128x4 and SHAKE-256x4, to absorb four distinct seeds and squeeze four independent streams of pseudo-random bits concurrently. The main logic of these functions is the same as presented in section 2.2.2 and section 2.2.3; the only difference is that all internal operations are applied simultaneously across the four streams.

2.3. Generation of Arrays of Numbers Modulo n

This chapter addresses the problem of generating an array of numbers modulo n, where $n \in \mathbb{N}$.

Definition 2.1 (Set of integers modulo n)**.** *Let n be a positive integer. The set $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$ is defined as the set of integers modulo n.*

We can formalize the set representing arrays of such integers:

Definition 2.2 (Set of arrays with coordinates in $\mathbb{Z}_n$)**.** *Let n and l be positive integers. The set $\mathbb{Z}_n^l$ is defined as the Cartesian product of l instances of $\mathbb{Z}_n$, representing all possible vectors of length l whose elements belong to $\mathbb{Z}_n$.*

From Definition 2.1, we can define the basic Number Modulo n Selection Problem as follows:

Problem 2.1 (Number modulo n selection problem)**.** *Select, uniformly at random, an integer x from the set $\mathbb{Z}_n$.*

At first sight, one might consider simply using the modulus operator (RAND_NUMBER mod n) to solve Problem 2.1. However, using this method, the uniformity condition is generally not respected: it introduces a statistical bias towards smaller numbers, which have a higher probability of being picked compared to larger numbers. A simple example can describe this problem, known as modulo bias.

Example 2.1 (Modulo bias with $n = 5$)**.** *Suppose a PRNG generates an integer uniformly distributed in the range $[0, 7]$ (i.e., 8 possible values), and we want to generate a number modulo $n = 5$. If we apply the modular operator, the possible outcomes are:*

- $P(X = 0) = \frac{2}{8}$ *(PRNG outputs 0 or 5)*

- $P(X = 1) = \frac{2}{8}$ *(PRNG outputs* 1 *or* 6*)*
- $P(X = 2) = \frac{2}{8}$ *(PRNG outputs* 2 *or* 7*)*
- $P(X = 3) = \frac{1}{8}$ *(PRNG outputs* 3*)*
- $P(X = 4) = \frac{1}{8}$ *(PRNG outputs* 4*)*

Consequently, smaller numbers (in this case $0, 1, 2$*) have a higher probability of being generated compared to larger numbers (*$3, 4$*).*

Finally, we can introduce the problem that the methods in the following sections aim to solve.

Problem 2.2 (Arrays with coordinates in $\mathbb{Z}_n$ selection problem)**.** *Let* n *and* l *be two positive integers. Select, uniformly at random, a vector of length* l *from the set* $\mathbb{Z}_n^l$*.*

2.3.1. Monolithic Arithmetic Decoding Method

Base idea As explained in Section 1.3, this method exploits the decoding phase of arithmetic coding [45] to generate an array of l integers. These elements are obtained by applying l successive division operations by a base n to a large pseudo-random integer x, where the l retrieved remainders represent the final output array.

Modular Bias prevention If the large integer x is not large enough, the arithmetic decoding method will suffer from the modular bias problem. To avoid this, we need to establish a lower bound on the required bits such that the modular bias is bounded by $2^{-\lambda}$ (i.e., the statistical distance from the uniform distribution does not exceed $2^{-\lambda}$), where λ is a security parameter (usually $\lambda \in \{128, 192, 256\}$).

Let $d = \lceil \log_2(n) \rceil$ be the minimum number of bits required to represent N. Each division will consume at most d bits of the starting integer. Consequently, if we let b be the minimum number of bits of x required to respect the security margin, we have the following inequality:

$$b \geq l \cdot d + \lambda.$$

Constant-Time Exact Division Divisions and modular operations on large numbers can be computationally expensive and inefficient. Furthermore, from a hardware perspective, as explained in Section 1.4, they can be non-constant time operations.

To overcome this problem, the algorithm employs the division by invariant integers method originally proposed by Granlund and Montgomery [27] and popularized in *Hacker's*

Delight [44].

The core mathematical intuition behind this method is to substitute the expensive integer division $\frac{y}{n}$ with a multiplication by its fractional inverse $\frac{1}{n}$, which can be approximated as the fraction $\frac{m}{2^k}$, transforming the division into $\frac{(y \cdot m)}{2^k}$. This operation does not require any division operation since dividing by a power of 2 is equivalent to applying right-shift operations to the numerator.

To guarantee that this approximation yields the exact quotient $q = \lfloor y/n \rfloor$ for every possible 64-bit dividend y, the precision parameter k must be sufficiently large, and the multiplier m, called the "Magic Number", must be properly rounded. By defining $k = 64 + s$, where $s = \lceil \log_2(n) \rceil$, the variables are pre-computed at the start of the algorithm as:

$$s = \lceil \log_2(n) \rceil$$
$$m = \left\lfloor \frac{2^{64+l}}{n} \right\rfloor + 1$$

The theoretical division simply requires computing $q = \left\lfloor \frac{y \cdot m}{2^{64+s}} \right\rfloor$, but this operation can lead to register overflows, since the product requires 128 bits and the magic number m itself might exceed 64 bits. To overcome this problem, the algorithm computes the division operation in three steps using strictly 64-bit hardware registers. Given a generic 64-bit integer dividend y, the exact quotient q is computed through the following sequence of constant-time operations:

$$q_1 = \left\lfloor \frac{y \cdot m}{2^{64}} \right\rfloor$$
$$q_2 = q_1 + \left\lfloor \frac{y - q_1}{2} \right\rfloor$$
$$q = \left\lfloor \frac{q_2}{2^{s-1}} \right\rfloor$$

In the sequence above, q_1 effectively extracts the high 64 bits of the 128-bit product. The computation of q_2 safely compensates for the potential overflow of the magic number, and the final operation shifts the result by the remaining $s - 1$ bits.

Finally, the exact remainder r is easily obtained with a single multiplication and subtraction:

$$r = y - (q \cdot n)$$

Base Change (Long Division) Now that an efficient engine for computing 64-bit exact divisions is established, we must address the fact that the monolithic integer x typi-

cally exceeds 2^{64}, meaning it cannot be contained within a single hardware register. To fix this, the algorithm employs a classical multi-precision long division approach, representing x as an array of 32-bit blocks (i.e., in base 2^{32}).

Algorithm 2.1 shows the pseudo-code of the Monolithic Arithmetic Decoding method. The division iterates over the array from the most significant digit down to the least significant digit. At each step, a 64-bit partial dividend is constructed by concatenating the 32-bit remainder from the previous operation with the current 32-bit block. The constant-time division is then performed on this 64-bit value, updating the quotient in-place and propagating the new carry to the next iteration. Once the last digit is processed, the final carry exactly represents the extracted element modulo n.

Complexity The overall computational complexity of this approach is quadratic, $O(l^2)$, with respect to the target array size l. This occurs because the outer loop performs l extractions, and the inner long division loop iterates over k blocks. Since the required bit-length b (and consequently k) scales linearly with l to respect the security margin, an increase in l results in an increase in the number of division operations required per extraction.

Section 2.3.2 and Section 2.3.3 present two methods to mitigate this quadratic-complexity problem at the cost of wasting some entropy of the PRNG.

2.3.2. Chunked Arithmetic Decoding Method

The Chunked Arithmetic Decoding approach is an alternative to the Monolithic Arithmetic Decoding method (Section 2.3.1). The main objective of this algorithm is to reduce the quadratic computational overhead presented in the previous subsection.

Base Idea Algorithm 2.2 shows the pseudo-code of the Chunked Arithmetic Decoding method. The main component of the algorithm is still the Monolithic Arithmetic Decoding method (Algorithm 2.1), but instead of processing the entire array of length l at once, it divides the workload into smaller, manageable chunks of a fixed size c_{size} (where $0 < c_{size} \leq l$).

Specifically, the algorithm computes $c = \lfloor l/c_{size} \rfloor$ full chunks. Each chunk is generated independently by invoking the monolithic algorithm to extract c_{size} numbers modulo n. If l is not a perfect multiple of c_{size}, a final smaller "tail" chunk is generated to cover the remaining elements. Finally, all the generated chunks are concatenated to form the requested array of l elements.

Input: `prng_state`, n (modulus), l (num elements), λ (security margin)

Output: a (an l-element array modulo n)

Data: x: Monolithic integer array of k 32-bit blocks

function `GenArrayMod(prng_state, n, l, λ):`
  `// Compute required bits to prevent modular bias`
  $d \leftarrow \lceil \log_2(n) \rceil$
  $b \leftarrow (l \times d) + \lambda$
  $k \leftarrow \lceil b/32 \rceil$ `// Number of 32-bit blocks required`
  `// Generate the monolithic integer` x
  $x[0 : k-1] \leftarrow$ GetRandBlocks32(`prng_state`, k)
  $m, s \leftarrow$ ComputeMagic(n)
  for $i \leftarrow 0$ **to** $l - 1$ **do**
    `carry` $\leftarrow 0$
    for $j \leftarrow k - 1$ ***downto*** 0 **do**
      $v \leftarrow$ (`carry` $\ll 32$) $+ x[j]$
      `quotient`, `remainder` $\leftarrow$ CT_DivMod(v, n, m, s)
      $x[j] \leftarrow$ `quotient`
      `carry` $\leftarrow$ `remainder`
    $a[i] \leftarrow$ `carry`
  return a

Algorithm 2.1: Monolithic Arithmetic Decoding

Optimization The algorithm can be heavily optimized by leveraging AVX2 intrinsics. Instead of processing each chunk sequentially, the optimized implementation groups the workload into macro-blocks of four chunks (i.e., $4 \times c_{size}$ elements). The algorithm performs the constant-time base change on these four independent chunks simultaneously using 256-bit vector registers.

Security Analysis The Chunked Arithmetic Decoding method preserves the constant-time execution property of the monolithic approach.

However, the primary trade-off of this variant lies in the entropy consumption (i.e., the total number of pseudo-random bits required from the PRNG). In the monolithic method, the security margin λ is added only once. In the chunked approach, the core algorithm is invoked independently for each of the c chunks (plus an additional invocation if a tail exists). Consequently, the security margin λ must be enforced for every individual chunk to prevent the modular bias.

Let $d = \lceil \log_2(n) \rceil$. To generate a single full chunk of size c_{size}, the required number of bits is $b_{chunk} \geq c_{size} \cdot d + \lambda$. Assuming for simplicity that l is a perfect multiple of c_{size},

the total number of required bits b_{total} becomes:

$$b_{total} \geq c \cdot (c_{size} \cdot d + \lambda) = l \cdot d + c \cdot \lambda$$

In the general case where a tail of size $t = l \bmod c_{size}$ exists, an additional $t \cdot d + \lambda$ bits are required, yielding:

$$b_{total} \geq l \cdot d + (c + 1) \cdot \lambda$$

In both scenarios, the randomness requirement is significantly greater than the $l \cdot d + \lambda$ bits required by the monolithic version. This entropy waste is the architectural cost paid to mitigate the quadratic computational scaling.

Complexity The computational complexity of the Chunked Arithmetic Decoding approach is $O(l \cdot c_{size})$.

By restricting the chunk size to a manageable constant, the quadratic scaling of the monolithic approach ($O(l^2)$) is successfully reduced to a linear complexity with respect to the target array size l.

However, the parameter c_{size} introduces a practical performance trade-off. If c_{size} is too large, the computational cost approaches the quadratic behavior of the monolithic version. Conversely, if c_{size} is too small, the PRNG will become the main bottleneck, since it would spend most of its time in generating the large random numbers, rather than generating the actual target elements.

2.3.3. Chunked Tree Arithmetic Decoding Method

The Chunked Tree Arithmetic Decoding method is a variant of the Monolithic and Chunked approaches (Section 2.3.1 and Section 2.3.2) to mitigate the quadratic computational cost discussed previously.

Base Idea Unlike the previous methods that rely on the Long Division algorithm over multi-precision arrays, this approach uses operands within 64-bit registers, performing secure and fast divisions on single-precision integers.

Algorithm 2.3 shows the pseudo-code of the Chunked Tree Arithmetic Decoding approach. The algorithm begins by pre-computing quadratic powers of the target modulus: n, n^2, n^4, n^8, ..., up to a maximum depth d. This value is chosen such that the value n^{2^d} can be represented with a 64-bit register. This value becomes the *root modulus* of the tree, and the tree will ultimately yield $k = 2^d$ leaf elements.

Input: `prng_state`, n (modulus), l (num elements), λ (security margin), c_{size} (chunk size)

Output: a (l-element array modulo n)

Data: `chunk`: Array holding temporary chunk results

function `ChunkedGenArrayMod(prng_state, n, l, λ, c_size)`:
- $c \leftarrow \lfloor l/c_{size} \rfloor$ `// Number of full chunks`
- `tail` $\leftarrow l \bmod c_{size}$ `// Remaining elements`
- $a \leftarrow$ empty array of size l
- `idx` $\leftarrow 0$
- `// Full chunks`
- **for** $i \leftarrow 0$ **to** $c - 1$ **do**
  - `chunk` $\leftarrow$ GenArrayMod(`prng_state`, n, c_{size}, λ)
  - a[`idx` : `idx` $+ c_{size} - 1$] $\leftarrow$ `chunk` `// Concatenate`
  - `idx` $\leftarrow$ `idx` $+ c_{size}$
- `// Tail`
- **if** `tail` > 0 **then**
  - `chunk` $\leftarrow$ GenArrayMod(`prng_state`, n, `tail`, λ)
  - a[`idx` : $l - 1$] $\leftarrow$ `chunk` `// Concatenate tail`
- **return** a

Algorithm 2.2: Chunked Arithmetic Decoding

Next, the algorithm utilizes the PRNG to generate a single large random number, called the *Root Value*. This value is reduced modulo n^{2^d}. To effectively prevent the modular bias problem, the random bytes used to generate this initial root must include the standard security margin λ.

Starting from the root node, the *Root Value* is divided by the modulus of the underlying level ($n^{2^{d-1}}$). This division operation yields a quotient and a remainder, which become the right and left child nodes, respectively. This "splitting" process is iteratively applied to all nodes level by level, doubling the number of nodes at each step. When the process reaches the final level, it produces exactly k leaf nodes. The mathematical properties of the division guarantee that these k values are strictly less than n, and they directly form a chunk of the output array.

To generate the complete array of l elements, this entire tree traversal is repeated. Specifically, the algorithm computes $c = \lfloor l/k \rfloor$ full trees. If l is not a perfect multiple of k, a final tree is evaluated to generate a smaller "tail", resulting in a total of $c + 1$ tree computations.

To better understand the splitting process, let us consider a practical example.

Example 2.2 (Chunked Tree generation with $n = 7$ and $l = 8$)**.** *Suppose we want to*

generate an array of $l = 8$ elements modulo $n = 7$. First, the algorithm computes the quadratic powers of the target modulus:

- *Level 0:* $n = 7$
- *Level 1:* $n^2 = 49$
- *Level 2:* $n^4 = 2401$
- *Level 3:* $n^8 = 5764801$

Since $l = 8$, we need a tree with $k = 8$ leaves. The depth of our tree will be $d = \log_2(8) = 3$. The root modulus is therefore $n^{2^3} = n^8 = 5764801$, which easily fits in a 64-bit register. The PRNG generates a sufficiently large random sequence, which is then reduced modulo n^8. Let us assume the resulting valid Root Value is 3456789*.*

The decoding phase operates as follows:

- ***Level 3 to Level 2 (Division by** $n^4 = 2401$**):** The root value is divided by the modulus of the level below.*

$$3456789/2401 = 1439 \quad (\textit{Right Child})$$

$$3456789 \bmod 2401 = 1750 \quad (\textit{Left Child})$$

- ***Level 2 to Level 1 (Division by** $n^2 = 49$**):** The process is repeated for both children.*
  - *Left node (*1750*):* $1750/49 = 35$ *(Right),* $1750 \bmod 49 = 35$ *(Left).*
  - *Right node (*1439*):* $1439/49 = 29$ *(Right),* $1439 \bmod 49 = 18$ *(Left).*
- ***Level 1 to Level 0 (Division by** $n = 7$**):** The four nodes are divided by 7 to extract the final $k = 8$ leaf elements.*
  - *Node* 35*:* $35/7 = 5$ *(Right),* $35 \bmod 7 = 0$ *(Left).*
  - *Node* 35*:* $35/7 = 5$ *(Right),* $35 \bmod 7 = 0$ *(Left).*
  - *Node* 18*:* $18/7 = 2$ *(Right),* $18 \bmod 7 = 4$ *(Left).*
  - *Node* 29*:* $29/7 = 4$ *(Right),* $29 \bmod 7 = 1$ *(Left).*

The final extracted chunk is the sequence of the 8 leaves: $[0, 5, 0, 5, 4, 2, 1, 4]$*. As expected, all generated elements are strictly bounded by* $0 \leq x < 7$*. A graphical representation of this specific tree traversal is shown in Figure 2.3.*

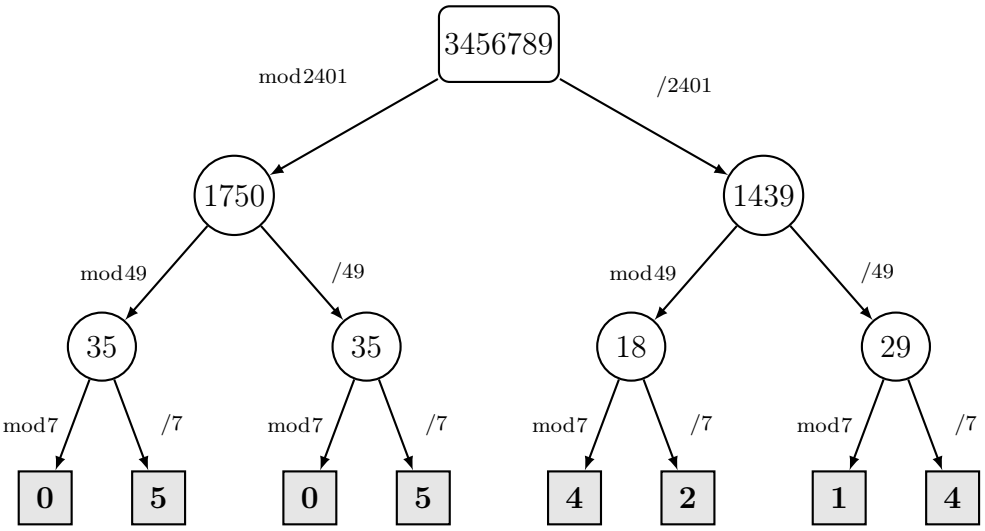

Figure 2.3: Visual representation of the decoding tree of Example 2.2

Optimization Similar to the Chunked variant (Section 2.3.2), the computational performance of this method can be enhanced by taking advantage of AVX2 intrinsics. The algorithm can process four independent trees simultaneously, reducing the overall execution time.

Security Analysis The Chunked Tree Arithmetic Decoding method maintains constant-time execution. However, much like the standard Chunked variant, it involves an entropy consumption trade-off. Because the decoding process is split across c trees, the algorithm requires to generate c Root Values. Consequently, the security margin λ must be enforced c times (once per root), resulting in a higher total requirement of pseudo-random bits compared to the monolithic approach.

Complexity The computational cost of this method depends on the value of the target modulus n. Specifically, this method is effective with small values of n. This is because the number of leaves k in each tree is determined by the highest quadratic power of n that fits within a 64-bit register.

2.3.4. Rejection Sampling Method

The Rejection Sampling method is a non-constant-time approach for solving Problem 2.2. This method is widely employed by the NIST Post-Quantum Cryptography (PQC) Round 2 additional digital signature candidates [36], such as CROSS [8], SDitH [4], and QR-UOV [5].

Input: `prng_state`, n (modulus), l (num elements), λ (security margin)

Output: a (l-element array modulo n)

Data: `nodes`, `next_nodes`: Arrays for tree traversal; `mod_level`: Array of moduli

function `TreeGenArrayMod(prng_state, n, l, λ):`
- $d \leftarrow 0$
- `mod_level`$[0] \leftarrow n$
- **while** `mod_level`$[d]^2 \leq 2^{64}$ ***and*** $2^d < 32$ **do**
  - `mod_level`$[d+1] \leftarrow$ `mod_level`$[d]^2$
  - $d \leftarrow d + 1$
- $k \leftarrow 2^d$ `// Number of leaves per tree`
- $c \leftarrow \lfloor l/k \rfloor$ `// Number of full trees`
- `tail` $\leftarrow l \bmod k$
- $a \leftarrow$ empty array of size l
- `idx` $\leftarrow 0$
- `total_trees` $\leftarrow c$
- **if** `tail` > 0 **then**
  - `total_trees` $\leftarrow c + 1$
- **for** $t \leftarrow 1$ **to** `total_trees` **do**
  - `root_val` $\leftarrow$ GenerateRoot(`prng_state`, `mod_level`$[d], \lambda$)
  - `nodes` $\leftarrow$ [`root_val`]
  - **for** `level` $\leftarrow d - 1$ ***downto*** 0 **do**
    - `next_nodes` $\leftarrow$ []
    - `divisor` $\leftarrow$ `mod_level`[`level`]
    - **for** ***each*** `val` ***in*** `nodes` **do**
      - `next_nodes`.append(`val` mod `divisor`) `// Left child`
      - `next_nodes`.append($\lfloor$`val`/`divisor`$\rfloor$) `// Right child`
    - `nodes` $\leftarrow$ `next_nodes`
  - `elements_to_extract` $\leftarrow k$
  - **if** $t =$ `total_trees` ***and*** `tail` > 0 **then**
    - `elements_to_extract` $\leftarrow$ `tail`
  - **for** $i \leftarrow 0$ **to** `elements_to_extract` $- 1$ **do**
    - a[`idx`] $\leftarrow$ `nodes`$[i]$
    - `idx` $\leftarrow$ `idx` $+ 1$
- **return** a

Algorithm 2.3: Chunked Tree Arithmetic Decoding

Base Idea Algorithm 2.4 shows the pseudo-code of the Rejection Sampling method. The algorithm starts by computing the minimum number of bits $d = \lceil \log_2(n) \rceil$ required to represent the target modulus n. Following this, the algorithm requests exactly d bits from the PRNG to form a random integer candidate x, such that $0 \leq x < 2^d$. This candidate is accepted and added to the output array only if $x < n$; otherwise, it is rejected and discarded. This generation and check process is repeated until exactly l

valid elements are successfully generated.

Optimization A naive implementation of this method might consume a 32-bit word from the PRNG for each candidate, wasting significant entropy if d is small. To optimize the PRNG usage, the implemented algorithm employs a 64-bit *bit-reservoir*, which acts as an accumulator of random bits. For each iteration, exactly d bits are masked and extracted from the reservoir. When the reservoir runs low on bits, it is efficiently refilled with a new 32-bit block from a larger pre-computed PRNG buffer.

Security Analysis The main drawback of this method lies in the rejection phase. Because a conditional branch is executed each time a candidate is checked, this approach is inherently non-constant-time, since its execution time strictly depends on the number of discarded candidates. However, this non-constant-time property does not represent an actual vulnerability regarding the secrecy of the accepted candidates. Since all generated pseudo-random numbers are independent, an adversary cannot infer any useful information about the final secret data values.

Complexity and Parameters The computational demand of the Rejection Sampling method is highly dependent on the value of the target modulus n. Since each candidate is drawn uniformly from the interval $\left[0, 2^d - 1\right]$, the probability of accepting a candidate is $p = \frac{n}{2^d}$, while the probability of rejecting it is consequently $q = 1 - \frac{n}{2^d}$.

Because the algorithm requires l successful extractions, the exact probability of encountering x rejections before obtaining l accepted candidates is:

$$P(X = x) = \binom{x + l - 1}{x} \left(\frac{n}{2^d}\right)^l \left(1 - \frac{n}{2^d}\right)^x$$

The expected number of generated candidates (accepted and rejected) can be formulated as:

$$l \cdot \frac{2^d}{n}$$

Indeed, if n is a value very close to 2^d, the probability of rejection is extremely small, and the number of iterations approaches the optimal value l. Conversely, in the worst-case scenario where n is just slightly larger than 2^{d-1}, the rejection probability approaches 50%, causing the expected number of iterations to double.

Pre-Computation Variant Algorithm 2.5 shows the pseudo-code for a variant of the above-discussed Rejection Sampling. Instead of dynamically requesting bits from the

Input: `prng_state`, n (modulus), l (num elements)

Output: a (l-element array modulo n)

Data: `sub_buf`: 64-bit reservoir for random bits

function `RejectionGenArrayMod(prng_state, n, l)`:
- $a \leftarrow$ empty array of size l
- $d \leftarrow \lceil \log_2(n) \rceil$
- `mask` $\leftarrow 2^d - 1$
- `sub_buf` $\leftarrow 0$
- `bits_in_buf` $\leftarrow 0$
- `curr` $\leftarrow 0$
- **while** `curr` $< l$ **do**
  - `// Refill the reservoir if it lacks enough bits`
  - **if** `bits_in_buf` $< d$ **then**
    - $r \leftarrow$ GetNextRand32(`prng_state`)
    - `sub_buf` $\leftarrow$ `sub_buf` $\vee (r \ll$ `bits_in_buf`$)$
    - `bits_in_buf` $\leftarrow$ `bits_in_buf` $+ 32$
  - `candidate` $\leftarrow$ `sub_buf` $\wedge$ `mask`
  - `// Accept candidate only if less than modulus`
  - **if** `candidate` $< n$ **then**
    - a[`curr`] $\leftarrow$ `candidate`
    - `curr` $\leftarrow$ `curr` $+ 1$
  - `// Consume the evaluated bits`
  - `sub_buf` $\leftarrow$ `sub_buf` $\gg d$
  - `bits_in_buf` $\leftarrow$ `bits_in_buf` $- d$
- **return** a

Algorithm 2.4: Rejection Sampling Method

PRNG during the generation loop, this approach aims to generate all the required pseudo-random bits at the start of the algorithm.

To safely pre-allocate the randomness without risking entropy exhaustion during execution, an upper bound on the required bits must be established.

We must compute the total number of bits b such that the probability of successfully extracting at least l valid candidates is bounded by the cryptographic security parameter λ:

$$Pr(\text{valid candidates} \geq l) \geq 1 - 2^{-\lambda}$$

Since each candidate extraction consumes exactly $d = \lceil \log_2(n) \rceil$ bits, an initial buffer of b bits allows for a maximum of $t = \lfloor b/d \rfloor$ independent trials. As established in section 2.3.4, the probability of a candidate being accepted is $p = \frac{n}{2^d}$. Therefore, the total number of valid candidates S obtained from t trials follows a Binomial distribution

$S \sim \text{Binomial}(t, p)$.

To ensure the algorithm completes successfully without exhausting the pre-computed bits, the cumulative probability of obtaining fewer than l valid candidates must be smaller than the acceptable failure rate $2^{-\lambda}$:

$$\sum_{i=0}^{l-1} \binom{t}{i} p^i (1-p)^{t-i} \leq 2^{-\lambda}$$

By finding the minimum integer t that satisfies this inequality, the optimal pre-computed bit-length is strictly determined as $b = t \cdot d$.

In practical implementations, t, and consequently b, are pre-computed for specific sets of parameters (i.e., specific combinations of n, l, and λ) and stored in a look-up table.

2.4. Generation of Fixed Weight Binary Vectors

This section deals with the problem of generating Fixed Weight Binary Vectos. First, we introduce the definition of the weight of a vector:

Definition 2.3 (Hamming Weight of a vector)**.** *Let l be an integer such that $l > 0$, and let $\mathbf{v}$ be a binary vector of length l. The Hamming weight of $\mathbf{v}$, typically denoted by $w_H(\mathbf{v})$ or simply w, is defined as the number of non-zero coordinates within the vector.*

From Definition 2.3, we can define the Fixed Weight Binary Vector Selection Problem as follows:

Definition 2.4. *Let l, w be integers such that $0 < w \leq l$. We denote the set of all binary vectors of length l and weight w as $\mathbf{S}_l^w$.*

Problem 2.3 (Fixed Weight Binary Vector Selection Problem)**.** *Select, uniformly at random, a vector from the set $\mathbf{S}_l^w$.*

All the following methods aim to solve Problem 2.3.

2.4.1. RepeatedAND method

The RepeatedAND method, proposed by Nir Drucker and Shay Gueron [21], takes advantage of the statistical properties of the bitwise AND operation applied to uniformly distributed random binary vectors.

Input: `prng_state`, n (modulus), l (num elements), λ (security margin)

Output: a (an l-element array modulo n)

Data: `prng_buffer`: Array of pre-computed squeezed bits

```
function SecureRejectionGenArrayMod(prng_state, n, l, λ):
    a ← empty array of size l
    d ← ⌈log2(n)⌉
    mask ← 2^d − 1
    b ← GetRequiredBitsLUT(n, l, λ)
    prng_buffer[0 : b − 1] ← Squeeze(prng_state, b)
    sub_buf ← 0
    bits_in_buf ← 0
    curr ← 0
    while curr < l do
        // Refill the reservoir from the pre-computed array
        if bits_in_buf < d then
            if no bits left in prng_buffer then
                return Failure                          // Entropy Exhaustion
            r ← ReadNext32Bits(prng_buffer)
            sub_buf ← sub_buf ∨ (r ≪ bits_in_buf)
            bits_in_buf ← bits_in_buf + 32
        candidate ← sub_buf ∧ mask
        if candidate < n then
            a[curr] ← candidate
            curr ← curr + 1
        sub_buf ← sub_buf ≫ d
        bits_in_buf ← bits_in_buf − d
    return a
```

Algorithm 2.5: Pre-Computed Rejection Sampling Method

In this context, we denote the bitwise AND operation between two binary vectors of length l as $\mathbf{c} = \mathbf{a} \wedge \mathbf{b}$, where, for each position $i \in \{0, 1, \ldots, l-1\}$, the i-th bit is given by $\mathbf{c}[i] = \mathbf{a}[i] \wedge \mathbf{b}[i]$.

Lemma 2.4.1. *Let $\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{n-1}$ be n pseudo-random independent binary vectors of length l. Then we have the following:*

- $\mathbb{E}[w_H(\mathbf{a}_0)] = \mathbb{E}[w_H(\mathbf{a}_1)] = \cdots = \mathbb{E}[w_H(\mathbf{a}_{n-1})] = \frac{l}{2}$
- $\mathbb{E}[w_H(\mathbf{a}_0 \wedge \mathbf{a}_1)] = \mathbb{E}[w_H(\mathbf{a}_0 \wedge \mathbf{a}_2)] = \cdots = \mathbb{E}[w_H(\mathbf{a}_{n-2} \wedge \mathbf{a}_{n-1})] = \frac{l}{4}$
- $\mathbb{E}[w_H(\mathbf{a}_0 \wedge \mathbf{a}_1 \wedge \mathbf{a}_2)] = \cdots = \mathbb{E}[w_H(\mathbf{a}_{n-3} \wedge \mathbf{a}_{n-2} \wedge \mathbf{a}_{n-1})] = \frac{l}{8}$
- $\ldots$

- $\mathbb{E}[w_H(\mathbf{a}_0 \wedge \mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \cdots \wedge \mathbf{a}_{n-1})] = \frac{l}{2^n}$

Base idea Algorithm 2.6 shows the pseudo-code of the RepeatedAND method. The core idea of the algorithm is to repeatedly apply the bitwise AND operation to newly generated pseudo-random vectors. As seen in Lemma 2.4.1, at each iteration, the expected weight of the vector halves. This iteration continues until the weight of the resulting vector is less than or equal to the number of '1's required (w_r). Finally, the candidate '1's are merged into the final output ($\mathbf{a}$), which acts as an accumulator, via a bitwise OR operation. If the weight $w_H(\mathbf{a})$ is still less than the desired weight w, the process is repeated.

Optimization To avoid generating '1's in positions that are already asserted in the accumulator $\mathbf{a}$, the candidate vector is masked using the bitwise negation of $\mathbf{a}$ ($\neg\mathbf{a}$).

Furthermore, another efficiency improvement can be obtained after the generation of a pseudo-random vector $\mathbf{a}_j$. Since $\mathbf{a}_j$ and $\neg\mathbf{a}_j$ are both valid binary vectors, it is possible to choose the one that preserves the highest number of '1's without exceeding the desired weight w_r.

Security Analysis This method is not constant-time. The number of iterations executed by the algorithm depends on the pseudo-random vectors provided by the PRNG.

In a best-case scenario, the generated candidate reaches exactly w ones, allowing the algorithm to end after a single iteration. In less favorable cases, the sequence of pseudo-random vectors will force the algorithm to perform multiple iterations until the target weight is achieved.

Consequently, an adversary observing the CPU clock cycles can deduce the number of times the PRNG is invoked. However, as long as the values of the generated pseudo-random vectors are not exposed, the attacker cannot obtain any useful information about the intermediate secret vectors that compose the final fixed-weight binary vector.

Complexity and Parameters The overall execution time of the RepeatedAND method depends on the parameters w and l.

As seen, the algorithm requires generating a new pseudo-random vector of length l at each inner iteration, meaning that the entropy consumption is directly proportional to the total number of PRNG invocations. Consequently, the expected number of iterations j required to generate the first intermediate candidate is $\mathbb{E}[j] \approx \lfloor \log_2(l/w) \rfloor$. Furthermore, the outer accumulator loop is expected to execute at most $\lceil \log_2 w \rceil$ times. The total

Input: `prng_state`, l, w

Output: $\mathbf{a}$ (an l-bit binary vector with weight w)

```
1  function GenVector(prng_state, l, w):
2      w_r ← w
3      a[l − 1 : 0] ← 0^l
4      repeat
5          j ← 0
6          a_j ← GetRand(prng_state, l)
7          ā ← a_j ∧ ¬a
8          repeat
9              j ← j + 1
10             a_j ← GetRand(prng_state, l)
11             if w_H(ā ∧ a_j) ≤ w_r and w_H(ā ∧ ¬a_j) ≤ w_r then
12                 ā ← max(ā ∧ a_j, ā ∧ ¬a_j)
13             else
14                 if w_H(ā ∧ a_j) ≤ w_r then
15                     ā ← ā ∧ a_j
16                 else
17                     ā ← ā ∧ ¬a_j
18         until w_H(ā) ≤ w_r
19         a ← a ∨ ā
20         w_r ← w − w_H(a)
21     until w_r = 0
22     return a
```

Algorithm 2.6: RepeatedAND method

amount of required randomness is inherently variable, but can be statistically bounded by multiplying the expected total number of iterations by the l bits consumed at each step.

The algorithm is particularly effective when the desired vector is "dense", specifically when the ratio $w/l \geq 0.2$. This is because generating a vector with a very small number of ones requires a high number of bitwise AND operations to logically halve the weight, as discussed in Lemma 2.4.1.

For this reason, alternative approaches, such as the Rejection Method (Section 2.4.3), are preferred when the $\frac{w}{l}$ ratio is low.

2.4.2. Sorting Method

The Sorting method, an approach that was proposed for the NTRU LPrime cryptosystem [17], employs a constant-time and efficient integer sorting algorithm, such as `djbsort` [15],

to uniformly shuffle the positions of ones and zeros within a binary vector.

Base idea Algorithm 2.7 shows the pseudo-code of the Sorting method. The algorithm begins by generating an array of l random 32-bit integers. Next, the Least Significant Bit (LSB) of the first w integers is set to `1`, while the LSB of the remaining $l - w$ integers is set to `0`.

The 32-bit random numbers act as keys for the efficient, constant-time sorting algorithm. This operation successfully shuffles the elements, effectively randomizing the positions of the embedded LSBs.

Finally, the LSBs are extracted to compose the output vector. Since the initial numbers were uniformly distributed, the resulting output will be a random vector with length l and weight w.

Security Analysis The security of the Sorting method is entirely dependent on the security of the employed sorting algorithm. If a constant-time sorting algorithm is used, such as `djbsort` [15], then the fixed weight generation process is guaranteed to be a constant-time algorithm. Similarly, if a non-constant-time sorting algorithm is employed, the Sorting method will be vulnerable to timing side-channel attacks. An adversary observing the CPU clock cycles can deduce information about the keys used by the unsafe sorting algorithm.

Complexity and Parameters The overall computational complexity is dominated by the efficiency of the sorting algorithm employed, which typically requires $O(l \log(l))$ operations.

From a randomness point of view, the algorithm generates one 32-bit pseudo-random key for each of the l coordinates, requiring a total of $32 \cdot l$ bits.

However, two or more pseudo-random sorting keys can be identical. The probability of these collisions can be bounded using the Birthday Paradox. If a collision occurs between a key holding a non-zero coordinate and a key holding a zero, the sorting algorithm introduces a statistical bias. A way to reduce the risk of collisions is to expand the length of the keys.

The main advantage of this approach is that its execution time is independent of the target weight w. On the other hand, since the computational complexity of the sorting algorithms scales with the number of elements to be sorted (in our case the target length l), this approach is efficient with small values of l.

Input: `prng_state`, l, w

Output: **a** (an l-bit binary vector with weight w)

Data: **b**: array of $32 \cdot l$ bits, SORT operates on an array of 32-bit integers

```
1  function GenVector(prng_state, l, w):
2      b[32l − 1 : 0] ← GetRand(prng_state, 2^{32l} − 1)
3      for i ← 0 to w − 1 do
4          b[32i] ← 1
5      for i ← w to l − 1 do
6          b[32i] ← 0
7      Sort(b, l)
8      for i ← 0 to l − 1 do
9          a[i] ← b[32i]
10     return a
```

Algorithm 2.7: Sorting method

2.4.3. Rejection Method

The Rejection method, employed for instance in NTRU Encrypt [28] and BIKE [7], is a non-constant-time method for solving Problem 2.3. Unlike the previously discussed methods, this approach requires the definition of a generator of random integers modulo l. These generated elements act as positional indices to manipulate the output binary vector.

Base idea Algorithm 2.8 shows the pseudo-code of the Rejection method. It starts by initializing the output vector with l zeros. At each iteration, a random integer $j \in \{0, 1, \ldots, l-1\}$ is generated; if the bit at position j in the output array is equal to `0`, it is set to `1`; otherwise the position is simply ignored.

The iterative process stops exactly when w bits in the output array are set to `1`.

Optimization Instead of generating a single integer j per iteration, the RandShaper library tracks the number of ones that still need to be set in the output array, and generates an entire array of integers modulo l in bulk, allowing the loop to iterate directly over this set. This is due to the fact that generating an array of x elements in a single batch is computationally more efficient than invoking the PRNG to generate a single element x separate times.

Furthermore, the RandShaper framework offers the opportunity to select any of the algorithms presented in Section 2.3 to perform the bulk generation of these numbers modulo l.

Input: `prng_state`, l, w

Output: $\mathbf{a}$ (l-bit binary vector with weight w)

```
1  function GenVector(prng_state, l, w):
2      a[l − 1 : 0] ← 0^l
3      w_r ← w            // Number of non-zero coordinates remaining to be set
4      while w_r > 0 do
5          r ← GetRandArray(prng_state, l, w_r)
6          for i ← 0 to w_r − 1 do
7              j ← r[i]
8              if a[j] = 0 then
9                  a[j] ← 1
10                 w_r ← w_r − 1
11     return a
```

Algorithm 2.8: Rejection method

Security Analysis The Rejection method is inherently non-constant-time. This is due to the branch created each time the bit at position j is checked and the variable number of iterations required to reach the target weight w, depending on the occurrence of collisions. However, as long as the generated pseudo-random indices are not exposed, an adversary cannot infer any information about the positions of the non-zero coordinates in the final binary vector $\mathbf{a}$.

Complexity and Parameters The overall computational complexity of this method is directly proportional to the efficiency of the underlying PRNG used to generate the arrays of integers modulo l.

The algorithms involves a variable number of PRNG invocations due to the possibility of encountering collisions. As more bits are set to one, the probability of selecting an index that already contains a 1 increases.

Furthermore, the algorithm is highly effective for large values of target length l and small values of the target weight w, i.e., when generating sparse vectors, since the probability of encountering a collision is low in such scenarios.

2.4.4. Fisher-Yates Method

The Fisher-Yates Method [22, 23] is a classical algorithm widely adapted (e.g., in CROSS [8]) for generating fixed-weight binary vectors, operating in variable time.

Base idea Algorithm 2.10 shows the pseudo-code of the Fisher-Yates approach. The algorithm starts by initializing the output array with w ones, followed by $l - w$ zeros.

To uniformly randomize the vector, the algorithm iterates over the array, swapping the element at the current index with another element chosen at random from the remaining unshuffled positions. Because standard PRNGs output uniformly distributed bit vectors, a rejection sampling check is performed in order to ensure that the candidate position value falls within a valid range (i.e., the length of the unshuffled positions).

Optimization To minimize wasted entropy, a 64-bit reservoir register is employed to save the random bits generated by the PRNG. Instead of consuming and discarding a 32-bit integer at each iteration, the algorithm computes the number of bits required to represent the current possible range. Therefore, only those bits are extracted and the 64-bit register is refilled when it is necessary.

Security Analysis The primary drawback of the Fisher-Yates method lies in the rejection sampling. Each iteration of the algorithm checks if the pseudo-randomly generated integer falls within a range, creating a conditional branch. This introduces a data-dependent execution path, making the algorithm non-constant time. As long as the pseudo-random integers are not exposed, an attacker cannot infer any information about the positions of the non-zero coordinates within the final vector $\mathbf{a}$.

Complexity and Parameters The computational demand of this algorithm is linear with respect to the computational demand of the PRNG, the number of times the swap is performed, and the target length l.

From an entropy perspective, the consumed randomness is highly dependent on the number of rejected indices during the sampling phase.

Since l iterations are performed and l possible swaps can be executed, this algorithm works best with small values of l.

Pre-Computation Variant Algorithm 2.10 shows the pseudo-code of a variant of the Fisher-Yates method. While the optimized method presented in Algorithm 2.9 calls the PRNG every time the 64-bit register reservoir runs out of bits, another variant, which generates enough pseudo-random bits at the start of the algorithm, can be proposed, establishing a strict mathematical margin of non-failure.

Firstly, an upper bound on the required bits must be established to safely pre-allocate the randomness without risking entropy exhaustion during execution.

For this reason, the total number of bits b must be computed such that the probability of successfully completing the entire sequence of l swaps is bounded by the cryptographic

Input: `prng_state`, l, w

Output: **a** (l-bit binary vector with weight w)

Data: `sub_buf`: 64-bit reservoir for random bits

function `GenVector(prng_state, l, w)`:
- $\mathbf{a}[0 : w-1] \leftarrow 1^w$
- $\mathbf{a}[w : l-1] \leftarrow 0^{l-w}$
- `sub_buf` $\leftarrow 0$
- `bits_in_buf` $\leftarrow 0$
- `curr` $\leftarrow 0$
- **while** `curr` $< l$ **do**
  - `modulus` $\leftarrow l -$ `curr`
  - `bits_req` $\leftarrow \lceil \log_2(\texttt{modulus}) \rceil$
  - `// Refill the sub-buffer if it lacks enough bits`
  - **if** `bits_in_buf` $<$ `bits_req` **then**
    - $r \leftarrow$ GetNextRand32(`prng_state`)
    - `sub_buf` $\leftarrow$ `sub_buf` $\vee (r \ll$ `bits_in_buf`$)$
    - `bits_in_buf` $\leftarrow$ `bits_in_buf` $+ 32$
  - `mask` $\leftarrow 2^{\texttt{bits\_req}} - 1$
  - `candidate_pos` $\leftarrow$ `sub_buf` $\wedge$ `mask`
  - `// Rejection sampling: accept only if within modulus`
  - **if** `candidate_pos` $<$ `modulus` **then**
    - `dest` $\leftarrow$ `curr` $+$ `candidate_pos`
    - Swap($\mathbf{a}$[`curr`], $\mathbf{a}$[`dest`])
    - `curr` $\leftarrow$ `curr` $+ 1$
  - `// Consume the used bits`
  - `sub_buf` $\leftarrow$ `sub_buf` $\gg$ `bits_req`
  - `bits_in_buf` $\leftarrow$ `bits_in_buf` $-$ `bits_req`
- **return a**

Algorithm 2.9: Fisher-Yates method

security parameter λ:

$$\Pr(\text{successful generation}) \geq 1 - 2^{-\lambda}$$

The Fisher-Yates method shrinks the target modulus m at each iteration; each candidate extraction consumes $d_m = \lceil \log_2(m) \rceil$ bits, and the probability of a candidate being accepted is $p_m = \frac{m}{2^{d_m}}$. Denoting $\mathcal{T}_m$ as the random variable representing the number of independent trials required to achieve a single success for modulus m, $\mathcal{T}_m$ follows a Geometric distribution $\mathcal{T}_m \sim \text{Geometric}(p_m)$.

To ensure the algorithm completes successfully without exhausting the pre-computed bits, the cumulative probability that the random variable of the total bit consumption $\mathcal{B}_{total} = \sum_{m=1}^{l} \mathcal{T}_m \cdot d_m$ exceeds the initial buffer of b bits must be smaller than the acceptable

Input: `prng_state`, l, w, λ (security margin)

Output: **a** (an l-bit binary vector with weight w)

Data: `prng_buffer`: Array of pre-computed squeezed bits

function `GenVector(prng_state, l, w, λ):`
- $\mathbf{a}[0 : w-1] \leftarrow 1^w$
- $\mathbf{a}[w : l-1] \leftarrow 0^{l-w}$
- $b \leftarrow$ GetRequiredBitsLUT(l, w, λ)
- `prng_buffer`$[0 : b-1] \leftarrow$ Squeeze(`prng_state`$, b)$
- `sub_buf` $\leftarrow 0$
- `bits_in_buf` $\leftarrow 0$
- `curr` $\leftarrow 0$
- **while** `curr` $< l$ **do**
  - `modulus` $\leftarrow l -$ `curr`
  - `bits_req` $\leftarrow \lceil \log_2(\texttt{modulus}) \rceil$
  - `// Refill the sub-buffer from the pre-computed array`
  - **if** `bits_in_buf` $<$ `bits_req` **then**
    - **if** *no bits left in* `prng_buffer` **then**
      - **return Failure** `// Entropy Exhaustion`
    - $r \leftarrow$ ReadNext32Bits(`prng_buffer`)
    - `sub_buf` $\leftarrow$ `sub_buf` $\vee (r \ll$ `bits_in_buf`$)$
    - `bits_in_buf` $\leftarrow$ `bits_in_buf` $+ 32$
  - `mask` $\leftarrow 2^{\texttt{bits\_req}} - 1$
  - `candidate_pos` $\leftarrow$ `sub_buf` $\wedge$ `mask`
  - **if** `candidate_pos` $<$ `modulus` **then**
    - `dest` $\leftarrow$ `curr` $+$ `candidate_pos`
    - Swap(**a**[`curr`], **a**[`dest`])
    - `curr` $\leftarrow$ `curr` $+ 1$
  - `sub_buf` $\leftarrow$ `sub_buf` $\gg$ `bits_req`
  - `bits_in_buf` $\leftarrow$ `bits_in_buf` $-$ `bits_req`
- **return a**

Algorithm 2.10: Pre-Computed Fisher-Yates method

failure rate $2^{-\lambda}$:

$$\Pr(\mathcal{B}_{total} > b) \leq 2^{-\lambda}$$

The optimal pre-computed bit-length b is strictly determined by finding the minimum admissible value.

In practical implementations, b is pre-computed for specific sets of parameters (i.e., specific combinations of l, w, and λ) and stored in a look-up table.

2.4.5. Sendrier's Fisher-Yates Method

Sendrier's variant of the Fisher-Yates shuffle [41] was specifically designed to securely sample constant-weight vectors for the BIKE cryptosystem [7]. The main property of this approach is that, unlike the classical formulation (Section 2.4.4), it is secure against timing and cache side-channel attacks, trading perfect uniformity for a constant-time execution, introducing a negligible statistical bias in the distribution of the output set.

Base idea Algorithm 2.11 shows the pseudo-code of Sendrier's Fisher-Yates method. The main objective of the algorithm is to randomly generate w possible candidate indices, representing the corresponding positions where the output bits must be set to `1`.

To map the uniformly distributed 32-bit output of the PRNG to an interval $[0, \text{range})$, the algorithm employs the fast integer multiplication reduction proposed by Lemire [29], for which the rejection phase typically required to eliminate modulo bias is omitted.

Because indices are generated independently without rejection, collisions can occur (i.e., the same random position might be extracted twice). To resolve this, the algorithm iterates backwards through the generated array, and if a collision is detected, the duplicate element is replaced with the index of the current iteration round.

Finally, a constant-time vector reconstruction is performed to set the corresponding positions to `1`.

Security Analysis As previously discussed, this method is inherently constant-time. An adversary monitoring CPU cycles cannot infer any information about the randomly generated candidate positions.

However, this method introduces a statistical bias in the distribution of the output set. Specifically, the absolute maximal probability over all possible outputs is:

$$\pi_{max} = w! \prod_{i=0}^{w-1} \frac{1}{l-i} \left(1 + \frac{(l-i) - (2^{32} \bmod (l-i))}{2^{32}}\right)$$

and the absolute minimal probability over all possible outputs is:

$$\pi_{min} = w! \prod_{i=0}^{w-1} \frac{1}{l-i} \left(1 - \frac{2^{32} \bmod (l-i)}{2^{32}}\right).$$

To better understand the practical impact of this bias, it is possible to analyze a concrete scenario with a 128-bit security margin, such as the parameters of the BIKE Category 1

cryptosystem. In this scheme, the target dimensions for the error vector are $l = 24646$ and $w = 134$. By evaluating the maximal relative deviation from the perfectly uniform distribution π_{uni}, we can compute the factor $\tau_{max} = \frac{\pi_{max}}{\pi_{uni}}$. As analyzed by Sendrier [41], for these specific dimensions, this bound strictly evaluates to $\tau_{max} \approx 1.00038$. This means that the most probable output vector is generated with a probability only 0.038% higher than a perfectly uniform random selection.

Sendrier demonstrated [41] that this deviation is close enough to the perfect uniform distribution, resulting in a negligible impact on the overall cryptographic security.

Complexity and Parameters The computational complexity of Sendrier's method strictly depends on the target weight w and the target length l. Specifically, the collision resolution phase requires $O(w^2)$ operations, as the algorithm iterates backwards and compares each candidate index against all previously resolved positions. Furthermore, the vector construction phase has a complexity of $O(l \cdot w)$, since the algorithm requires scanning the entire output vector of length l and masking it against all w generated indices. Consequently, this method is practical for small values of w.

From a randomness-consumption aspect, w 32-bit indices are generated, so the PRNG consumes a total of $32 \cdot w$ pseudo-random bits.

2.5. Generation of Permutations

This chapter deals with the problem of generating and securely applying a random permutation to an array of l elements.

Definition 2.5 (Permutation Array). *Given a source array* $[0, 1, \ldots, l-1]$*, a permutation array* π *is a rearrangement of the elements from the source array into a new, unique order, containing all original elements exactly once.*

Problem 2.4 (Permutation Generation Problem). *Select, uniformly at random, a permutation array* π *of length* L*.*

Problem 2.5 (Secure Shuffling Problem). *Given an array* $\mathbf{a}$ *and a randomly generated permutation* π*, compute the shuffled array* $\mathbf{a}'$ *such that* $\mathbf{a}'[i] = \mathbf{a}[\pi[i]]$*, without leaking any information about* π *or* $\mathbf{a}$ *through side-channels.*

The presented method offers a strictly constant-time approach to both Problem 2.4 and Problem 2.5. By taking advantage of sorting networks and routing algorithms, we eliminate any data-dependent branches or memory accesses. The following sections detail the

Input: `prng_state`, l, w

Output: **a** (an l-bit binary vector with weight w)

Data: **p**: Array of size w storing generated positions

function `GenVector(prng_state, l, w):`
- $\mathbf{a}[l-1:0] \leftarrow 0^l$
- $\mathbf{r} \leftarrow$ GetRandArray32(`prng_state`, w)
- `// Phase 1: Generate positions`
- **for** $i \leftarrow 0$ **to** $w-1$ **do**
  - `range` $\leftarrow l - i$
  - `// Lemire's reduction`
  - $\mathbf{p}[i] \leftarrow i + \lfloor(\mathbf{r}[i] \times$ `range`$)/2^{32}\rfloor$
- `// Phase 2: Constant-time collision resolution`
- **for** $i \leftarrow w-1$ ***downto*** 0 **do**
  - **for** $j \leftarrow i+1$ **to** $w-1$ **do**
    - $\mathbf{p}[j] \leftarrow$ CT_Select$(\mathbf{p}[j] == \mathbf{p}[i], i, \mathbf{p}[j])$
- `// Phase 3: Constant-time vector construction`
- **for** $i \leftarrow 0$ **to** $w-1$ **do**
  - `pos` $\leftarrow \mathbf{p}[i]$
  - **for** $k \leftarrow 0$ **to** $l-1$ **do**
    - `// Set` $\mathbf{a}[k] = 1$ `if` $k ==$ `pos`
    - $\mathbf{a}[k] \leftarrow \mathbf{a}[k] \vee$ CT_IsEqual$(k,$ `pos`$)$
- **return a**

Algorithm 2.11: Sendrier's Fisher-Yates method

mathematical foundations and the implementation of this approach, which is based on the usage of Beneš networks [13].

2.5.1. Beneš Networks

An in-place Beneš network [13] is a routing topology capable of applying any arbitrary permutation to an array of $n = 2^m$ elements, $m \in \mathbb{N}$, using $2m - 1$ stages. Each stage consists of $\frac{n}{2}$ 2×2 switches, enabling the network to perform $\frac{n}{2}$ conditional swaps in parallel. Every switch takes two inputs, yields two outputs, and is regulated by a single control bit: if the control bit is zero, the two inputs are left unchanged; if it is one, they are swapped.

The conditional swap operations are evaluated between pairs of elements located at fixed distances. Specifically, the first stage operates on elements at distance 1, the second stage at distance 2, doubling step by step up to distance 2^{m-1} in the middle stage. Afterward, the distance halves at each subsequent stage, decreasing back to 1 in the final stage.

The primary challenge of performing permutation operations is to evaluate the network

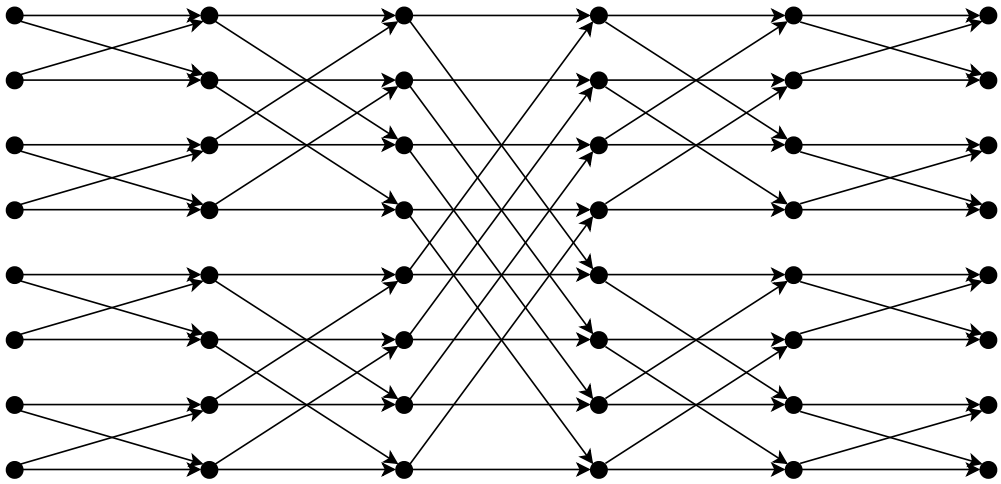

Figure 2.4: Data flow in an in-place Beneš network for permuting 8 inputs.

in a constant-time manner. A naive permutation application, such as $\texttt{out}[\texttt{i}] = \texttt{in}[\pi[\texttt{i}]]$, requires secret-dependent RAM accesses, making the implementation vulnerable to timing side-channel attacks. The adoption of Beneš networks mitigates this issue: since the pairs of indices evaluated at each stage are fixed and predetermined by the network's topology, the memory access pattern becomes completely deterministic and independent of the applied permutation. Finally, to avoid data-dependent branching, the conditional swaps are executed using constant-time bitwise operations (i.e., masks), effectively acting as software multiplexers.

Figure 2.4 shows an example of a Beneš network for the permutation of $n = 8$ inputs.

2.5.2. Generation of Control Bits for Beneš Networks

Daniel J. Bernstein proposed a set of verified formulas [16] to compute the control bits of a target permutation π in strictly constant time. The algorithm is based on the canonical looping algorithm by Waksman and Stone [42] and its parallel variant by Nassimi and Sahni [30]. A core requirement to maintain the constant-time property is the employment of a secure and possibly efficient sorting algorithm, such as `djbsort` [15].

The Logic Behind Control Bits Generation

From a high-level perspective, the generation of control bits relies on locating routing dependencies between the switches of the first stage F and the last stage L. These dependencies arise from the topological constraints of the Beneš network: if two inputs share the same switch in the first stage, they must be routed to opposite sub-networks

(Top and Bottom). The same reasoning applies to two outputs arriving at the same final switch: one output must come from the Top network, while the second output must come from the Bottom one.

To solve these constraints, the algorithm employs a tracing function called `XbackXforth` [16], which maps the physical dependencies between the network's nodes by building a "chain of dependencies".

The `XbackXforth` function can be formulated as:

$$z = \textsc{XbackXforth}(x) = \pi(\pi^{-1}(x \oplus 1) \oplus 1)$$

where x is an output node. By analyzing this function step by step:

- $x \oplus 1$: Look at $x \oplus 1$ (the neighbor wire at the output switch). If x comes from the Top network, $x \oplus 1$ must come from the Bottom one.
- $\pi^{-1}(x \oplus 1)$: Trace back to the input side. Let input u be the source of $x \oplus 1$. Since the destination is the Bottom network, input u must be routed to the Bottom network.
- $\pi^{-1}(x \oplus 1) \oplus 1$: Look at $u \oplus 1$ (the partner wire at the input switch). If u goes to the Bottom network, partner $u \oplus 1$ must go to the Top network.
- $\pi(\pi^{-1}(x \oplus 1) \oplus 1)$: Trace forward: call z the output where input $u \oplus 1$ lands. Since $u \oplus 1$ is routed to the Top network, output z must come from the Top network as well.

From this, it is possible to state that outputs x and z are inextricably linked. They must always have the same routing decision.

By repeatedly applying this function, the algorithm groups the dependent elements into disjoint cycles. After tracing these cycles, the `cyclemin` function is introduced. This function aims to find the minimum element of the cycle, which will act as a "leader" of the cycle. A naive implementation of this algorithm would require a linear visit of all the elements of the cycle ($O(N)$). An efficient and parallel way to solve this problem is the usage of the well-known Pointer Jumping algorithm, where each node scans the neighbors at exponential distances $(1, 2, 4, \ldots, 2^{m-1})$, finding the minimum value of the cycle in $O(\log_2 N)$ steps.

Bernstein proved that, for an element x:

$$\textsc{cyclemin}(x \oplus 1) = \textsc{cyclemin}(x) \oplus 1$$

This means that the leaders of two neighbor nodes are two consecutive numbers, implying that one is an odd number, while the other one is an even number. From this property, it is possible to assign the routing decisions to each input: if the leader is even, the node is sent to the Top network, otherwise it is sent to the Bottom one. Therefore, for each control bit of the switches in the First stage $(f_0, f_1, \ldots, f_{N/2-1})$ and in the Last stage $(l_0, l_1, \ldots, l_{N/2-1})$:

$$f_i = \text{CYCLEMIN}(\pi(2i)) \bmod 2$$

$$l_i = \text{CYCLEMIN}(2i) \bmod 2$$

Once the control bits for the outermost stages are determined, the algorithm proceeds with a divide-and-conquer strategy. The routing problem is decomposed into two independent sub-permutations of half the size, which correspond respectively to the top and bottom subnetworks. The `XbackXforth` tracing logic is then recursively applied to these smaller permutations. At each recursive step, the algorithm resolves the outermost available stages, progressively working through until it converges to the base case (i.e., a single switch) at the innermost middle stage, yielding the complete set of control bits.

Constant-Time Implementation via Sorting

While the recursive mathematical formulation effectively computes the control bits, translating it into a secure software implementation presents a significant challenge. The theoretical algorithm requires traversing cycles and following permutation mappings (e.g., evaluating $\mathbf{a}[\pi[i]]$ or inverting π), which translates to non-sequential, secret-dependent memory accesses in standard implementations.

To prevent timing side-channel leaks, the cycle traversal is entirely replaced by data-oblivious sorting operations. The core of this constant-time implementation is represented by the `composeinv` function [16], which safely computes the inverse composition of an array $\mathbf{v}$ with a permutation π, yielding $\mathbf{o}[j] = \mathbf{v}[\pi^{-1}(j)]$.

The operational flow of this data-oblivious evaluation is divided into three deterministic phases:

1. **Packing Phase**: The algorithm starts by packing the permutation indices and their corresponding target values into an array of 64-bit integers. Specifically, for each index $i \in \{0, \ldots, N-1\}$, the sorting key $(\pi[i])$ is shifted into the 32 most significant bits, while the payload $(\mathbf{v}[i])$ is stored in the 32 least significant bits.

2. **Sorting Phase**: A constant-time sorting algorithm, such as `djbsort` [15], is applied

to the packed array. By sorting the array in increasing order, the 64-bit integers are rearranged based on their most significant bits (i.e., the keys $\pi[i]$). Consequently, the element that moves to position j is exactly the one where $\pi[i] = j$.

3. **Unpacking Phase**: Finally, the payload is extracted by masking out the sorting keys, yielding the correctly routed values.

By employing this sequence of `composeinv`, the `XbackXforth` tracing logic is transformed into a series of linear scans and data-oblivious sorts, guaranteeing predictable memory access pattern.

2.5.3. Evaluation of the Beneš Network from Control Bits

Once the control bits are generated, the Beneš network must be evaluated to route the elements. The network can securely shuffle any generic payload of data (Problem 2.5).

The core routing engine evaluates the $2m - 1$ stages of the network iteratively as follows:

1. **Shift and Gap Calculation:** An outer loop, representing each stage index $i \in \{0, \ldots, 2m - 2\}$, is executed to evaluate the whole network. At each stage, the distance (or gap) between the paired elements grows by powers of two up to the middle stage, and then shrinks symmetrically. The shift amount s for stage i is dynamically computed without branching using a constant-time minimum function:

$$s = \min(i, 2m - 2 - i)$$

   The gap g between the two elements of a switch is then simply $g = 2^s$.

2. **Switch Mapping and Position Calculation:** In the inner loop (i.e., in each stage i), $n/2$ switches with index $j \in \{0, \ldots, n/2 - 1\}$ are processed in parallel. To evaluate the switch j, the algorithm must find the exact memory position p of the first element of the pair, which is achieved through bitwise logic and the use of the shift s and gap g values computed in the outer loop:

$$p = (j \wedge (g - 1)) + ((j \gg s) \ll (s + 1))$$

   The position of the second element of the pair is easily computed as $p + g$.

3. **Constant-Time Conditional Swap (CSWAP):** Once the elements $\mathbf{a}[p]$ and $\mathbf{a}[p+g]$ are identified, their specific routing bit $b \in \{0, 1\}$ is extracted from the packed control bits array. A conditional swap is then performed using a bitwise XOR ($\oplus$)

strategy to maintain the constant-time property. First, the control bit b is expanded into a 32-bit mask M using two's complement negation (i.e., $M = -b$), resulting in a mask of all zeros (`0x00000000`) if $b = 0$, and all ones (`0xFFFFFFFF`) if $b = 1$. Then, the conditional difference Δ is calculated:

$$\Delta = (\mathbf{a}[p] \oplus \mathbf{a}[p+g]) \wedge M$$

Finally, the difference is applied to both elements:

$$\mathbf{a}[p] \leftarrow \mathbf{a}[p] \oplus \Delta$$

$$\mathbf{a}[p+g] \leftarrow \mathbf{a}[p+g] \oplus \Delta$$

Table 3.1: Experimental setup used for the performance benchmarks.

Component	**Specification**
Operating System	Debian GNU/Linux 13 (trixie)
Linux Kernel	6.12.74+deb13+1-amd64
CPU Model	Intel® Xeon® W-2123 CPU @ 3.60GHz
Architecture	x86_64
Hardware Extensions	AVX2, AVX512, AES-NI
Available RAM	32 GB (31 GiB usable)
Compiler	GCC 14.2.0
Build System	CMake 3.25
C Standard	C11
Optimization Flags	`-O3 -march=native -mavx2 -maes`

3 | Experimental Evaluation

This chapter presents the benchmark results of the proposed implementations.

3.1. Experimental Setup

The setup used for the performed tests and result measurements is summarized in Table 3.1.

For counting CPU cycles, the `libcpucycles` microlibrary [14] is employed. For each benchmark, 20,000 iterations are performed: in each iteration, the number of CPU cycles is saved in an array, and the median is ultimately used as the evaluation metric; we do not use the average because, for certain methods, the best-case and worst-case performances differ significantly from the average case, for example in the RepeatedAND method (Section 2.4.1) and the rejection method (Section 2.4.3).

Figure 3.1 shows the code setup used to measure the CPU cycles across all 20,000 iterations.

For the entropy measurement (i.e., tracking how many bits from the PRNG are required

```
long long cycles[20000];
for(size_t i = 0; i < 20000; i++){
    long long t0 = cpucycles();
    method_to_be_tested(parameters);
    long long t1 = cpucycles();

    cycles[i] = t1 - t0;
}
```

Figure 3.1: The benchmark code setup

to generate the target object), counters are used and updated each time the squeeze operation is called.

The tested algorithms are the candidates of the NIST Round 2 for additional digital signatures.

The employed PRNG is the SHAKE-256 parallelized version that exploits the SIMD capabilities, which processes four Keccak operations simultaneously. Also, for the squeeze operations, the Fast-Aligned Version is employed.

Appendix A contains all the tables with the results obtained.

3.2. Experimental Results

3.2.1. Binary string generation results

Figure 3.2 shows the performance results of the primary and fastest PRNGs implemented in the RandShaper framework when squeezing various amounts of binary data, specifically for output sizes that are powers of 2.

An analysis of Figure 3.2 reveals that, for cryptographically relevant data sizes, the ChaCha20 implementation achieves the highest throughput among the evaluated PRNGs. It is followed in efficiency by the hardware-accelerated AES-256-NI-CTR-DRBG, the vectorized SHAKE-256-X4, and the standard SHAKE-256. Conversely, the purely software-based AES-256-CTR-DRBG exhibits the poorest performance, ranking last across the benchmark.

3.2.2. Modular array generation results

Regarding the generation of arrays of numbers modulo n, an object required by the majority of the algorithms, only moduli that are not a power of 2 are reported. This is

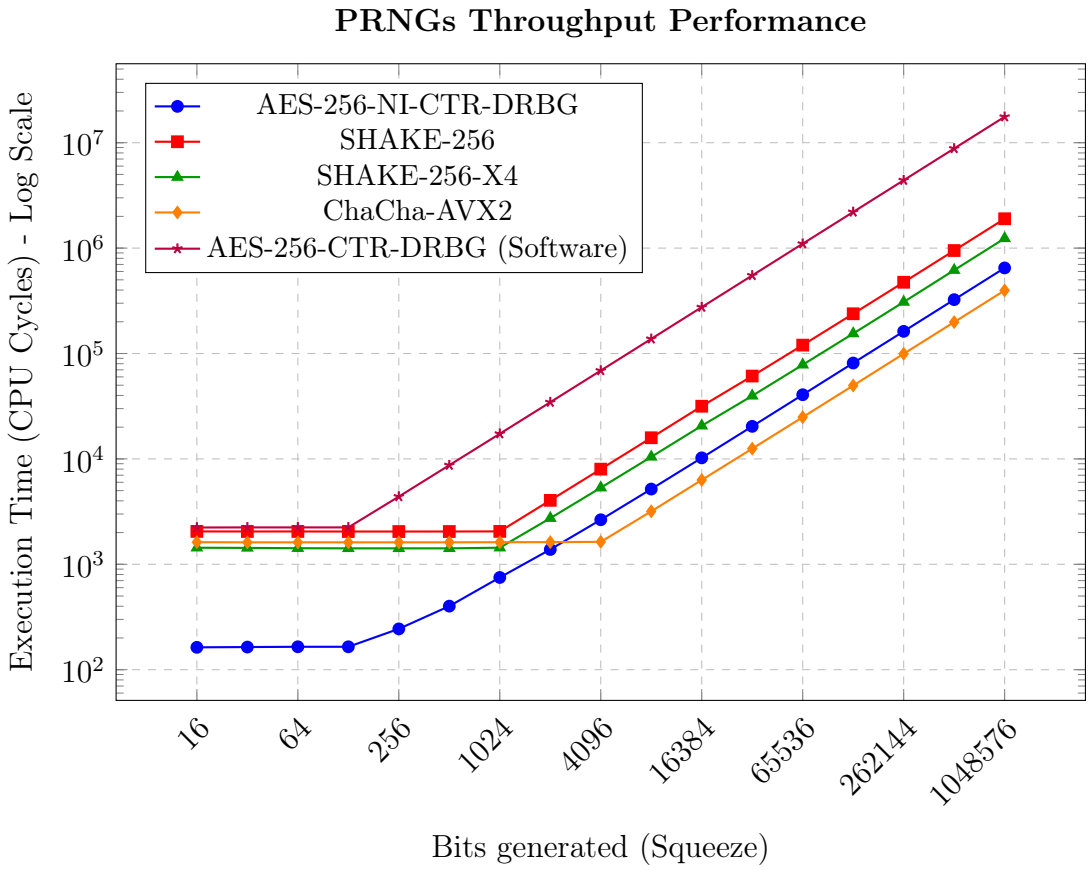

Figure 3.2: Performance comparison (CPU cycles) of the squeeze operation throughput for the evaluated PRNGs.

because, if $n = 2^x, x \in \mathbb{N}$, the binary string generator can be directly employed: for each number, an x-bit random string is directly generated.

Figures 3.3 to 3.5 illustrate the performance results for the modular array generation. For all the Arithmetic Decoding methods, the employed security parameter is $\lambda = 256$; additionally, for the Chunked Arithmetic Decoding approach, the utilized chunk size is 8.

By observing the results obtained across all three NIST security levels, the Rejection Sampling method is the fastest method in all the tests. This performance is due to the fact that this method is a non-constant-time approach, and the number generation and rejection processes are not computationally expensive operations.

Among the Arithmetic Decoding methods, the Chunked and the Chunked Tree approaches are the fastest ones. We can see that they exhibit similar performance; the only difference, as discussed in the previous chapter, is that the Chunked Tree approach is slightly faster in cases where n^4 fits entirely in a 64-bit machine word (i.e., $n^4 \leq 2^{64}$). The slowest approach is consistently the Monolithic one; this is due to the fact that while it achieves

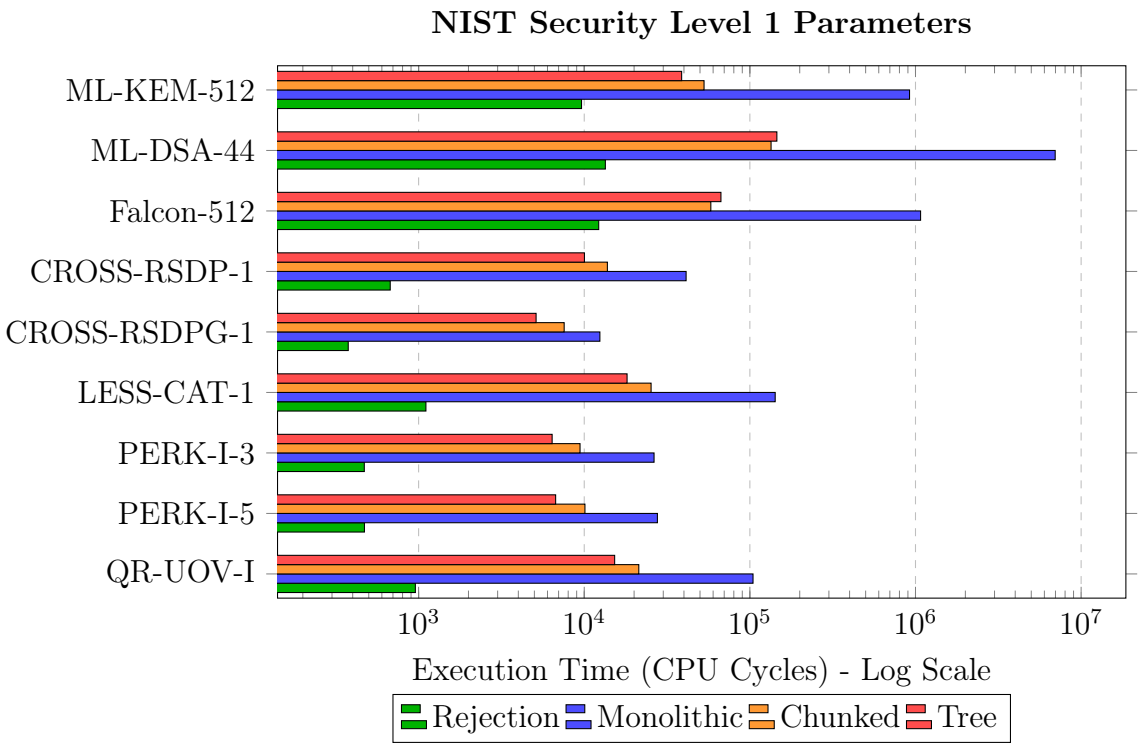

Figure 3.3: Performance comparison of modular array generation for NIST Security Level 1 parameters.

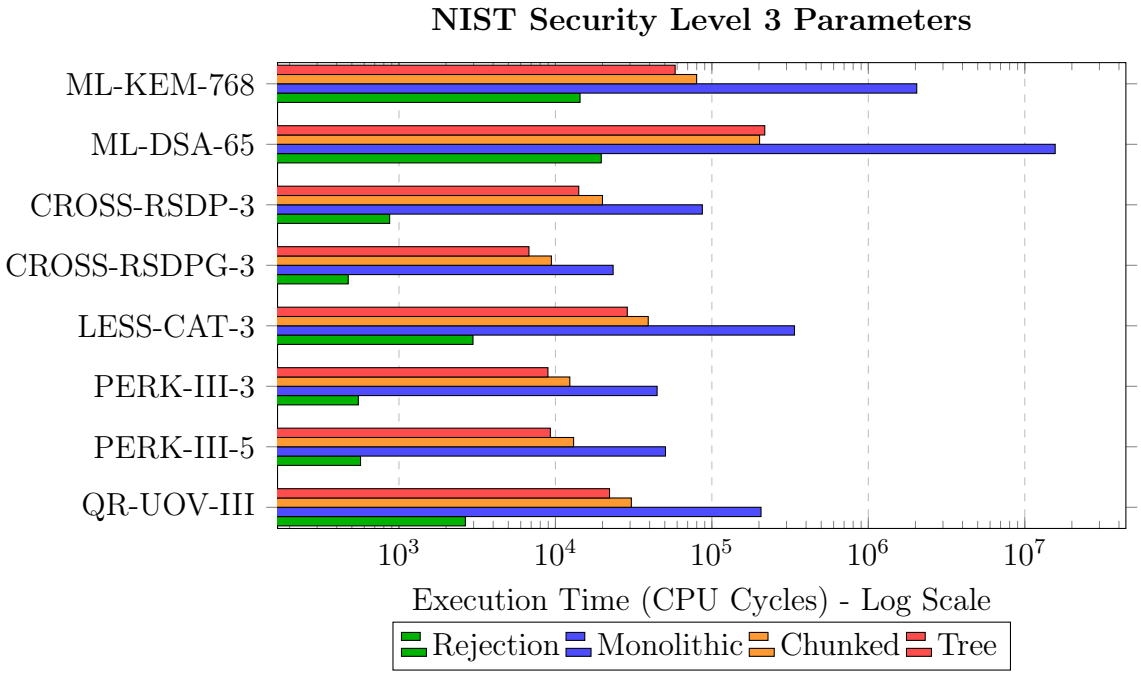

Figure 3.4: Performance comparison of modular array generation for NIST Security Level 3 parameters.

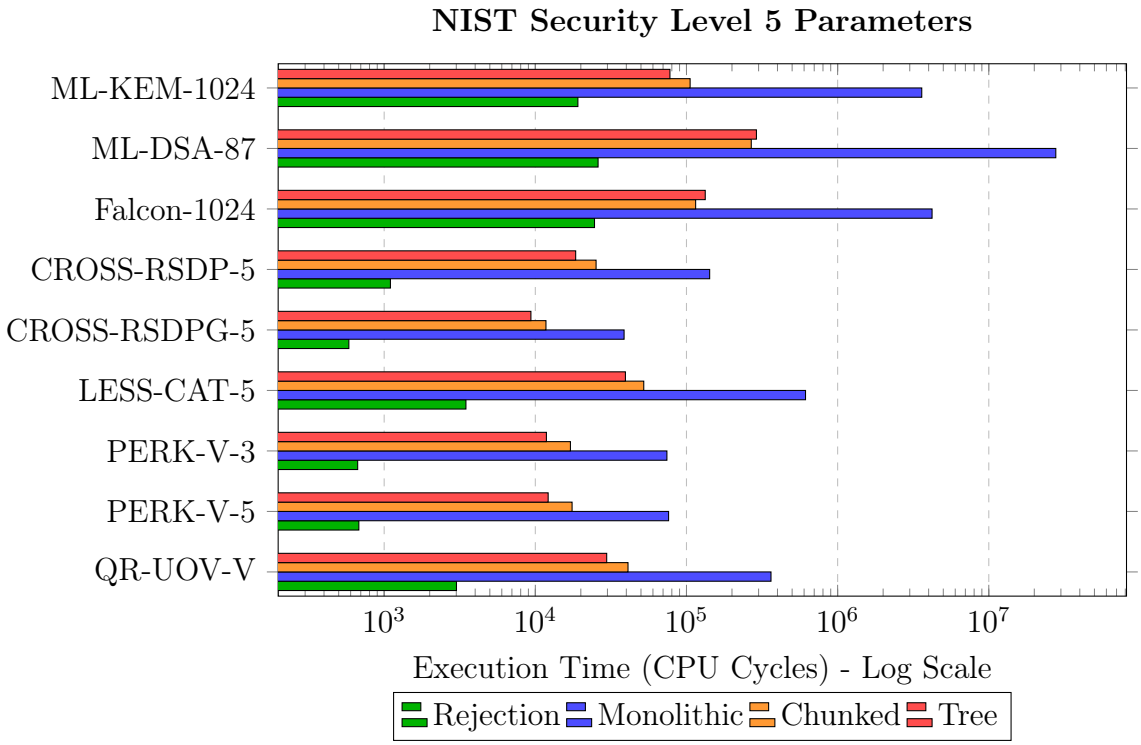

Figure 3.5: Performance comparison of modular array generation for NIST Security Level 5 parameters.

optimal entropy usage, it requires much more expensive and slower multi-precision division operations on a single, large dividend.

3.2.3. Fixed weight vector generation results

Figures 3.6 to 3.8 show the performance tests on the five illustrated methods for fixed weight vectors generation. For the Rejection Sampling method, the Rejection approach for the generation of modular array numbers is employed.

By analyzing the results obtained across all three NIST security levels, a clear performance pattern emerges based on the density of the target vectors. Among the non-constant-time approaches, the RepeatedAND and the Rejection methods consistently outperform the others, whereas the classical Fisher-Yates method is strictly dominated in all evaluated scenarios. Specifically, the RepeatedAND method provides the highest throughput for the generation of dense vectors (where the ratio $w/l \geq 0.2$), while the Rejection approach proves to be the most efficient for sparse vectors.

A symmetric behavior is observed within the constant-time approaches: the Sorting method is highly optimized and dominates the generation of dense vectors, whereas

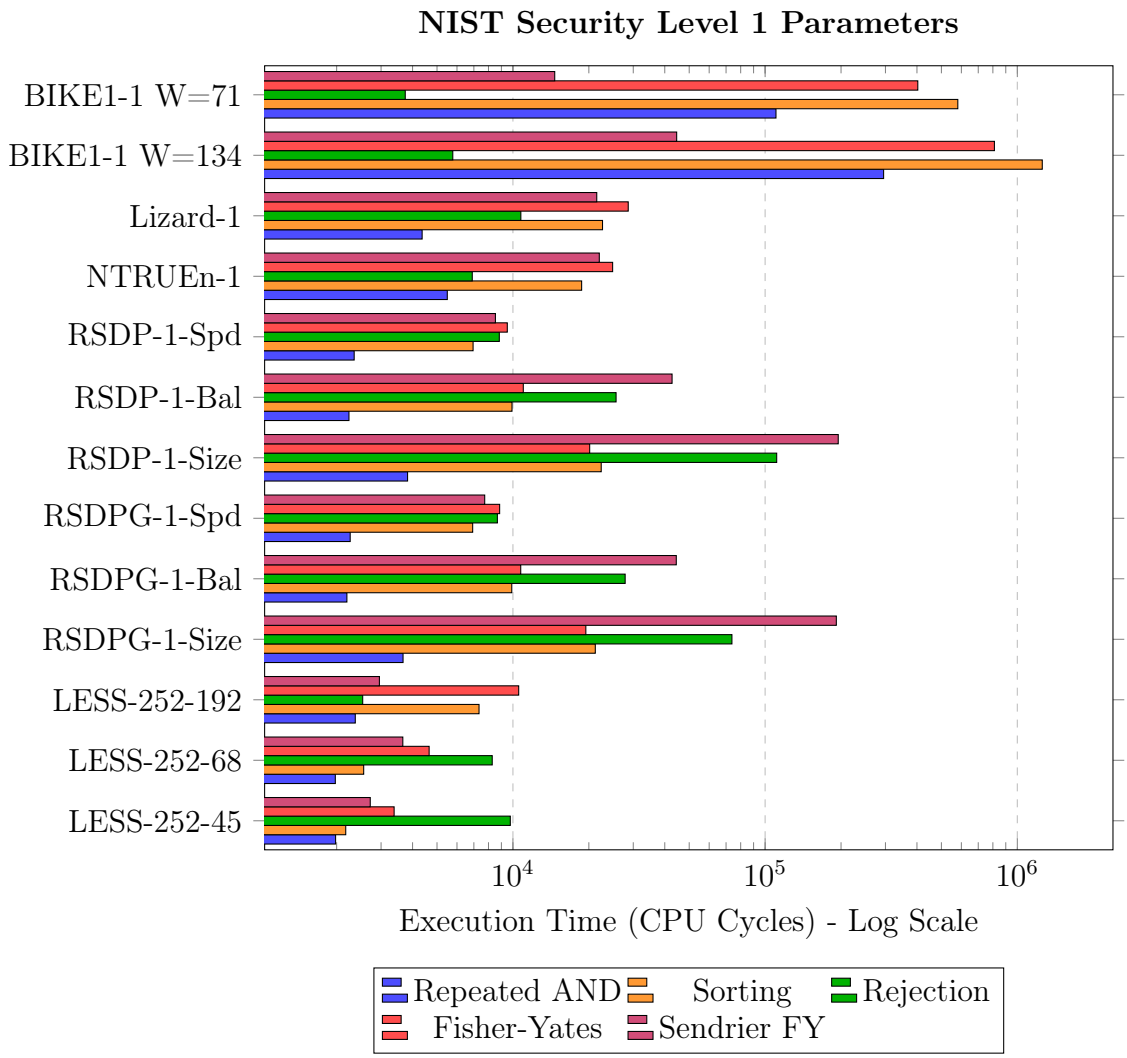

Figure 3.6: Performance comparison of fixed-weight vector generation for NIST Security Level 1 parameters.

Sendrier's Fisher-Yates variant scales much better and represents the fastest constant-time solution for sparse vectors.

3.2.4. Permutation generation results

Figure 3.9 shows the performance results of the control bits generation and routing for Beneš networks for different power of 2 inputs.

For control bits generation in Beneš networks, performance clearly depends on the size of the network. Increasing the size to the next power of 2 results in a doubling or tripling of the computational overhead. The same reasoning applies to the routing process, which is exponentially faster compared to control bits generation.

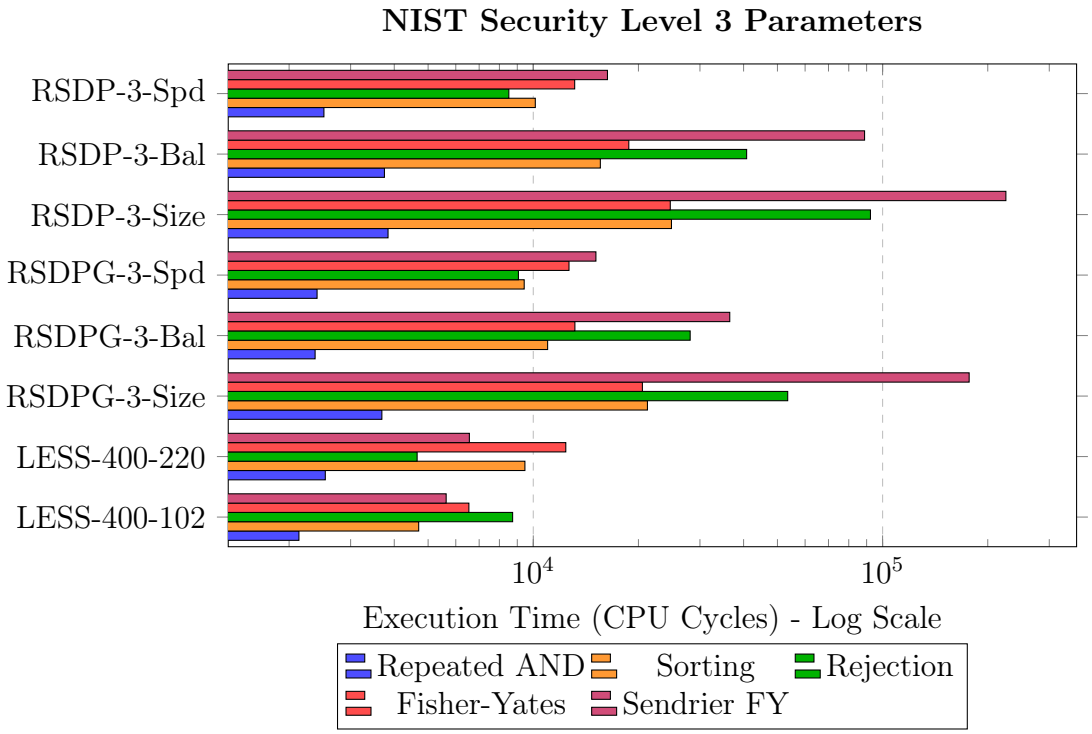

Figure 3.7: Performance comparison of fixed-weight vector generation for NIST Security Level 3 parameters.

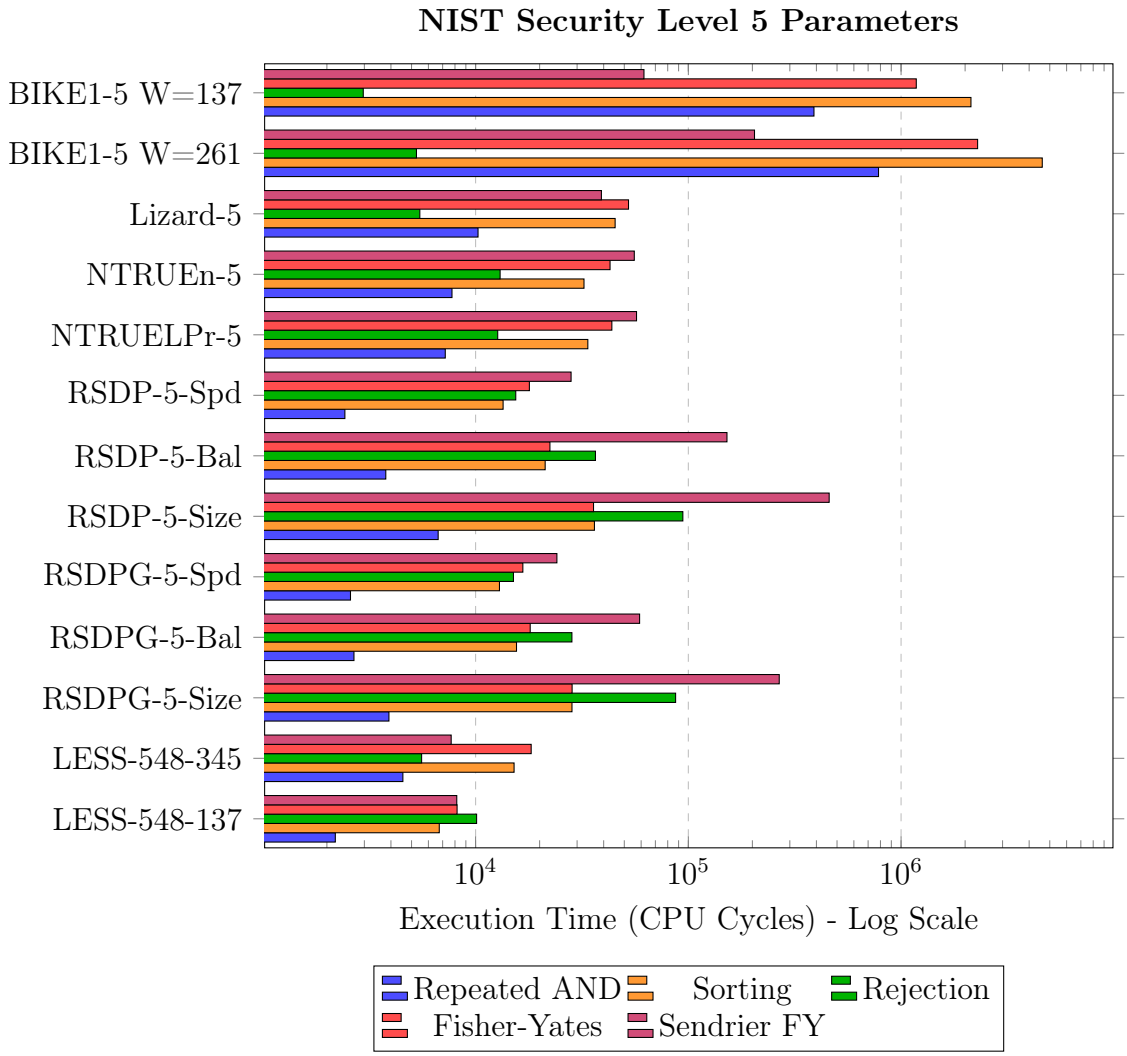

Figure 3.8: Performance comparison of fixed-weight vector generation for NIST Security Level 5 parameters.

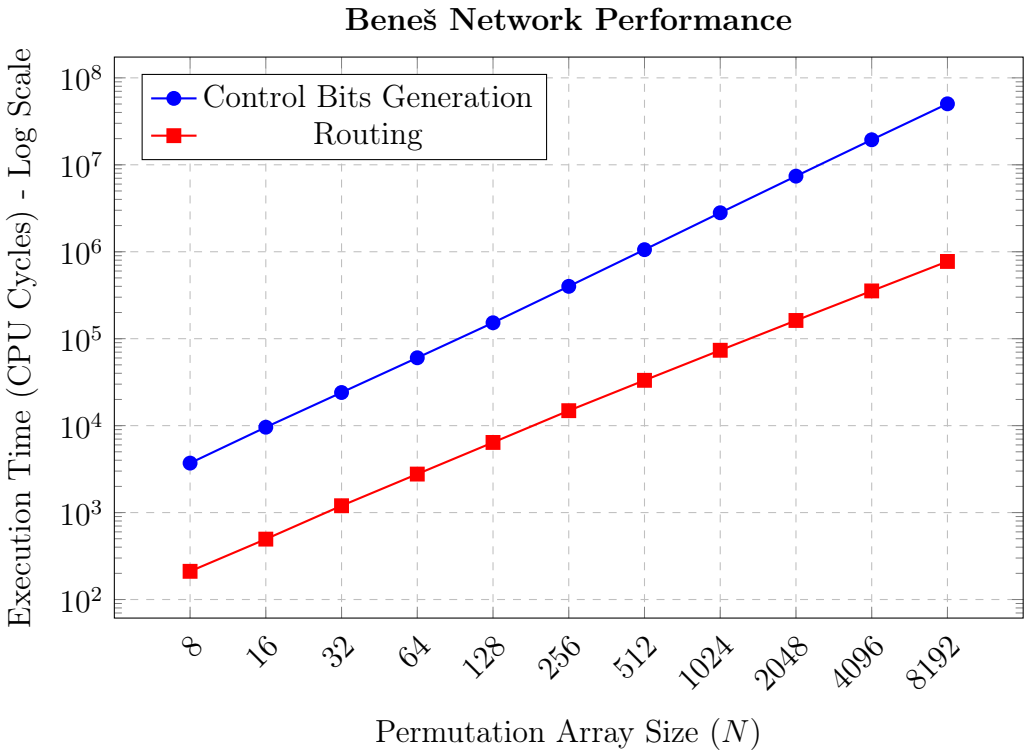

Figure 3.9: Performance comparison of control bits generation and routing for Beneš networks of varying sizes.

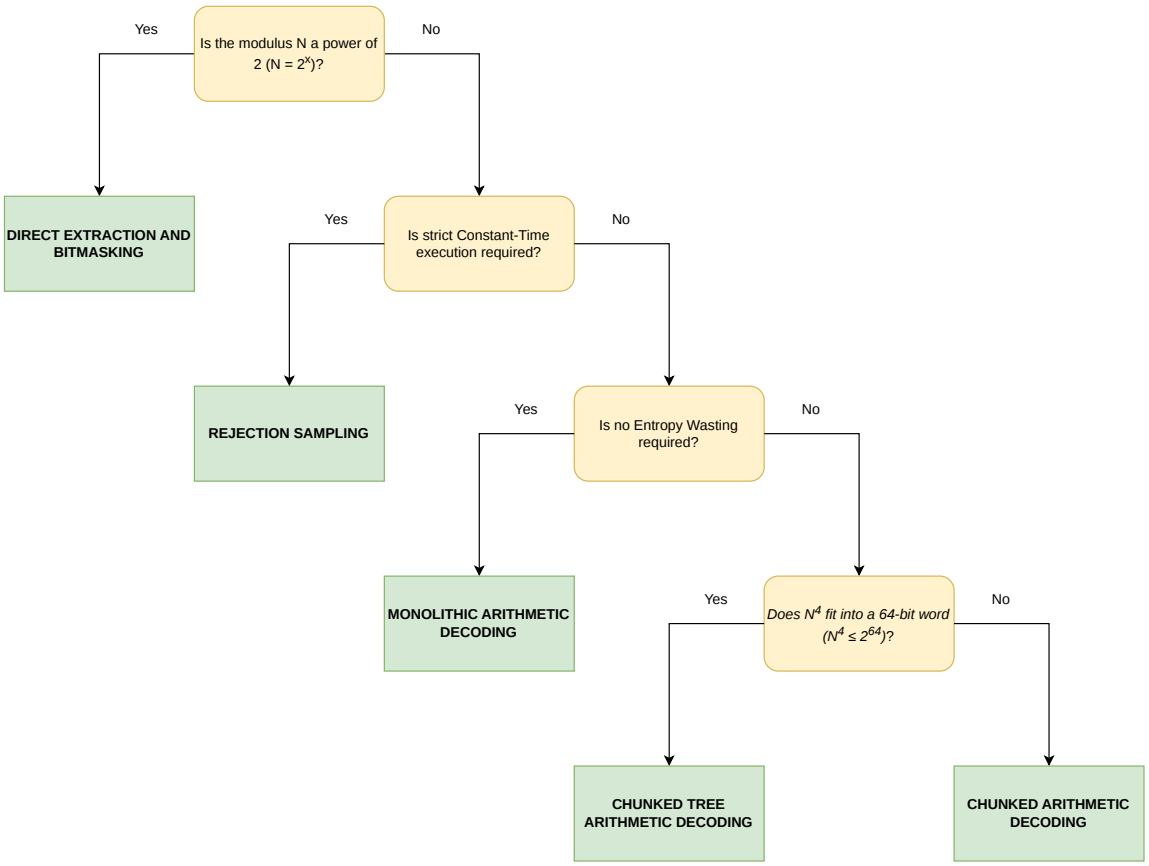

Figure 3.10: Modular Array Generation decision tree

3.3. Experimental Analysis

Observing the results obtained in Section 3.2, we can notice that the performance of the algorithms mostly depends on the input parameters.

Figure 3.10 shows a decision tree for selecting the method to be used, depending on the input parameters and the properties to be achieved. If the modulus is a power of 2, the binary string generator primitive can be directly employed; otherwise, one of the methods introduced in Section 2.3 must be used.

As discussed in Section 3.2.2, the Rejection Sampling method is the fastest approach, at the cost of not respecting the constant-time property. Among the arithmetic decoding methods, if optimal entropy usage is strictly required, the Monolithic approach is the one to be chosen. Otherwise, the Chunked and the Chunked Tree methods achieve significantly faster performance at the cost of higher entropy waste.

For fixed-weight vector generation (Section 3.2.3), Figure 3.11 illustrates a decision tree

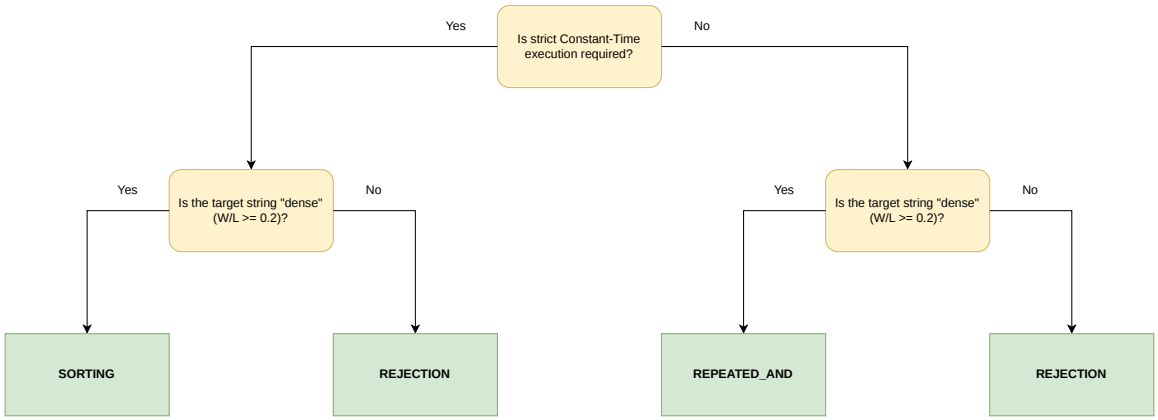

Figure 3.11: Fixed Weight Vector decision tree

designed to help select the optimal algorithmic approach. Note that the classical Fisher-Yates method is excluded from this tree, as it is consistently outperformed by alternative methods across all benchmarks.

The selection process is driven by two main factors: the constant-time requirement and the vector density. If the constant-time property is not required, the RepeatedAND method is the optimal strategy for generating dense vectors, while the Rejection approach is the most efficient choice for sparse vectors.

When strict constant-time execution is enforced, the Sorting method is the superior choice for dense vectors, whereas Sendrier's Fisher-Yates variant becomes the mandatory solution to efficiently handle sparse vectors.

4 | Conclusion

This thesis presented the RandShaper framework, a library designed to support post-quantum cryptographic schemes in the generation and shaping of pseudo-random objects: binary strings, fixed-weight vectors, modular numbers, and permutations. In particular, where possible, the algorithms were implemented in a constant-time manner, an approach required to prevent side-channel timing attacks. Additionally, the methods were implemented by leveraging the SIMD capabilities of modern CPUs, such as AVX2 instructions, to process multiple data elements simultaneously.

The development of this library comes at a critical time: as the NIST standardization program advances, the RandShaper framework comes into play in providing approaches and algorithms able to support post-quantum developers in a secure, side-channel resistant way to generate pseudo-random objects.

Observing the obtained results, we can conclude that there is no universally dominant method for the generation of a pseudo-random object; some methods excel for certain sets of parameters, while others are more computationally efficient for different ones. Furthermore, the choice of the algorithm to be employed in a post-quantum cryptographic scheme does not depend solely on the efficiency of the method, but also on the specific objectives set by the scheme, such as the requirement of minimal entropy waste or the necessity of the constant-time property. For this reason, the results obtained and discussed in Section 3.2 and the decision trees presented in Section 3.3 aim to support developers in deciding which method to employ in the generation of these pseudo-random objects.

The RandShaper library can be easily expanded in the future by integrating new Pseudo-Random Number Generators (PRNGs), as well as new shaping algorithms dedicated to the generation of the above-mentioned pseudo-random objects.

Bibliography

[1] Najwa Aaraj, Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Victor Dyseryn, Andre Esser, Thibauld Feneuil, Philippe Gaborit, Mukul Kulkarni, Victor Mateu, Marco Palumbi, Lucas Perin, Matthieu Rivain, Jean-Pierre Tillich, and Keita Xagawa. PERK signature scheme. `https://pqc-perk.org/`, 2023.

[2] Marius A. Aardal, Gora Adj, Diego F. Aranha, Andrea Basso, Isaac Andrés Canales Martínez, Jorge Chávez-Saab, Maria Corte-Real Santos, Pierrick Dartois, Luca De Feo, Max Duparc, Jonathan Komada Eriksen, Tako Boris Fouotsa, Décio Luiz Gazzoni Filho, Basil Hess, David Kohel, Antonin Leroux, Patrick Longa, Luciano Maino, Michael Meyer, Kohei Nakagawa, Hiroshi Onuki, Lorenz Panny, Sikhar Patranabis, Christophe Petit, Giacomo Pope, Krijn Reijnders, Damien Robert, Francisco Rodríguez-Henríquez, Sina Schaeffler, and Benjamin Wesolowski. SQIsign. Technical report, 2025.

[3] Gora Adj, Nicolas Aragon, Stefano Barbero, Magali Bardet, Emanuele Bellini, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Victor Dyseryn, Andre Esser, Thibauld Feneuil, Philippe Gaborit, Romaric Neveu, Matthieu Rivain, Luis Rivera-Zamarripa, Carlo Sanna, Jean-Pierre Tillich, Javier Verbel, and Floyd Zweydinger. Mirath signature scheme. `https://pqc-mirath.org/`, 2023.

[4] Carlos Aguilar-Melchor, Slim Bettaieb, Loïc Bidoux, Thibauld Feneuil, Philippe Gaborit, Nicolas Gama, Shay Gueron, James Howe, Andreas Hülsing, David Joseph, Antoine Joux, Mukul Kulkarni, Edoardo Persichetti, Tovohery H. Randrianarisoa, Matthieu Rivain, and Dongze Yue. SDitH: Additional Digital Signature Scheme for NIST PQC Standardization. `https://sdith.org/`, 2023.

[5] Rika Akiyama, Hiroki Furue, Yasuhiko Ikematsu, Fumitaka Hoshino, Koha Kinjo, Haruhisa Kosuge, Satoshi Nakamura, Shingo Orihara, Tsuyoshi Takagi, and Kimihiro Yamakoshi. QR-UOV: Additional Digital Signature Scheme for NIST PQC Standardization. `https://info.isl.ntt.co.jp/crypt/qruov/`, 2023.

[6] Nicolas Aragon, Magali Bardet, Loïc Bidoux, Jesús-Javier Chi-Domínguez, Vic-

tor Dyseryn, Thibauld Feneuil, Philippe Gaborit, Antoine Joux, Romaric Neveu, Matthieu Rivain, Jean-Pierre Tillich, and Adrien Vincotte. RYDE signature scheme. `https://pqc-ryde.org/`, 2023.

[7] Nicolas Aragon and et al. Bike - bit flipping key encapsulation. `https://bikesuite.org/`, 2017.

[8] M. Baldi, A. Barenghi, S. Bitzer, P. Karl, F. Manganiello, A. Pavoni, G. Pelosi, P. Santini, J. Schupp, F. Slaughter, A. Wachter-Zeh, and V. Weger. Cross - codes and restricted objects signature scheme. `https://www.cross-crypto.com/`, 2023.

[9] Marco Baldi, Alessandro Barenghi, Luke Beckwith, Tung Chou, Jean-François Biasse, Andre Esser, Kris Gaj, Patrick Karl, Kamyar Mohajerani, Gerardo Pelosi, Edoardo Persichetti, Markku-Juhani Saarinen, Paolo Santini, Robert Wallace, and Floyd Zweydinger. LESS: Linear equivalence signature scheme. `https://www.less-project.com/`, 2023.

[10] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators. Technical Report NIST Special Publication (SP) 800-90A Rev. 1, National Institute of Standards and Technology, June 2015.

[11] Carsten Baum, Ward Beullens, Lennart Braun, Cyprien Delpech de Saint Guilhem, Michael Klooß, Christian Majenz, Shibam Mukherjee, Emmanuela Orsini, Sebastian Ramacher, Christian Rechberger, Lawrence Roy, and Peter Scholl. FAEST signature scheme. `https://faest.info/`, 2023.

[12] Ryad Benadjila, Charles Bouillaguet, Thibauld Feneuil, and Matthieu Rivain. MQOM: Mq on my mind. `https://mqom.org/`, 2023.

[13] V.E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Bell Telephone Laboratories, Incorporated Murray Hill, New Jersey, 1965.

[14] Daniel J. Bernstein. libcpucycles: A microlibrary for counting cpu cycles. `https://cpucycles.cr.yp.to/`.

[15] Daniel J. Bernstein. djbsort: A library for software integer sorting. `https://sorting.cr.yp.to/index.html`, 2018.

[16] Daniel J. Bernstein. Verified fast formulas for control bits for permutation networks. Document ID: 782136154c6ea5bb93257e2a5f667a579df7a48e, 2020.

[17] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. Ntru prime: Reducing attack surface at low cost. In *Selected Areas*

in Cryptography - SAC 2017: 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers, pages 235–260. Springer, 2017.

[18] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess, and Matthias J. Kannwischer. MAYO signature scheme. `https://pqmayo.org/`, 2023.

[19] Ward Beullens, Ming-Shing Chen, Jintai Ding, Boru Gong, Matthias J. Kannwischer, Jacques Patarin, Bo-Yuan Peng, Dieter Schmidt, Cheng-Jhih Shih, Chengdong Tao, and Bo-Yin Yang. UOV: Unbalanced oil and vinegar. `https://www.uovsig.org/`, 2023.

[20] Joppe W. Bos, Olivier Bronchain, Léo Ducas, Serge Fehr, Yu-Hsuan Huang, Thomas Pornin, Eamonn W. Postlethwaite, Thomas Prest, Ludo N. Pulles, and Wessel van Woerden. Hawk: Fast and secure signatures from lattices. `https://hawk-sign.info/`, 2023.

[21] Nir Drucker and Shay Gueron. Generating a random string with a fixed weight. In *Cyber Security Cryptography and Machine Learning: Third International Symposium, CSCML 2019, Beer-Sheva, Israel, June 27–28, 2019, Proceedings 3*, pages 141–155. Springer, 2019.

[22] Richard Durstenfeld. Algorithm 235: Random permutation. In *Communications of the ACM, Volume 7, Issue 7*, page 420, 1964.

[23] R.A. Fisher, F. Yates, and et al. Statistical tables for biological, agricultural and medical research, 1949.

[24] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-fourier lattice-based compact signatures over ntru. `https://falcon-sign.info/`, 2017.

[25] Philippe Gaborit, Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jérôme Lacan, Jean-Marc Robert, Pascal Véron, Paulo L. Barreto, Santosh Ghosh, Shay Gueron, Tim Güneysu, Rafael Misoczki, Jan Richter-Brokmann, Nicolas Sendrier, Jean-Pierre Tillich, and Valentin Vasseur. Hqc: Hamming quasi-cyclic. `https://pqc-hqc.org/`, 2017.

[26] Marco Gianvecchio, Alessandro Barenghi, and Gerardo Pelosi. Towards efficient post-quantum signatures: Parallelizing keccak in cross and contributing to opensource libraries. Master's thesis, Politecnico di Milano, October 2024.

[27] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. *ACM SIGPLAN Notices*, 29(6):61–72, 1994.

[28] Jeff Hoffstein, Nick Howgrave-Graham, Jill Pipher, and William Whyte. Practical lattice-based cryptography: Ntruencrypt and ntrusign. In *The LLL Algorithm - Survey and Applications*, pages 349–390. Springer, 2009.

[29] Daniel Lemire. Fast random integer generation in an interval. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 29(1):1–12, 2019.

[30] David Nassimi and Sartaj Sahni. Parallel algorithms to set up the benĕs permutation network. *IEEE Transactions on Computers*, 31(2):148–154, 1982.

[31] National Institute of Standards and Technology. FIPS 197: Advanced Encryption Standard (AES) . Technical report, U.S. Department of Commerce, November 2001.

[32] National Institute of Standards and Technology. FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical report, U.S. Department of Commerce, August 2015.

[33] National Institute of Standards and Technology. FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard. Technical report, U.S. Department of Commerce, August 2024.

[34] National Institute of Standards and Technology. FIPS 204: Module-Lattice-Based Digital Signature Standard. Technical report, U.S. Department of Commerce, August 2024.

[35] National Institute of Standards and Technology. FIPS 205: Stateless Hash-Based Digital Signature Standard. Technical report, U.S. Department of Commerce, August 2024.

[36] National Institute of Standards and Technology. Post-Quantum Cryptography: Round 2 Additional Digital Signatures. `https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures`, 2024.

[37] National Institute of Standards and Technology (NIST). Post-quantum cryptography standardization: Security (evaluation criteria). `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria)`, 2016.

[38] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. Technical report, June 2018.

[39] National Institute of Standards and Technology (NIST). Announcing request for nominations for public-key post-quantum cryptographic algorithms. Federal Register, Vol. 81, No. 244, 2016.

[40] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, James Vornbusch, Andrew Banks, Lawrence Heckert, Stefan Dray, and San Terrell. A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST Special Publication 800-22 Revision 1a, National Institute of Standards and Technology (NIST), 2010.

[41] Nicolas Sendrier. Secure sampling of constant-weight words - application to BIKE. HAL archive hal-03534005, 2022.

[42] Abraham Waksman. A permutation network. *Journal of the ACM (JACM)*, 15(1):159–163, 1968.

[43] Lih-Chung Wang, Chun-Yen Chou, Jintai Ding, Yen-Liang Kuan, Jan Adriaan Leegwater, Ming-Siou Li, Bo-Shu Tseng, Po-En Tseng, and Chia-Chun Wang. SNOVA: Simple noncommutative-ring based UOV with key-randomness alignment. `https://snova.pqclab.org/`, 2023.

[44] Henry S. Warren. *Hacker's delight*. Pearson Education, 2nd edition, 2012.

[45] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

A | Appendix: Additional Tables

Table A.1: Performance comparison (CPU cycles) for binary string generation.

Bits generated	Execution Time (CPU Cycles)				
	AES256-Software	**AES256-NI-CTR**	**SHAKE256**	**SHAKE256-X4**	**ChaCha-AVX2**
16	2234	163	2051	1436	1622
32	2239	164	2048	1431	1615
64	2239	165	2049	1421	1616
128	2239	165	2046	1417	1615
256	4389	244	2046	1417	1616
512	8713	401	2047	1417	1616
1024	17311	750	2052	1437	1620
2048	34502	1382	4044	2738	1623
4096	68870	2646	7997	5330	1633
8192	137608	5174	15875	10450	3180
16384	275136	10230	31607	20633	6312
32768	550093	20360	61113	39668	12504
65536	1100281	40631	120109	78160	24890
131072	2200387	81205	238199	154591	49684
262144	4401437	162224	474243	307903	99404
524288	8808937	324321	948478	617643	198569
1048576	17613410	648466	1898700	1236025	396920

Table A.2: Comparison of entropy consumption (bits used) for modular array generation.

Algorithm	Modulus	N	Consumed Entropy (Bits)			
			Rejection	Monolithic	Chunked	Tree
ML-KEM-512	3329	512	8192	6400	22528	40960
ML-KEM-768	3329	768	12288	9472	33792	61440
ML-KEM-1024	3329	1024	16384	12544	45056	81920
ML-DSA-44	8380417	1024	24576	23808	56320	163840
ML-DSA-65	8380417	1536	36864	35584	84480	245760
ML-DSA-87	8380417	2048	48883	47360	112640	327680
Falcon-512	12289	512	10241	7424	23552	73728
Falcon-1024	12289	1024	20458	14592	47104	147456
CROSS-RSDP-1	127	127	2048	1145	4985	9216
CROSS-RSDP-3	127	187	2048	1565	7453	13536
CROSS-RSDP-5	127	251	2048	2013	9949	18144
CROSS-RSDPG-1	509	55	2048	751	2287	4480
CROSS-RSDPG-3	509	79	2048	967	3271	6400
CROSS-RSDPG-5	509	106	2048	1210	4538	8640
LESS-CAT-1	127	252	2048	2020	9956	18144
LESS-CAT-3	127	400	4096	3056	15600	28800
LESS-CAT-5	127	548	4096	4092	21500	39456
PERK-I-3	1021	79	2048	1046	3350	6400
PERK-I-5	1021	83	2048	1086	3646	6720
PERK-III-3	1021	112	2048	1376	4704	8960
PERK-III-5	1021	116	2048	1416	5000	9280
PERK-V-3	1021	146	2048	1716	6324	11840
PERK-V-5	1021	150	2048	1756	6364	12160
QR-UOV-I	127	210	2048	1726	8382	15264
QR-UOV-III	127	306	4096	2398	12126	22176
QR-UOV-V	127	411	4096	3133	16189	29664

Table A.3: Performance comparison (CPU cycles) for modular array generation.

Algorithm	Modulus	N	Execution Time (CPU Cycles)			
			Rejection	Monolithic	Chunked	Tree
ML-KEM-512	3329	512	9631	920602	52894	38743
ML-KEM-768	3329	768	14371	2039625	79928	58247
ML-KEM-1024	3329	1024	19127	3602835	105703	77732
ML-DSA-44	8380417	1024	13413	6972986	134284	145564
ML-DSA-65	8380417	1536	19637	15645608	201818	217855
ML-DSA-87	8380417	2048	26014	27758617	268335	290272
Falcon-512	12289	512	12229	1073380	58123	66893
Falcon-1024	12289	1024	24635	4214564	114990	133097
CROSS-RSDP-1	127	127	674	41213	13795	10027
CROSS-RSDP-3	127	187	872	86930	19994	14099
CROSS-RSDP-5	127	251	1100	142374	25232	18492
CROSS-RSDPG-1	509	55	376	12429	7567	5120
CROSS-RSDPG-3	509	79	474	23379	9422	6776
CROSS-RSDPG-5	509	106	584	38664	11759	9350
LESS-CAT-1	127	252	1107	142198	25329	18144
LESS-CAT-3	127	400	2973	337369	39202	28800
LESS-CAT-5	127	548	3475	613255	52169	39456
PERK-I-3	1021	79	470	26395	9433	6400
PERK-I-5	1021	83	471	27658	10099	6720
PERK-III-3	1021	112	550	44668	12363	8960
PERK-III-5	1021	116	568	50526	13068	9280
PERK-V-3	1021	146	668	74233	17071	11840
PERK-V-5	1021	150	680	76165	17507	12160
QR-UOV-I	127	210	956	104486	21353	15264
QR-UOV-III	127	306	2658	206160	30570	22176
QR-UOV-V	127	411	3009	362447	41039	29664

Table A.4: Comparison of entropy consumption (bits used) for fixed weight generation.

Algorithm	L	W	Consumed Entropy (Bits)				
			Rep.AND	**Sort.**	**Rej.**	**FY**	**Sendrier FY**
BIKE1-1	10163	71	344434	650496	2424	186679	2304
BIKE1-1	20326	134	856017	1300992	4872	402208	4352
BIKE1-5	32749	137	1173861	2096128	4624	636760	4480
BIKE1-5	65498	261	2543261	4192000	7104	1362928	8448
Lizard-1	536	140	9538	34304	10295	7128	4480
Lizard-5	1024	200	24655	65536	7304	14288	6400
NTRUEn-1	443	143	8311	28416	9606	6144	4608
NTRUEn-5	743	247	15810	47616	12769	10240	7936
NTRUELPr-5	761	250	16897	48896	12583	10287	8064
RSDP-1-Spd	157	82	1873	10240	12727	2048	2688
RSDP-1-Bal	256	215	3072	16384	47385	4096	6912
RSDP-1-Size	520	488	5915	33280	138859	6159	15616
RSDP-3-Spd	239	125	3100	15360	13944	4096	4096
RSDP-3-Bal	384	321	5087	24576	52023	6080	10368
RSDP-3-Size	580	527	7649	37120	101554	8192	16896
RSDP-5-Spd	321	167	4639	20736	16736	4096	5376
RSDP-5-Bal	512	427	7144	32768	54846	6144	13696
RSDPG-5-Size	832	762	12141	53248	121599	12255	24448
RSDPG-1-Spd	147	76	1626	9472	12456	2048	2432
RSDPG-1-Bal	256	220	2916	16384	52666	4096	7040
RSDPG-1-Size	512	484	5808	32768	145477	6144	15488
RSDPG-3-Spd	224	119	2863	14336	14070	3992	3840
RSDPG-3-Bal	268	196	3458	17152	30544	4096	6272
RSDPG-3-Size	512	463	6656	32768	95197	6144	14848
RSDPG-5-Spd	300	153	4012	19200	16159	4096	4992
RSDPG-5-Bal	356	258	5296	22784	31467	6144	8320
RSDPG-5-Size	642	575	8876	41216	95666	8543	18432
LESS-252-192	192	36	2759	12288	5056	2104	1152
LESS-252-68	68	42	594	4352	13878	2048	1408
LESS-252-45	45	34	344	3072	19197	2048	1152
LESS-400-220	220	68	3107	14080	7559	3879	2176
LESS-400-102	102	61	981	6656	14464	2048	2048
LESS-548-345	345	75	6560	22272	6496	4207	2432
LESS-548-137	137	79	1530	8960	14272	2048	2560

Table A.5: Performance comparison (CPU cycles) for fixed weight generation.

Algorithm	L	W	Execution Time (CPU Cycles)				
			Rep.AND	**Sort.**	**Rej.**	**FY**	**Sendrier FY**
BIKE1-1	10163	71	110429	580880	3742	403112	14648
BIKE1-1	20326	134	295038	1255960	5775	811395	44598
BIKE1-5	32749	137	389579	2130669	2965	1178972	61871
BIKE1-5	65498	261	783676	4610492	5273	2290008	204672
Lizard-1	536	140	4365	22661	10750	28657	21498
Lizard-5	1024	200	10268	45281	5469	52341	39035
NTRUEn-1	443	143	5492	18729	6899	24852	22003
NTRUEn-5	743	247	7747	32330	13039	42861	55720
NTRUELPr-5	761	250	7203	33699	12706	43705	57090
RSDP-1-Spd	157	82	2347	6951	8831	9499	8523
RSDP-1-Bal	256	215	2238	9916	25644	10993	42761
RSDP-1-Size	520	488	3822	22392	111148	20153	194985
RSDP-3-Spd	239	125	2516	10135	8513	13139	16309
RSDP-3-Bal	384	321	3750	15561	40812	18774	88797
RSDP-3-Size	580	527	3840	24867	92315	24673	225392
RSDP-5-Spd	321	167	2430	13469	15447	17901	28111
RSDP-5-Bal	512	427	3784	21235	36598	22359	151941
RSDPG-5-Size	832	762	6669	36196	94240	35841	459226
RSDPG-1-Spd	147	76	2263	6925	8677	8858	7730
RSDPG-1-Bal	256	220	2196	9895	27863	10739	44437
RSDPG-1-Size	512	484	3667	21215	73855	19463	191610
RSDPG-3-Spd	224	119	2405	9420	9068	12656	15113
RSDPG-3-Bal	268	196	2374	10993	28128	13158	36516
RSDPG-3-Size	512	463	3687	21225	53503	20530	176932
RSDPG-5-Spd	300	153	2582	12949	15067	16680	24110
RSDPG-5-Bal	356	258	2681	15573	28364	18060	59018
RSDPG-5-Size	642	575	3915	28388	87143	28425	267495
LESS-252-192	192	36	2371	7339	2537	10541	2955
LESS-252-68	68	42	1976	2560	8278	4653	3660
LESS-252-45	45	34	1981	2173	9763	3380	2718
LESS-400-220	220	68	2539	9464	4651	12391	6563
LESS-400-102	102	61	2134	4699	8728	6542	5632
LESS-548-345	345	75	4552	15168	5576	18253	7678
LESS-548-137	137	79	2191	6750	10110	8185	8157

Table A.6: Performance comparison (CPU cycles) for control bits generation and routing.

Permutation array size	Execution Time (CPU Cycles)	
	Control Bits Generation	Routing
8	3706	211
16	9591	495
32	24035	1198
64	60306	2772
128	152474	6405
256	399978	14855
512	1056412	33200
1024	2807063	73584
2048	7403606	161982
4096	19398960	354248
8192	50266986	773302

List of Figures

List of Tables