IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 0, No. 0, pp. 0–0. DOI:10.46586/tosc.v0.i0.0-0

LEAP: High-Performance Lattice-Based Pseudorandom Number Generator

Yu Zhang[1,2], Xianhui Lu[1,2] (✉), Yijian Liu[1,2], Yongjian Yin[1,2] and Kunpeng Wang[1,2]

[1] State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China,
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
{zhangyu1999,luxianhui,liuyijian,yinyongjian,wangkunpeng}@iie.ac.cn

Abstract. At EUROCRYPT2012, Banerjee, Peikert, and Rosen introduced Ring Learning With Rounding (RLWR) problem and constructed lattice-based pseudorandom functions for the first time. Subsequently, Banerjee, Brenner, Leurent, Peikert, and Rosen named this family of lattice-based pseudorandom functions as SPRING, reanalyzed the security, and gave two practical instances. Building upon the SPRING family, Bouillaguet, Delaplace, Fouque, and Kirchner further extended it to a pseudorandom number generator called SPRING-RS. It is quite fast but still has a certain gap compared with the classical pseudorandom number generator based on symmetric cryptography, and the key size is large.

In this work, we present LEAP, a lattice-based pseudorandom number generation scheme characterized by high performance, adaptable parameter selection, and extensive support for parallel processing. Unlike the RLWR problem used in public key cryptography, LEAP treats the public parameter in the RLWR problem as the key as well. Hiding the public parameters leads to larger lattice dimensions and higher standard deviations of error in the concrete security analysis compared to RLWR under identical parameters. These adjustments imply enhanced security, allowing smaller parameters while maintaining the same security level, thereby improving performance. Additionally, we introduce a novel framework that reuses multiple parameters, significantly enhancing overall performance. To mitigate the issue of increased key size caused by treating the public parameter as the key, we design a pseudorandom number generator leveraging the small key size characteristic of a variant of the NTRU assumption, which provides the key required for the high-performance pseudorandom number generator.

Compared with the SPRING-RS, the LEAP can reduce the key size by 1.71X while improving performance by 3.30X at the same security level. Under the `AVX2` and `AVX512` implementations, the performance reaches 1.61 Cycles/byte and 1.14 Cycles/byte, and the throughput reaches 16.12 Gbps and 22.60 Gbps, respectively.

Keywords: Lattice-based cryptography · PRNG · NTRU · RLWR

1 Introduction

Pseudorandom Number Generators (PRNGs), which can stretch a short random bit sequence into a longer pseudorandom bit sequence, play a vital role in cryptography. They are extensively employed in cryptographic schemes to generate cryptographically secure random numbers, such as generating keys from seed, providing initialization vectors for symmetric encryption, and supplying salts to be hashed with passwords. Most classical high-performance PRNGs are based on symmetric cryptography, such as Yarrow [KSF99] based

Licensed under Creative Commons License CC-BY 4.0.

on SHA-1, AES-CTR [HW03] based on AES, and Fortuna [FSK15] based on SHA256. Similarly, the PRNG design mechanisms recommended by the National Institute of Standards and Technology (NIST) SP 800-90A [BK15] also utilize symmetric cryptography, respectively employing hash functions, keyed hash message authentication codes (HMACs), and block ciphers as fundamental components.

Lattice-based cryptographic schemes provide provable security, which can defend against attacks from both classical and quantum computers. They demonstrate significant performance advantages over traditional cryptosystems like RSA and ECDH, positioning them as highly promising and mature candidates in post-quantum cryptography. Moreover, the algebraic structure of lattices enables efficient homomorphic operations, further extending the high-performance benefits of lattice-based schemes to a broader range of application scenarios. In recent years, substantial progress has been made in lattice-based public key cryptography [PW08, BCD+16, BDK+18], identity-based encryption [CHKP12, ABB10a, ABB10b], and homomorphic encryption schemes [BV14, BGV14, CKKS17, CGGI20]. Despite these advancements, the use of lattices in constructing symmetric cryptographic primitives, particularly pseudorandom number generators (PRNGs), remains relatively underexplored. The lattice-based pseudorandom number generator construction will introduce a variety of algorithms based on different principles, thereby advancing the design of pseudorandom number generation schemes.

Recent research predominantly focused on pseudorandom functions (PRFs) based on lattices. There exists a close relationship between PRNGs and PRFs. By evaluating a PRF on a series of different inputs, it is simple to construct a PRNG from a PRF. The Ring Learning With Rounding (RLWR) problem, introduced by Banerjee, Peikert, and Rosen in [BPR12] (BPR), eliminates randomness in the Ring Learning With Errors (RLWE) problem. As a result, RLWR provides a deterministic and efficient mechanism for generating errors while preserving robust security guarantees, making it a strong foundation for constructing PRFs and PRNGs. BPR proposed two lattice-based PRF constructions: one leveraging the pseudorandom synthesizer of Naor and Reingold [NR95], and the other based on the subset-product structure. However, due to the large parameter requirements in their security proofs, these designs remained theoretical and did not yield practical PRF schemes. Later, Banerjee, Brenner, Leurent, Peikert, and Rosen [BBL+15] revisited the subset-product-based construction from BPR, introducing SPRING, along with two concrete and efficient PRF instances named SPRING-BCH and SPRING-CRT. These instances relaxed the stringent parameters of BPR while achieving high performance and avoiding concrete attacks on the function family. SPRING-BCH uses a prime modulus for the Number Theoretic Transform (NTT), which introduces rounding bias and makes the generated distribution distinguishable from uniform. To address this, it employs BCH codes to eliminate bias. In contrast, SPRING-CRT utilizes the Chinese Remainder Theorem (CRT) with a composite modulus, ensuring efficient computation without introducing bias during rounding. Bouillaguet, Delaplace, Fouque, and Kirchner [BDFK17] further reassessed SPRING as a PRNG, proposing a variant named SPRING-RS. This variant builds upon SPRING-BCH by replacing BCH codes with rejection sampling to eliminate rounding bias, achieving171 superior performance. SPRING-RS is currently regarded as the most efficient practical lattice-based PRNG. In another study, Chuengsatiansup and Stehlé [CS20] explored PRFs based on LWR and module-LWR, following the Goldreich, Goldwasser, and Micali (GGM) construction [GGM84], further advancing the efficiency and practicality of lattice-based PRFs.

1.1 Our Results

In this work, we propose a high-performance pseudorandom number generator scheme called LEAP, which is short for "**L**attice-based **E**fficient **A**daptable **P**seudorandom number generator". The LEAP scheme significantly improves the performance of lattice-based

PRNGs, allows adaptable parameter selection, and supports high parallelization, making it suitable for current scenarios with varying requirements. Moreover, compared with classical PRNGs, the algebraic structure of LEAP offers better performance in homomorphic evaluations, demonstrating homomorphic-friendly property.

For the the LEAP scheme, we give parameter selections for 128-bit, 192-bit, and 256-bit security levels, and provide three versions of implementations in plain `C`, `AVX2`, and `AVX512`. On our test machine, the `AVX2` and `AVX512` implementations achieve performances of 1.61 Cycles/byte and 16.12 Gbps, and 1.14 Cycles/byte and 22.60 Gbps. Compared to the current best lattice-based pseudorandom number generator, SPRING-RS, our scheme reduces the key size by 1.71X while enhancing performance by 3.30X at the same security level.

1.2 Our Techniques

Before delving into the techniques, we provide a brief overview of the intuition behind this work. The Ring Learning With Rounding (RLWR) problem, widely used in lattice-based public key cryptographic schemes, requires a public parameter. However, in the context of PRNG design, disclosing this parameter is unnecessary. To leverage this, we hide the public parameter in RLWR, thereby achieving improvements in other aspects. In concrete security analysis, hiding the public parameter increases the lattice dimensions and the standard deviation of the error compared to RLWR with equivalent parameters, thereby achieving higher security. This allows for a reduction in parameter size while maintaining the same security level, leading to improved performance. Furthermore, leveraging our novel framework, which efficiently reuses multiple parameters, enhances the scheme's performance even further. We refer to this process as LEAP.PRNG. Nevertheless, adopting the public parameter as the key in LEAP.PRNG will undoubtedly increase the key size, resulting in adding communication overhead. To mitigate this, we leverage the shorter key size characteristics of a variant of the NTRU problem and design a new PRNG, LEAP.GEN, to generate the key required for LEAP.PRNG, markedly reducing the key size.

Hiding Parameter. Ring Learning With Rounding (RLWR) problem was introduced by Banerjee, Peikert, and Rosen [BPR12] to construct lattice-based PRFs. Let $\mathcal{R}$ denote the cyclotomic polynomial ring $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$. For any integer modulus q, define the quotient ring $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. The rounding function $\lfloor\cdot\rceil_p : \mathbb{Z}_q \to \mathbb{Z}_p$ is defined as $\lfloor x\rceil_p = \lfloor (p/q)\cdot\bar{x}\rceil \bmod p$, where $\bar{x} \in \mathbb{Z}$ and $\bar{x} \equiv x \bmod q$, with the condition $q \geq p \geq 2$. The definition of RLWR problem is as follows: given many independent pairs $(a_i, b_i) \in \mathcal{R}_q \times \mathcal{R}_p$, where each a_i is uniformly random and $b_i = \lfloor a_i \cdot s\rceil_p$ for a random secret s. The decisional RLWR assumption is that for a given distribution over s, the advantage of distinguishing the generated distribution (a_i, b_i) from the uniformly random distribution over $\mathcal{R}_q \times \mathcal{R}_p$ is negligible. The definition of decisional RLWR assumption implies the reuse of s. As previously noted, in designing a PRNG, the provision of numerous independent $b_i \in \mathcal{R}_p$ suffices without the necessity of giving pairs $(a_i, b_i) \in \mathcal{R}_q \times \mathcal{R}_p$. If we hide a_i implying its treatment as the key, the hardness is at least as hard as the decisional RLWR assumption. However, in concrete security evaluation, smaller parameters can be used to achieve higher computational efficiency while maintaining the same security level as the RLWR assumption. This is advantageous for constructing high-performance PRNGs.

Novel Framework. As mentioned in [BPR12], if each a_i is reused across multiple independent secrets, the decisional RLWR problem turns to distinguishing distributions of $(a_i, \lfloor a_i \cdot s_0\rceil_p, \lfloor a_i \cdot s_1\rceil_p, \cdots, \lfloor a_i \cdot s_{k-1}\rceil_p) \in \mathcal{R}_q \times \mathcal{R}_p^k$ and uniformly at random from $\mathcal{R}_q \times \mathcal{R}_p^k$, where $k = \mathrm{poly}(n)$. Inspired by this, while reusing s, we further improve the performance by reusing a_i. In the LEAP.PRNG scheme, the input includes a uniformly randomly generated

$\boldsymbol{a} \in \mathcal{R}_q^m$ and s generated according to the secret distribution. The pseudorandom numbers generated in the j-th loop are calculated by computing $(\lfloor a_0 \cdot s_j \rceil_p, \cdots, \lfloor a_{m-1} \cdot s_j \rceil_p)$, which are divided into two parts: one is used as the output, and the other part is used to generate a new s_{j+1}, which will be calculated with $\boldsymbol{a}$ in the next loop, and this process iterates with $\boldsymbol{a}$ unchanged. For the same s_j, we perform computations with each a_i, effectively reusing s. Since $\boldsymbol{a}$ remains unchanged, we operate on different continuously updated s_j for the same a_i, which means reusing $\boldsymbol{a}$. Through the combination of this novel framework with the hiding of $\boldsymbol{a}$, the performance of LEAP.PRNG is greatly improved compared to other lattice-based PRNGs.

Shorter Key. While hiding $\boldsymbol{a}$ achieve high performance, it comes at the cost of treating $\boldsymbol{a}$ as the key, resulting in an increased key size. To address this issue, we construct another PRNG with a shorter key to generate $\boldsymbol{a}$ and s. We refer to this key generation stage as LEAP.GEN and the high-performance pseudorandom number generator as LEAP.PRNG. The LEAP.GEN maintains high performance of the scheme while significantly mitigating the increase in key size. It can be viewed as a simplified version of the LEAP.PRNG. To reduce the key size, we replace the use of multiple polynomials $\boldsymbol{a} \in \mathcal{R}_q^m$ with a single polynomial $a' \in \mathcal{R}_q$. The LEAP.GEN uses the same framework as LEAP.PRNG to generate pseudorandom numbers, and then samples the $\boldsymbol{a}$ and s for LEAP.PRNG. To further reduce the key size, LEAP.GEN uses an extension of the NTRU assumption called Vectorized-NTRU (VNTRU), which allows for a smaller key size when generate pseudorandom numbers, thus meeting the requirements for constructing the polynomial a'. The decisional Vectorized-NTRU assumption states that distribution of $(g_0/f, g_1/f, \cdots, g_{k-1}/f) \in \mathcal{R}_q^k$ is computationally indistinguishable from the uniform random distribution on $\mathcal{R}_q^k$. However, unlike the RLWR assumption, which allows the simultaneous reuse of both $\boldsymbol{a}$ and s, the VNTRU assumption only permits the reuse of f. Simultaneous reuse of g would expose the scheme to a distinguishing attack. The inability to simultaneously reuse parameters makes the performance of LEAP.GEN inferior to that of LEAP.PRNG.

1.3 Comparison to the State of the Art

In our comparative analysis, we compare our scheme with the SPRING family[1], [CS20][2], AES-CTR[3] and Keccak[4] in terms of key size, public parameter size, and performance at the same security level on an identical machine. Except for AES and Keccak, the size of the key and public parameter is the number of bytes in the compressed representation of elements in $\mathcal{R}_q$. AES-CTR is one of the fastest and most widely used classical PRNGs, benefiting from the dedicated `AES-NI` instruction set. `AES-NI` accelerates the encryption and decryption operations of the AES algorithm, enhancing the performance. The Keccak algorithm, selected as the winner of the SHA-3 Cryptographic Hash Algorithm Competition by NIST, is also extensively employed in cryptographic schemes to produce cryptographically secure pseudorandom numbers. We test various highly optimized implementations of AES-CTR and Keccak at different security levels.

In our scheme, we present performance results for plain `C`, `AVX2`, and `AVX512` implementations. The performance data for LEAP in Table 1 corresponds to the optimal m value on our test machine (specifically, with m set to 208). Under the new design, variations in parameter choices across the three security levels are minimal, resulting in similar performance for large values of m (see Section 5.2 for details). Table 1 shows the specific data comparison.

[1]The testing codes for SPRING family are obtained from https://github.com/cbouilla/spriiiiiiiing

[2]The testing code for [CS20] is obtained from https://github.com/BeJade/mlwr-prf

[3]The testing code for AES-CTR is obtained from the OpenSSL version available at https://bench.cr.yp.to/primitives-stream.html

[4]The testing code for Keccak is obtained from https://github.com/XKCP/XKCP

Table 1: Comparison of SPRING family, [CS20], AES-CTR, Keccak and LEAP

Scheme	Key (Bytes)	Public (Bytes)	Cycles/byte				Security Level
			C	AES-NI	AVX2	AVX512	
Our LEAP-128	224	0	13.21	-	1.61	1.14	128-bit
SPRING-BCH	8198	129	-	-	40.65	-	
SPRING-CRT	18444	145	-	-	20.85	-	
SPRING-RS	385	0	-	-	5.32	-	
[CS20] module-LWR	192	24576	-	-	31.39	-	
[CS20] LWR	200	2944000	-	-	51.15	-	
AES128-CTR	16	0	5.91	0.33	-	-	
Keccakc512	16	0	40.28	-	9.25	5.56	
Our LEAP-192	576	0	13.25	-	1.61	1.14	192-bit
AES192-CTR	24	0	6.97	0.41	-	-	
Keccakc768	24	0	52.01	-	11.28	7.84	
Our LEAP-256	800	0	13.42	-	1.61	1.14	256-bit
AES256-CTR	32	0	8.05	0.45	-	-	
Keccakc1024	32	0	74.31	-	16.17	11.26	

2 Preliminaries

2.1 Notations

Let $\mathbb{Z}_q$ represent the residue class ring modulo q, where $q \geq 1$ is an integer. The ring $\mathcal{R} = \mathbb{Z}[x]/(x^n+1)$ and $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n+1)$ denote the polynomial ring modulo x^n+1 with cofficients over $\mathbb{Z}$ and $\mathbb{Z}_q$, respectively. We denote by $\mathcal{R}_q^*$ the set of invertible elements in $\mathcal{R}_q$. Regular type letters (e.g., a) represent elements in $\mathcal{R}$ or $\mathcal{R}_q$ (which includes elements in $\mathbb{Z}$ or $\mathbb{Z}_q$), bold lowercase letters (e.g., $\boldsymbol{a}$) denote vectors with coefficients in $\mathcal{R}$ or $\mathcal{R}_q$, and bold uppercase letters (e.g., $\mathbf{A}$) denote matrices with elements in $\mathbb{Z}$ or $\mathbb{Z}_q$. The notation $a[i]$ represents the coefficients of x^i in a polynomial a, the notation $\mathbf{v}[i]$ represents the i-th entry in a vector $\mathbf{v}$ using zero-based indexing, and the notation $\mathbf{M}[i]$ represents the i-th row vector in matrix $\mathbf{M}$ using zero-based indexing. We use $\mathbf{0}$ to denote the matrix full of zero entires. For a polynomial $a \in \mathcal{R}$, we denote by $\mathbf{Vec}(a) = (a[0], a[1], \ldots, a[n-1]) \in \mathbb{Z}^n$ the row vector of coefficients of a and $||(a[0], a[1], \ldots, a[n-1])|| = (\sum_{i=0}^{n-1} a[i]^2)^{\frac{1}{2}}$ the l_2 norm of corresponding vector. For two polynomials $a, b \in \mathcal{R}$, we use $\mathbf{Mat}(a) = \left(\mathbf{Vec}(a)^T, \mathbf{Vec}(a \cdot x)^T, \ldots, \mathbf{Vec}(a \cdot x^{n-1})^T\right)^T \in \mathbb{Z}^{n \times n}$ to represent the matrix such that $\mathbf{Vec}(a) \cdot \mathbf{Mat}(b) = \mathbf{Vec}(a \cdot b)$, where x is the formal variable corresponding to the ring $\mathcal{R}$ and the upper index T means the transpose matrix. For clarity, since $a \cdot x^i$ remains an element of the ring $\mathcal{R}$, its vector representation is given by $\mathbf{Vec}(a \cdot x^i) = (-a[n-i], \ldots, -a[n-1], a[0], \ldots, a[n-i-1])$.

We define rounding function $\lfloor \cdot \rceil_p : \mathbb{Z}_q \to \mathbb{Z}_p$ $(q \geq p \geq 2)$ as $\lfloor x \rceil_p = \lfloor (p/q) \cdot \bar{x} \rceil \bmod p$, where $\bar{x} \in \mathbb{Z}$ and $\bar{x} \equiv x \bmod q$. The rounding function is applied coefficient-wise to the ring element. If A is a set, $a \leftarrow A$ represents that a is drawn uniformly at random from A. If A is a distribution, $a \leftarrow A$ represents that a is drawn according to A. The distribution U_n for an integer n denotes the uniform distribution over $\{0, 1, \cdots, n-1\}$. We denote by $\mathbf{B}^t$ the set of byte arrays of length t, and $\boldsymbol{b}^t \in \mathbf{B}^t$ represents a byte array of length t. We use $\boldsymbol{b}[i:j]$ to represent the elements of array $\boldsymbol{b}$ with index i to j. & represents the logical AND operation. >> and << represent the right and left shift, respectively.

2.2 NTRU

The NTRU problem was introduced by Hoffstein, Pipher, and Silverman [HPS98]. It is considered one of the prominent candidates for post-quantum cryptography due to its

security guarantees and efficiency. The works [DLP14, FHK+19, PMS21] have extensively utilized and studied the decisional NTRU problem, establishing a robust theoretical foundation. As mentioned in [PMS21], an average-case search variant of the NTRU problem can be reduced to the worst-case approximate Shortest Vector Problem (SVP) over ideal lattices when using a small secret distribution. The definition of the decisional NTRU problem is as follows:

Definition 1 (**Decisional NTRU**)**.** Let n and q be positive integers. For a distribution D_f over $\mathcal{R}_q^*$ and a distribution D_g over $\mathcal{R}$, the decisional NTRU problem $\mathsf{DNTRU}_{n,q,D_f,D_g}$ is to distinguish (with advantage non-negligible in n) two distributions of $g/f \in \mathcal{R}_q$ and u drawn uniformly at random from $\mathcal{R}_q$, where f is drawn according to D_f and g is drawn according to D_g.

In the LEAP scheme, we employ a variant of the NTRU problem, referred to as the Vectorized-NTRU (VNTRU) problem in [XZD+23]. As mentioned in [GGH+19], this problem is merely a syntactic extension of the standard NTRU problem and does not fundamentally alter the security assumptions. The decisional VNTRU problem is defined as follows:

Definition 2 (**Decisional VNTRU**)**.** Let n and q be positive integers. For a distribution D_f over $\mathcal{R}_q^*$ and a distribution D_g over $\mathcal{R}$, the decisional VNTRU problem $\mathsf{DVNTRU}_{n,q,k,D_f,D_g}$ is to distinguish (with advantage non-negligible in n) two distributions of $(g_0/f, g_1/f, \cdots, g_{k-1}/f) \in \mathcal{R}_q^k$ and $\boldsymbol{u}$ drawn uniformly at random from $\mathcal{R}_q^k$, where f is drawn according to D_f and $(g_0, g_1, \cdots, g_{k-1})$ is drawn according to D_g^k.

2.3 Ring Learning With Rounding

Banerjee, Peikert, and Rosen [BPR12] proposed a derandomization based variant of the Ring Learning With Errors (RLWE) problem called Ring Learning With Rounding (RLWR). The core idea of the derandomization technique for RLWE simplifies the process by replacing the addition of a small random error term with deterministic rounding $a \cdot s$ to a relatively nearby value. Furthermore, the study conducted by [BPR12] also provides a security analysis of the RLWR problem with sub-exponential modulus q. Subsequent research [AKPW13, BGM+16, ASA16, LW20] showed that, under the premise of the polynomial number of samples, polynomial moduli are sufficient to ensure the security of the RLWR problem. The decisional RLWR problem is defined as follows:

Definition 3 (**Decisional RLWR**)**.** Let n, q and p be positive integers, where $q \geq p$. For a distribution D_s over $\mathcal{R}$, the decisional RLWR problem $\mathsf{DRLWR}_{n,q,p,D_s}$ is to distinguish (with advantage non-negligible in n) two distributions of $(a, \lfloor a \cdot s \rceil_p)$ and (a, u) drawn uniformly at random from $\mathcal{R}_q \times \mathcal{R}_p$, where a is drawn uniformly at random from $\mathcal{R}_q$ and s is drawn according to D_s.

2.4 Pseudorandom Number Generators

Pseudorandom number generators (PRNGs) can employ a limited quantity of true randomness to produce a substantial volume of pseudorandomness. In other words, a PRNG is an efficient and deterministic algorithm that takes a short random sequence *seed* as input and produces a long pseudorandom sequence that is computationally indistinguishable from a truly uniform random sequence. Therefore, in any cryptographic application, if using a truly random sequence is secure, then replacing it with a pseudorandom sequence remains secure. By using a PRNG, the desired cryptographically secure random sequence can be generated efficiently with less reliance on true randomness. Definition 4 provides the formal definition of the PRNG introduced by Blum, Micali [BM82], and Yao [Yao82].

Definition 4 (**PRNG**)**.** A polynomial-time deterministic algorithm G is a pseudorandom number generator if it satisfies the following:

- **Expansion:** There exists a function $l(\cdot) : \mathbb{N} \to \mathbb{N}$, satisfying $l(n) > n$ for all n, such that for any s, $|G(s)|{=}l(|s|)$. The function $l(\cdot)$ is called the expansion factor.
- **Pseudorandom:** For a uniform distribution u_n over $\{0,1\}^n$, $\{G(u_n)\}_{n\in\mathbb{N}}$ and $\{u_{l(n)}\}_{n\in\mathbb{N}}$ are computationally indistinguishable.

2.5 Number Theoretic Transform

The Number Theoretic Transform (NTT) is an efficient computational method derived from the Fast Fourier Transform (FFT) algorithm, tailored for polynomial multiplication over finite fields. By applying the Chinese Remainder Theorem for Rings, the following isomorphic mapping can be obtained:

$$\pi : \mathbb{Z}_q[x]/f(x) \to \mathbb{Z}_q[x]/g(x) \times \mathbb{Z}_q[x]/h(x),$$

where $f(x) = g(x) \cdot h(x)$ with coprime $g(x)$ and $h(x)$. This mapping can be extended recursively by decomposing factors $h(x)$ and $g(x)$ into more factors. The NTT algorithm offers advantages for computing multiplication or inversion in $\mathcal{R}_q$ by evaluating this map, performing the corresponding operations on these factors, and subsequently applying the inverse of the map (known as Inverse Number Theoretic Transform (INTT) algorithm.

In cryptography, a frequently used ring is $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$, where n is a power-of-2 and $f(x) = x^n + 1$ is the $2n\text{-}th$ cyclotomic polynomial. If $2n \mid (q-1)$, $\mathbb{Z}_q$ contains a $2n$-th primitive roots of unity denoted as ζ (i.e., $\zeta^{2n} \equiv 1 \bmod q$ and $\zeta^n \equiv -1 \bmod q$). By using this mapping, we can obtain

$$\mathbb{Z}_q[x]/(x^n + 1) \to \mathbb{Z}_q[x]/(x^{\frac{n}{2}} - \zeta^{\frac{n}{2}}) \times \mathbb{Z}_q[x]/(x^{\frac{n}{2}} + \zeta^{\frac{n}{2}}).$$

Clearly, $(x^{\frac{n}{2}} - \zeta^{\frac{n}{2}})$ and $(x^{\frac{n}{2}} + \zeta^{\frac{n}{2}})$ can be continuously decomposed until n polynomials of degree 1, which is regarded as the complete NTT. Through the aforementioned process, a polynomial can be decomposed into $\log n$ layers. If using the Cooley-Tukey [CT65] and Gentleman-Sande [GS66] butterfly transformations, each layer requires only $n/2$ multiplications and n additions. Therefore, the NTT algorithm reduces the time complexity of polynomial multiplication from $O(n^2)$ to $O(n \log n)$. If $2n \nmid (q-1)$, which means the modulus does not allow for a complete NTT, the polynomial can only be decomposed into n/k polynomials of degree k, which is referred to as incomplete NTT.

The multiplication of polynomials $a \in \mathcal{R}_q$ and $b \in \mathcal{R}_q$ can be computed using NTT given by

$$a \cdot b = \mathsf{INTT}_{q,n}(\mathsf{BaseMul}_{q,n}(\mathsf{NTT}_{q,n}(a), \mathsf{NTT}_{q,n}(b))),$$

where the $\mathsf{BaseMul}_{q,n}$ algorithm performs a point-wise vector multiplication for the complete NTT or a schoolbook polynomial multiplication for the incomplete NTT. The subscripts $_{q,n}$ indicate the modulus q and the order n of the polynomial used in the computation. Similarly, the computation of the inverse of a polynomial on $\mathcal{R}_q$ using NTT can be expressed as

$$a^{-1} = \mathsf{INTT}_{q,n}(\mathsf{BaseInv}_{q,n}(\mathsf{NTT}_{q,n}(a)),$$

and the $\mathsf{BaseInv}_{q,n}$ algorithm calculates a point-wise vector inversion in $\mathbb{Z}_q$ for the complete NTT or a schoolbook polynomial inversion for the incomplete NTT.

3 LEAP PRNG

In this section, we give a detailed algorithm description and parameter selection strategy of a PRNG scheme, called LEAP. The LEAP scheme consists of two algorithms: 1) the key

generation algorithm, LEAP.GEN, which is based on the VNTRU and RLWR assumptions; and 2) the high-performance pseudorandom numbers generation algorithm, LEAP.PRNG, which is based on the RLWR assumption. First of all, we give an overview of the LEAP scheme.

Reuse $\boldsymbol{a}$ and $\boldsymbol{s}$. To improve the speed of generating pseudorandom numbers, the LEAP.PRNG algorithm reuses $\boldsymbol{a}$ and s separately while ensuring security. In LEAP.PRNG, we do not update $\boldsymbol{a}$ and only efficiently update s using the generated partial pseudorandom numbers. The structure of LEAP.PRNG is shown in Figure 1.

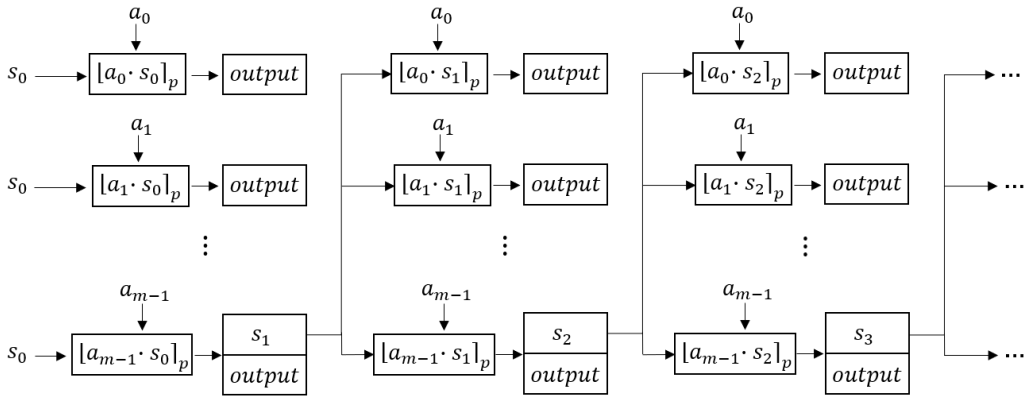

Figure 1: The Structure of LEAP.PRNG

As illustrated in the figure, we utilize the approach of reusing each a_j across multiple different s_i within each row (e.g., the first row outputs $\lfloor a_0 \cdot s_0 \rceil_p, \lfloor a_0 \cdot s_1 \rceil_p, \lfloor a_0 \cdot s_2 \rceil_p, \cdots$). Meanwhile, within each column, we reuse each s_i across several independent a_j (e.g., for a given s_i, the output is $\lfloor a_0 \cdot s_i \rceil_p, \lfloor a_1 \cdot s_i \rceil_p, \cdots \lfloor a_{m-1} \cdot s_i \rceil_p$). It is only necessary to compute the NTT-representation of s_i once and operate it with various directly generated NTT-representation of a_j. By simultaneously reusing $\boldsymbol{a}$ and s, it significantly reduces the computational cost of the NTT calculation and greatly improves performance.

Reduce key size. The LEAP.PRNG treats $\boldsymbol{a}$ as the key, demonstrating good performance but requiring a large key size. To address this issue, we utilize LEAP.GEN algorithm to generate the $\boldsymbol{a}$ and s for LEAP.PRNG. The structure of LEAP.GEN is illustrated in Figure 2.

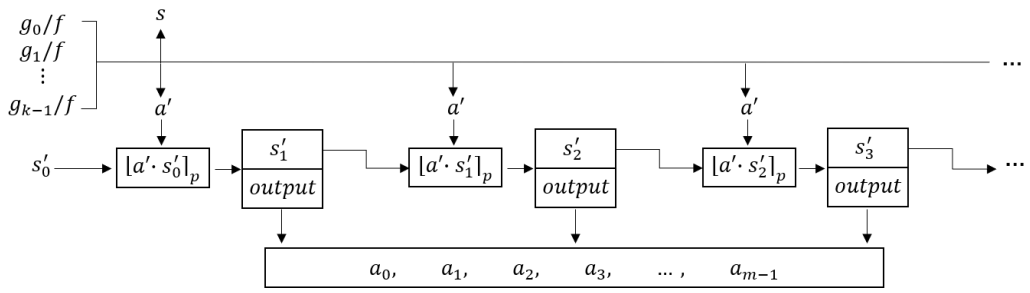

Figure 2: The Structure of LEAP.GEN

The LEAP.GEN algorithm can be viewed as a mini-scale version of LEAP.PRNG. Unlike LEAP.PRNG, which uses $\boldsymbol{a} \in \mathcal{R}_q^m$, the key size in LEAP.GEN is reduced by using a single polynomial $a' \in \mathcal{R}_q$. Similar to LEAP.PRNG, LEAP.GEN maintains a' unchanged while updating s'_i, implying the reuse of a'. The result of $\lfloor a' \cdot s'_i \rceil_p$ is split into two parts: one serves as the output of pseudorandom numbers, and the other is used to generate the new s'_{i+1}. Once the pseudorandom numbers are obtained, the key $\boldsymbol{a}$ for LEAP.PRNG is sampled according to the required distribution. To further compress the key size, we use the VNTRU assumption, which requires a smaller key size, to generate the required a'. To optimize the utilization of the pseudorandom numbers generated by VNTRU, we also employ them to generate the key s required by LEAP.PRNG.

Reduce NTTs. The NTT algorithm is utilized in our scheme to perform polynomial multiplication. After conducting a detailed analysis, we reduce certain NTT computations within LEAP. Specifically, when generating pseudorandom numbers based on the VNTRU assumption, we directly use values in the NTT domain as pseudorandom numbers, bypassing the inverse NTT operation (Algorithm 3 step 6). This approach is possible because the inverse NTT operation does not depend on any additional information and can be executed independently by any adversary. Omitting the inverse NTT operation not only enhances performance but also does not cause security loss. Furthermore, similar to Kyber [BDK+18], we directly sample the NTT-representation of a' using the pseudorandom numbers, reducing computational overhead. Likewise, LEAP.GEN generates $\boldsymbol{a}$ directly in the NTT domain, eliminating the need to compute $\boldsymbol{a}$ in the NTT domain in LEAP.PRNG, thus further improving computational efficiency.

3.1 Algorithm Description

Rejection Sampling In practical PRNGs, pseudorandom numbers are typically packed into a byte array. However, under the VNTRU and RLWR assumptions, we generate a uniform distribution over $\mathcal{R}_q$, where the polynomial coefficients are uniformly distributed in $\mathbb{Z}_q$. To map the uniform distribution from $\mathcal{R}_q$ to a uniform byte array, a secure and efficient mapping from $\mathbb{Z}_q$ to $\mathbb{Z}_{256}$ is required, which presents a challenging task. After several attempts, we found that rejection sampling provides the most effective solution. For the rejection sampling from $\mathbb{Z}_q$ to $\mathbb{Z}_{256}$, we fix an integer q', where q' is a multiple of 256. The rejection sampling process proceeds as follows: for an integer drawn from the uniform distribution on $\mathbb{Z}_q$, reject it if it exceeds q', and if it does not, take the lower 8 bits as the output. This process ensures that the outputs are uniformly distributed over $\mathbb{Z}_{256}$, providing a secure and efficient mapping from $\mathbb{Z}_q$ to $\mathbb{Z}_{256}$.

Parameters Description Before describing the algorithm, we first provide an overview of the parameters used.

- N: the order of the ring $\mathcal{R}_Q = \mathbb{Z}_Q[x]/(x^N + 1)$
- Q: the modulus of the ring $\mathcal{R}_Q = \mathbb{Z}_Q[x]/(x^N + 1)$
- Q': an integer used for rejection sampling in LEAP.GEN
- k: the dimension of polynomial vector $(g_0, g_1, \cdots, g_{k-1}) \in \mathcal{R}_Q^k$
- u_f: the bit length for a coefficient of f
- u_g: the bit length for a coefficient of g
- $u_{s'}$: the bit length for a coefficient of s'
- n: the order of the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$

- q: the modulus of the ring $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$
- m: the dimension of the polynomial vector $\boldsymbol{a} \in \mathcal{R}_q^m$
- q': an integer used for rejection sampling in LEAP.PRNG
- p: an integer used for rounding function
- u_s: the bit length for a coefficient of s
- l: the length of the output pseudorandom byte array

To begin with, we define two subroutines: Bin2Poly and Bin2qPoly. The Bin2Poly algorithm, introduced in Algorithm 1, is used to generate a polynomial $c \in \mathcal{R}_{2^u} = \mathbb{Z}_{2^u}[x]/(x^n + 1)$ using a byte array. The Bin2qPoly subroutine, introduced in Algorithm 2, is used to generate a polynomial $c \in \mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$ using a byte array[5].

Algorithm 1 Bin2Poly

Input:
Integers u and n
A byte array $\boldsymbol{b} \in \mathbf{B}^{\lceil n/\lfloor 8/u \rfloor \rceil}$
Output:
A polynomial $c \in \mathcal{R}_{2^u} = \mathbb{Z}_{2^u}[x]/(x^n + 1)$

$t = \lfloor 8/u \rfloor$
for $i = 0$ to $\lceil n/t \rceil - 1$ **do**
  for $j = 0$ to $t - 1$ **do**
    if $i \cdot t + j < n$ **then**
      $c[i \cdot t + j] = \boldsymbol{b}[i] \& (2^u - 1))$
      $\boldsymbol{b}[i] = \boldsymbol{b}[i] >> u$
    end if
  end for
end for
return c.

Algorithm 2 Bin2qPoly

Input:
Integers q and n
A byte array $\boldsymbol{b} \in \mathbf{B}^*$
Output:
A polynomial $c \in \mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$

$i = j = 0$
while $j < n$ **do**
  $val = \boldsymbol{b}[i] + \boldsymbol{b}[i + 1] \cdot 256$
  if $val < q$ **then**
    $c[j] = val$
    $j = j + 1$
  end if
  $i = i + 2$
end while
return c.

[5]Since the modulus q in LEAP is constrained to 16 bits, two bytes are concatenated to generate the coefficient of polynomial.

Algorithm 3 LEAP.GEN

Input:

Integers N, Q', k, $u_{s'}$, n, m, q, q', p and u_s, where q' and Q' are multiples of 256
A polynomial $f \in \mathcal{R}_{2^{u_f}} = \mathbb{Z}_{2^{u_f}}[x]/(x^N+1)$ invertible over $\mathcal{R}_Q = \mathbb{Z}_Q[x]/(x^N+1)$
A polynomial vector $(g_0, g_1, \cdots, g_{k-1})$, where $g_i \in \mathcal{R}_{2^{u_g}} = \mathbb{Z}_{2^{u_g}}[x]/(x^N+1)$
A polynomial $s' \in \mathcal{R}_{2^{u_{s'}}} = \mathbb{Z}_{2^{u_{s'}}}[x]/(x^n+1)$

Output:

A polynomial vector $\boldsymbol{a}$ in NTT domain $\hat{\boldsymbol{a}} \in \mathcal{R}_q^m$, where $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n+1)$
A polynomial $s \in \mathcal{R}_{2^{u_s}} = \mathbb{Z}_{2^{u_s}}[x]/(x^n+1)$

$\hat{f} = \mathsf{NTT}_{Q,N}(f)$
$\hat{f}^{-1} = \mathsf{BaseInv}_{Q,N}(\hat{f})$
$t = 0$
for $i = 0$ to $k-1$ **do**
  $\hat{g}_i = \mathsf{NTT}_{Q,N}(g_i)$
  $\hat{h} = \mathsf{BaseMul}_{Q,N}(\hat{g}_i, \hat{f}^{-1})$
  for $j = 0$ to $N-1$ **do**
    if $\hat{h}[j] < Q'$ **then**
      $\boldsymbol{b}[t] = \hat{h}[j]\&255$ ▷ extract pseudorandom byte
      $t = t+1$
    end if
  end for
end for
$s = \mathsf{Bin2Poly}(u_s, n, \boldsymbol{b}[t-(u_s \cdot n/8) : t])$
$\hat{a'} = \mathsf{Bin2qPoly}(q, n, \boldsymbol{b})$ ▷ $a' \in \mathcal{R}_q = \mathbb{Z}_q[x]/x^n+1$
$i = 0$
while $i < m \cdot n$ **do**
  $\hat{s'} = \mathsf{NTT}_{q,n}(s')$
  $\hat{h} = \mathsf{BaseMul}_{q,n}(\hat{a'}, \hat{s'})$
  $h = \mathsf{INTT}_{q,n}(\hat{h})$
  $t = 0$
  for $j = 0$ to $n-1$ **do**
    if $h[j] < q'$ **then**
      $h[j] = \lfloor h[j] \rceil_p$ ▷ $\lfloor h[j] \rceil_p = \lfloor (p/q') \cdot (x \bmod q') \rceil \bmod p$
      $\boldsymbol{b}[t] = h[j]\&255$ ▷ extract pseudorandom byte
      $t = t+1$
    end if
  end for
  $s' = \mathsf{Bin2Poly}(u_{s'}, n, \boldsymbol{b}[t-(u_{s'} \cdot n/8) : t])$
  $t = t-(u_{s'} \cdot n/8)$
  $j = 0$
  while $j < t$ **do**
    $val = \boldsymbol{b}[j] + (\boldsymbol{b}[j+1] \cdot 256$
    if $i < m \cdot n$ **and** $val < q$ **then**
      $\hat{\boldsymbol{a}}[i] = val$
      $i = i+1$
    end if
    $j = j+2$
  end while
end while
return $(\hat{\boldsymbol{a}}, s)$.

Algorithm 3 describes the deterministic algorithm LEAP.GEN, which is used to generate the key for LEAP.PRNG. The core steps of the algorithm are as follows: 1) Generate pseudorandom bytes based on the VNTRU assumption (steps 1-13); 2) Use the generated pseudorandom bytes to sample the s, the key of LEAP.PRNG, and a' using the Bin2Poly (step 14) and Bin2qPoly (step 15) algorithms, respectively; 3) Use the generated a' and the input s' to cyclically generate pseudorandom bytes based on the RLWR assumption (steps 18-28); 4) At the end of each round, update s' with the generated pseudorandom bytes (step 29); 5) Use the generated pseudorandom bytes to produce the NTT-representation $\hat{\boldsymbol{a}}$ (steps 31-39). As previously stated, the polynomial a' is sampled using values in the NTT domain as pseudorandom numbers, without performing the inverse NTT operation (step 6), to enhance performance. The key idea is that the NTT-representation of the polynomial h is already uniformly distributed over the ring $\mathcal{R}_Q$, due to the properties of the NTT. The inverse NTT deterministically maps one uniformly distributed element in $\mathcal{R}_Q$ to another and is a public algorithm. As a result, even if the inverse NTT is applied to obtain h, an adversary can easily compute the NTT of h to recover its NTT-representation. Therefore, the inverse NTT step can be safely omitted to improve performance without compromising security. For the polynomial s, we sample it in the coefficient domain rather than directly in the NTT domain. This choice is motivated by the fact that sampling s directly in the NTT domain requires more pseudorandom bytes, which in turn increases the value of k in the VNTRU assumption (i.e., the dimension of the polynomial vector $(g_0, g_1, \ldots, g_{k-1})$). A larger k results in an increased key size. To preserve key compactness, we therefore sample s in the coefficient domain and subsequently apply the NTT transformation. With appropriate parameter selection, excessive bit rejection leads to an exponentially small probability of failure in generating s (step 14), a' (step 15), and s' (step 29), and a more detailed analysis is provided in Section 4.1.

Algorithm 4 LEAP.PRNG

Input:

Integers l, m, n, q, q', p and u_s, where q' is a multiple of 256

A polynomial vector $\boldsymbol{a}$ in NTT domain $\hat{\boldsymbol{a}} \in \mathcal{R}_q^m$, where $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n+1)$

A polynomial $s \in \mathcal{R}_{2^{u_s}} = \mathbb{Z}_{2^{u_s}}[x]/(x^n+1)$

Output:

A pseudorandom byte array $\boldsymbol{b}^l$

1: $t = 0$
2: **while** $t < l$ **do**
3: $\quad \hat{s} = \mathsf{NTT}_{q,n}(s)$
4: $\quad$ **for** $i = 0$ to $m-1$ **do**
5: $\quad\quad \hat{h} = \mathsf{BaseMul}_{q,n}(\hat{\boldsymbol{a}}[i], \hat{s})$
6: $\quad\quad h = \mathsf{INTT}_{q,n}(\hat{h})$
7: $\quad\quad$ **for** $j = 0$ to $n-1$ **do**
8: $\quad\quad\quad$ **if** $h[j] < q'$ **then**
9: $\quad\quad\quad\quad h[j] = \lfloor h[j] \rceil_p$ ▷ $\lfloor h[j] \rceil_p = \lfloor (p/q') \cdot (x \bmod q') \rceil \bmod p$
10: $\quad\quad\quad\quad \boldsymbol{b}[t] = h[j] \& 255$ ▷ extract pseudorandom byte
11: $\quad\quad\quad\quad t = t + 1$
12: $\quad\quad\quad$ **end if**
13: $\quad\quad$ **end for**
14: $\quad$ **end for**
15: $\quad s = \mathsf{Bin2Poly}(n, u_s, \boldsymbol{b}[t - (u_s \cdot n/8) : t])$
16: $\quad t = t - (u_s \cdot n/8)$
17: **end while**
18: **return** $\boldsymbol{b}[0 : l-1]$.

Algorithm 4 provides the description of the deterministic algorithm LEAP.PRNG. It takes $\hat{\boldsymbol{a}}$ and s generated by LEAP.GEN as inputs, and outputs an array of pseudorandom bytes. At the end of each round, we use the Bin2Poly algorithm (step 15) to generate a new s based on the distribution of s, using the generated pseudorandom numbers. Similarly, with appropriate parameter selection, the probability of being unable to generate a new s due to excessive bit rejections is exponentially small (see Section 4.1). Due to the use of rejection sampling, the length of output in each loop may be less than n. However, the PRNG is expected to return a specified quantity of output numbers. Therefore, we use l to control the output with a fixed length and truncate the array $\boldsymbol{b}$, outputting only the first l elements.

3.2 Parameter Selection

In general, whether it is LEAP.GEN or LEAP.PRNG, we mainly consider the trade-off between security, NTT computation efficiency, and rejection rate of rejection sampling in the parameter selection strategy.

The order of the ring [N and n] Fast computation using the NTT algorithm require the use of power-of-2 cyclotomic polynomials, implying that both n and N must be powers of 2, which restricts the choice of order. The primary goal in selecting the orders is to minimize it while ensuring the desired level of secuiity is preserved. Polynomials with smaller orders can be computed more efficiently, thereby enhancing the overall computational performance of the scheme.

The modulus of the ring [Q and q] Our strategy for selecting these moduli focuses on two main considerations. First, the objective is to select the smallest modulus possible. A smaller modulus allows for the use of the NTT optimization technique called Lazy Reduction, which improves NTT performance (see Section 5.1 for a detailed description). However, blindly pursuing an excessively small modulus is not advisable. Since polynomial multiplication is the most computationally expensive part of the scheme, the performance of NTT directly impacts overall efficiency. Fully decomposed NTT calculations are more efficient than incomplete NTT. Due to security constraints, the order of the ring must not be too small. As the modulus decreases, only more incomplete NTT can be performed for relatively large order, leading to a loss in performance. Furthermore, due to the condition that $Q > Q'$ in LEAP.GEN and $q > q' > p$ in LEAP.PRNG, as Q and q decrease, the available Q' and p may fall below 256. This necessitates bit concatenation during pseudorandom byte generation, which reduces algorithm performance. Therefore, an excessively small modulus should be avoided. Second, the goal is to minimize the rejection rate in rejection sampling. We aim to minimize the rejection rates $(Q - Q')/Q$ and $(q - q')/q$, thereby enhancing output efficiency while maintaining the required security level.

The dimension of $\boldsymbol{g} \in \mathcal{R}_Q^k$ [k] To generate sufficient pseudorandom bytes for a' in LEAP.GEN after rejection sampling, we need to select a value of k that ensures the probability of failure for the uniform sampling algorithm 2 in $\mathcal{R}_q$ is exponentially small. Specifically, we calculate this probability using the binomial distribution and ensure that the probability of failing to generate a' is less than 2^{-256}. Additionally, k should be as small as possible to minimize the key size, thus providing an optimal value for k.

Rounding function parameter [p] To generate pseudorandom bytes, we need to map an element from the uniform distribution over $\mathbb{Z}_p$ to an element from a uniform distribution over $\mathbb{Z}_{256}$. To do so efficiently, we set p to a multiple of 256, aligning it with the byte size. This enables the direct extraction of the lowest 8 bits, facilitating the efficient generation of pseudorandom bytes.

Rejection sampling paramater [Q' and q'] To efficiently generate pseudorandom bytes, we employ a rejection sampling technique (described in detail in 3.1). We set the rejection sampling parameter Q' to be a multiple of 256, ensuring that the lowest 8 bits

can be directly extracted to generate a pseudorandom byte. Furthermore, for q', we not only ensure it is a multiple of 256 but also set it to a power-of-2 multiple of p, allowing for fast rounding operation through bit shift in the rounding function, thereby further improving performance.

Distributions for polynomial f and g [u_f and u_g] Through analysis, we conclude that the uniform distribution over $\mathcal{R}_{2^d}$ allows for a short key size while ensuring high security. Therefore, for the distributions of polynomials f and g, we use uniform distributions over $\mathcal{R}_{2^{u_f}}$ and $\mathcal{R}_{2^{u_g}}$, respectively. The selection criteria for parameters u_f and u_g are unified, aiming to choose the smallest possible values while maintaining security, thus achieving a smaller key size.

Distributions for polynomial s' and s [$u_{s'}$ and u_s] The distributions for the polynomials s' in LEAP.GEN and s in LEAP.PRNG follow a consistent selection strategy. As shown in Figures 1 and 2, we use the generated pseudorandom numbers to update the polynomials s' and s. To ensure efficient update operation, we adopt a uniform power-of-2 distribution for s' and s, enabling efficient sampling of the polynomial using bit shifting and AND operation (Algorithm 1). The parameters u_s and u'_s denote the bit lengths of the coefficients of the input polynomials s and s', used in LEAP.PRNG and LEAP.GEN, respectively. Since the adversary gains access to more RLWR samples in LEAP.PRNG than in LEAP.GEN when $m > 1$, a larger standard deviation for the distribution of s (i.e., a larger u_s) is sufficient for LEAP.PRNG to maintain the same security level. Both $u_{s'}$ and u_s are required to be minimized while maintaining the necessary security level, thereby reducing the number of pseudorandom numbers needed for updates and improving output efficiency.

The dimension of $a \in \mathcal{R}_q^m$ [m] The selection of m in the LEAP.PRNG algorithm allows for achieving different performances and entails varying memory requirements. The optimal value of m depends on the software and hardware conditions of the specific test machine. In actual applications, m can be adjusted according to the specific requirements of each application.

Table 2 presents parameter sets for three security levels: LEAP-128, LEAP-192, and LEAP-256, which correspond to security levels of 128-bit, 192-bit, and 256-bit, respectively. The 'Key Size' indicates the size of key for each security level. For a detailed explanation of all parameters, see 3.1.

Table 2: Parameters Sets for LEAP

Scheme	Key size (Bytes)	N	Q	Q'	k	u_f	u_g	$u_{s'}$	n	q	q'	p	u_s	m
LEAP-128	224	512	257	256	2	1	1	1	256	3329	3072	1536	1	208
LEAP-192	576	512	257	256	2	2	3	2	256	3329	3072	1536	3	208
LEAP-256	800	512	257	256	2	3	4	3	256	3329	3072	1536	4	208

We also calculate the concrete the probability of being unable to generate polynomials due to excessive bit rejections during rejection sampling for the specified parameter sets. We use δ_1 to denote the probability that LEAP.GEN fails to generate the polynomials s, a', and s', and we use δ_2 to denote the probability that LEAP.PRNG fails to update the polynomial s_i while running at least 2^{256} iterations. The specific values of δ_1 and δ_2 across different security levels are presented in Table 3. All computed probabilities are bounded below 2^{-256} to align the security level requirements, ensuring that our scheme operates continuously with negligible probability of failure.

Table 3: Failure Probabilities of Rejection Sampling

Scheme	δ_1	δ_2
LEAP-128	$2^{-415.7}$	$2^{-196009.3}$
LEAP-192	$2^{-344.4}$	$2^{-195156.1}$
LEAP-256	$2^{-309.8}$	$2^{-194757.1}$

4 Security Analysis of LEAP

This section presents a comprehensive security analysis of the LEAP scheme. Unlike the classical PRNGs, we reduce the security of our scheme to hard problems in lattices, where the hardness of lattice problems ensures the security of LEAP. Therefore, we need only focus on the security analysis of lattices. Most state-of-the-art lattice-based schemes (e.g. Kyber[BDK+18]) also employ this method. The LEAP.GEN algorithm can be reduced to the VNTRU and RLWR problems, while the LEAP.PRNG algorithm can be reduced to the RLWR problem. Figure 3 illustrates the relationship between the LEAP.GEN and LEAP.PRNG algorithms, as well as their connections to the security assumptions. Solid arrows indicate the relationships between the algorithms, while dotted lines represent the direction of security reduction.

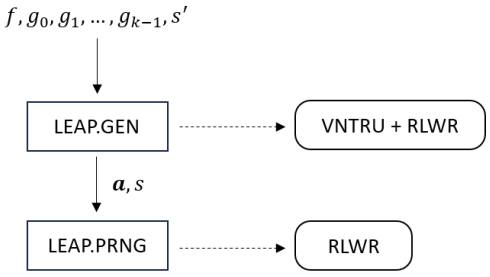

Figure 3: Relationship between algorithms and security assumptions

4.1 Security Reduction of LEAP

We begin by defining the advantage of an adversary $\mathcal{A}$ in attacking the algorithm Π, as well as the VNTRU and RLWR problems. Let $\mathbf{B}^t$ denote the set of byte arrays of length t, and let $\boldsymbol{b}^t \in \mathbf{B}^t$ represent a byte array of length t. D_s and D_g are the distributions over $\mathcal{R}$, while D_f denotes the distribution over $\mathcal{R}_q^*$. The definitions are as follows:

$$\mathbf{Adv}_{\Pi}(\mathcal{A}) = \left|\Pr\left[b = 1 : \boldsymbol{b}^t \leftarrow \Pi; b \leftarrow \mathcal{A}(\boldsymbol{b}^t)\right] - \Pr\left[b = 1 : \boldsymbol{b}^t \leftarrow \mathbf{B}^t; b \leftarrow \mathcal{A}(\boldsymbol{b}^t)\right]\right|$$

$$\mathbf{Adv}_{\mathsf{RLWR}}(\mathcal{A}) = \left|\Pr\left[b = 1 : \begin{array}{l} a \leftarrow \mathcal{R}_q; s \leftarrow D_s; \\ t = \lfloor a \cdot s \rceil_p; \\ b \leftarrow \mathcal{A}(a, t) \end{array}\right] - \Pr\left[b = 1 : \begin{array}{l} a \leftarrow \mathcal{R}_q; \\ t \leftarrow \mathcal{R}_p; \\ b \leftarrow \mathcal{A}(a, t) \end{array}\right]\right|$$

$$\mathbf{Adv}_{\mathsf{VNTRU}}(\mathcal{A}) = \left|\Pr\left[b = 1 : \begin{array}{l} f \leftarrow D_f; \boldsymbol{g} \leftarrow D_g^k; \\ \boldsymbol{t} = (g_0/f, \cdots, g_{k-1}/f); \\ b \leftarrow \mathcal{A}(\boldsymbol{t}) \end{array}\right] - \Pr\left[b = 1 : \begin{array}{l} \boldsymbol{t} \leftarrow \mathcal{R}_q^k; \\ b \leftarrow \mathcal{A}(\boldsymbol{t}) \end{array}\right]\right|$$

If $\mathbf{Adv}_{\mathsf{RLWR}}(\mathcal{A})/\mathbf{Adv}_{\mathsf{VNTRU}}(\mathcal{A})$ is negligible, it is referred to as the decisional RLWR/VNTRU assumption.

4.1.1 Security Reduction of LEAP.GEN

The security of LEAP.GEN can be reduced to the VNTRU and RLWR assumptions. Therefore, we ensure that the distribution of the generated pseudorandom numbers is indistinguishable from a uniform distribution, thereby guaranteeing that the $\boldsymbol{a}$ and s sampled from these pseudorandom numbers satisfy the requirements of LEAP.PRNG.

Theorem 1. *Let* Π_{GEN} *be the key generation algorithm* LEAP.GEN *defined by Algorithm 3. Let* δ_1 *be the probability that the algorithm* LEAP.GEN *fails to produce output due to the rejection sampling. If there is a probabilistic polynomial time (PPT) adversary* $\mathcal{A}$ *against the algorithm* Π_{GEN}*, then there exists a PPT adversary* $\mathcal{B}$ *against the decisional* VNTRU *problem with roughly the same running time as that of* $\mathcal{A}$ *or a PPT adversary* $\mathcal{C}$ *against the decisional* RLWR *problem with with roughly the same running time as that of* $\mathcal{A}$ *such that* $\mathbf{Adv}_{\Pi_{\mathsf{GEN}}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathsf{VNTRU}}(\mathcal{B}) + d^2 \cdot \mathbf{Adv}_{\mathsf{RLWR}}(\mathcal{C}) + \delta_1$*, where* $d = \mathcal{O}(mn)$ *is the number of iterations in the* RLWR*.*

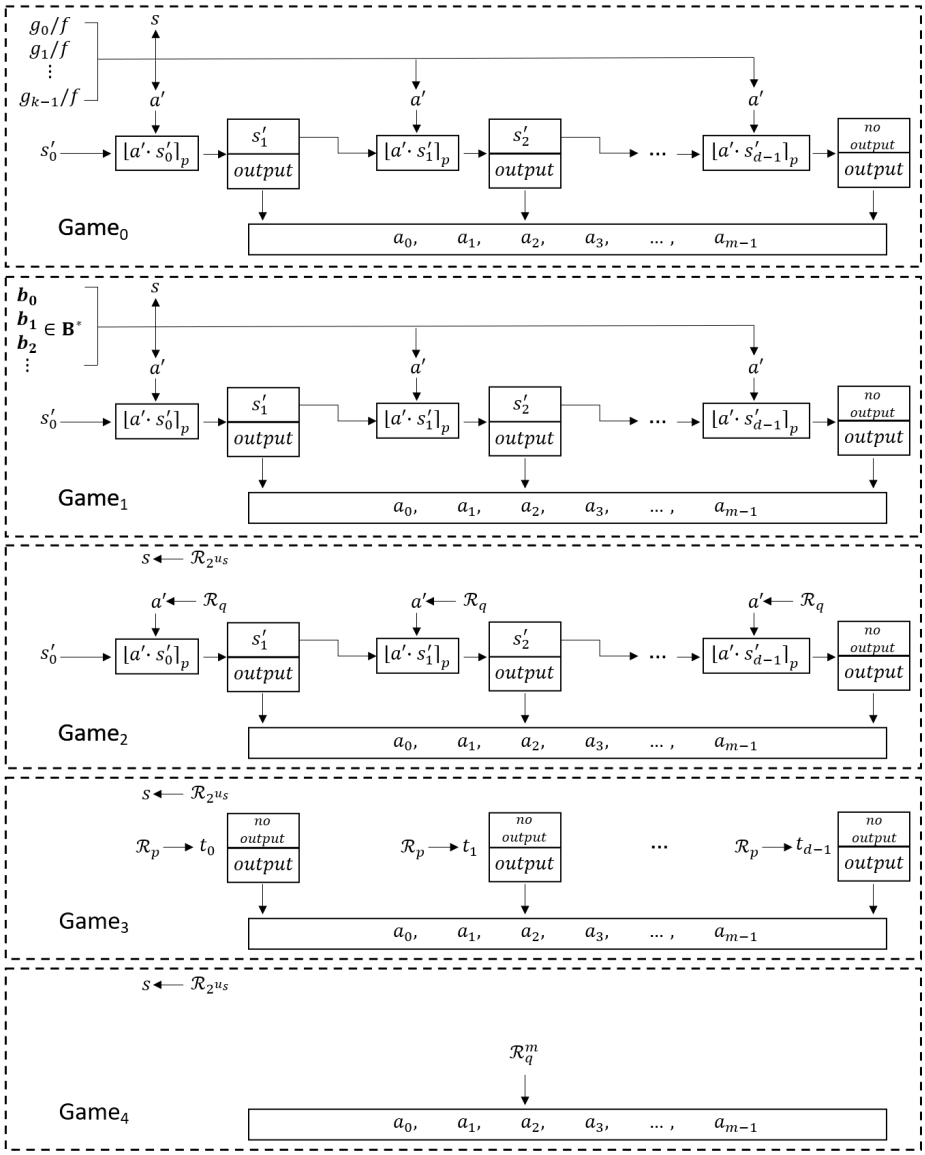

Figure 4: Games for Theorem 1

Proof. For the key generation algorithm Π_{GEN}, which outputs the key in the space $\mathcal{R}_q^m \times \mathcal{R}_{2^{u_s}}$, we define the Π_{GEN} security game as follow. The challenger begins by randomly selecting a bit $b \in \{0,1\}$. If $b = 0$, the challenger genarates the key $\boldsymbol{a} = (a_0, \ldots, a_{m-1})$[6] and s using the algorithm Π_{GEN} and sends it to the adversary $\mathcal{A}$. If $b = 1$, the challenger instead samples $(\hat{\boldsymbol{a}}, \hat{s})$ uniformly at random from $\mathcal{R}_q^m \times \mathcal{R}_{2^{u_s}}$, and sends these to $\mathcal{A}$. The adversary $\mathcal{A}$ then outputs a guess $b' \in \{0,1\}$ for the value of b. The game concludes by checking whether $b = b'$. Let $\mathbf{Adv}_{\Pi_{\mathsf{GEN}}}(\mathcal{A})$ [7] denote the advantage of $\mathcal{A}$ in distinguishing the real output of Π_{GEN} from a uniformly random one.

To prove Theorem 1, we introduce a sequence of games: Game_0, Game_1, Game_2, Game_3 and Game_4. The first game, Game_0, corresponds to the Π_{GEN} security game as defined for the scheme $\mathsf{LEAP.GEN}$, while the remaining games, Game_1 through Game_3, are defined based on the modifications illustrated in Figure 4. The security of the $\mathsf{LEAP.GEN}$ scheme is established by demonstrating that Game_0 and Game_4 are computationally indistinguishable from the perspective of the adversary $\mathcal{A}$. Let $\mathbf{Adv}_{\mathsf{Game}_i}(\mathcal{A})$ denote the advantage of $\mathcal{A}$ in distinguishing game Game_i, where $i \in \{0,1,2,3\}$. Note that due to rejection sampling, the algorithm Π_{GEN} may fail to produce the output. In the security reduction to the assumptions, it is assumed that the algorithm always produces the output. The increase in the advantage of the adversary, denoted by δ_1, resulting from this potential failure is analyzed in Lemma 1.

Game$_0$: This game is the real Π_{GEN} security game described above.

- **The item** $(\boldsymbol{a}, s)$: This pair is generated according to Game_0, as depicted in Figure 4.
- **The item** $(\hat{\boldsymbol{a}}, \hat{s})$: This pair is generated by sampling uniformly at random from $\mathcal{R}_q^m \times \mathcal{R}_{2^{u_s}}$.
- **Challenge:** The challenger randomly selects a bit $b \in \{0,1\}$. If $b = 0$, the challenger sends $(\boldsymbol{a}, s)$ to the adversary $\mathcal{A}$; otherwise, it sends $(\hat{\boldsymbol{a}}, \hat{s})$. The adversary $\mathcal{A}$ then outputs a guess b' for the value of b.

By definition, we have $\mathbf{Adv}_{\mathsf{Game}_0}(\mathcal{A}) = \mathbf{Adv}_{\Pi_{\mathsf{GEN}}}(\mathcal{A})$.

Game$_1$: This game is similar to Game_0 except that the chanllenger changes the generation of the item $(\boldsymbol{a}, s)$.

- **The item** $(\boldsymbol{a}, s)$: This pair is generated according to Game_1, as depicted in Figure 4.

The only difference between Game_1 and Game_0 lies in the generation of the pair $(\boldsymbol{a}, s)$. In Game_0, (a', s) is derived from a VNTRU instance of the form $(g_0/f, \ldots, g_{k-1}/f)$, whereas in Game_1, (a', s) is generated by uniformly sampling bytes at random. This modification may increase the advantage of adversary $\mathcal{A}$, but the increase is bounded by the advantage in solving the decisional VNTRU problem. Formally, $|\mathbf{Adv}_{\mathsf{Game}_1}(\mathcal{A}) - \mathbf{Adv}_{\mathsf{Game}_0}(\mathcal{A})| \leq \mathbf{Adv}_{\mathsf{VNTRU}}(\mathcal{B})$, where $\mathcal{B}$ is an adversary against the decisional VNTRU assumption.

Game$_2$: This game is similar to Game_1 except that the chanllenger changes the generation of the item $(\boldsymbol{a}, s)$.

- **The item** $(\boldsymbol{a}, s)$: This pair is generated according to Game_2, as depicted in Figure 4.

The only difference between Game_2 and Game_1 lies in the generation of the pair $(\boldsymbol{a}, s)$. In Game_1, (a', s) is derived from a random byte array using Algorithms 2 and 1, whereas

[6] In the actual algorithm, the adversary obtains the NTT-representation $\hat{\boldsymbol{a}}$ of $\boldsymbol{a}$. This distinction does not affect the security analysis.

[7] This security game gives a concrete definition of the advantage $\mathbf{Adv}_{\Pi}(\mathcal{A})$ in Section 4.1.

in Game_2, (a', s) is sampled uniformly at random from $\mathcal{R}_q \times \mathcal{R}_{2^{u_s}}$. Since Algorithms 2 and 1 effectively transform a uniform byte array to a uniform distribution over the corresponding ring, the output remains uniformly distributed if the input is uniformly random. Consequently, the view of adversary in Game_1 and Game_2 is identical. Therefore, we have $\mathbf{Adv}_{\mathsf{Game}_2}(\mathcal{A}) = \mathbf{Adv}_{\mathsf{Game}_1}(\mathcal{A})$.

Game$_3$: This game is similar to Game_2 except that the chanllenger changes the generation of the item $(\boldsymbol{a}, s)$.

- **The item** $(\boldsymbol{a}, s)$: This pair is generated according to Game_3, as depicted in Figure 4.

The only difference between Game_3 and Game_2 is that, in Game_3, each term $\lfloor a' \cdot s'_i \rceil_p$ is replaced by t_i, where $i \in \{0, \ldots, d-1\}$ and each t_i is sampled uniformly at random from $\mathcal{R}_p$. The difference in the advantage of adversary $\mathcal{A}$ is solely due to this modification. Let $\mathcal{A}'$ be an adversary tasked with distinguishing $(\lfloor a' \cdot s'_0 \rceil_p, \ldots, \lfloor a' \cdot s'_{d-1} \rceil_p)$ from $(t_0, \ldots, t_{d-1})$. We now proceed to analyze the advantage of $\mathcal{A}'$ in this distinguishing task. If the adversary $\mathcal{A}'$ can distinguish between the $(\lfloor a' \cdot s'_0 \rceil_p, \ldots, \lfloor a' \cdot s'_{d-1} \rceil_p)$ and $(t_0, \ldots, t_{d-1})$, then, by the definition of the games and the analysis above, the distinguishing advantage is given by $|\mathbf{Adv}_{\mathsf{Game}_3}(\mathcal{A}) - \mathbf{Adv}_{\mathsf{Game}_2}(\mathcal{A})|$. According to the standard hybrid argument, there must exist an adversary $\mathcal{A}''$ that can distinguish $\{t_0, \ldots, t_{i-1}, \lfloor a' \cdot s'_i \rceil_p, \lfloor a' \cdot s'_{i+1} \rceil_p, \ldots, \lfloor a' \cdot s'_{d-1} \rceil_p\}$ from $\{t_0, \ldots, t_{i-1}, t_i, \lfloor a' \cdot s'_{i+1} \rceil_p, \ldots, \lfloor a' \cdot s'_{d-1} \rceil_p\}$ with the advantage at least $1/d \cdot |\mathbf{Adv}_{\mathsf{Game}_3}(\mathcal{A}) - \mathbf{Adv}_{\mathsf{Game}_2}(\mathcal{A})|$, where $0 \le i < d$. We claim that $\mathbf{Adv}(\mathcal{A}'') \le \mathbf{Adv}_{\mathsf{RLWR}}(\mathcal{C})$, and we prove this by constructing an adversary $\mathcal{C}$ against the RLWR assumption using $\mathcal{A}''$.

Given a challenge (a, h) from the RLWR challenger, where (a, h) is either of the form $(a, \lfloor a \cdot s \rceil_p)$ or is sampled uniformly at random from $\mathcal{R}_q \times \mathcal{R}_p$, we proceed as follows: We use h in place of the j-th term $\lfloor a' \cdot s'_i \rceil_p$. The remaining values $\lfloor a' \cdot s'_{j+1} \rceil_p, \ldots, \lfloor a' \cdot s'_{d-1} \rceil_p$ are computed as in Game_2, and $t_0, \ldots, t_{j-1}$ are sampled as in Game_3. If (a, h) corresponds to an RLWR instance of the form $(a, \lfloor a \cdot s \rceil_p)$, then the result is $\{t_0, \ldots, t_{j-1}, \lfloor a' \cdot s'_i \rceil_p, \lfloor a' \cdot s'_{j+1} \rceil_p, \ldots, \lfloor a' \cdot s'_{d-1} \rceil_p\}$. Otherwise, if (a, h) is uniformly random, the result is $\{t_0, \ldots, t_{j-1}, t_j, \lfloor a' \cdot s'_{j+1} \rceil_p, \ldots, \lfloor a' \cdot s'_{d-1} \rceil_p\}$. This result is then given to $\mathcal{A}''$, whose output is used by $\mathcal{C}$ to decide the RLWR challenge. If the adversary $\mathcal{A}''$ succeeds, then $\mathcal{C}$ successfully breaks the decisional RLWR assumption. Since the correct value of i is unknown to the adversary $\mathcal{C}$, it must guess the j uniformly at random from the set $\{0, \ldots, d-1\}$. The guess is correct, i.e., $j = i$, with probability $1/d$.

We conclude that $|\mathbf{Adv}_{\mathsf{Game}_3}(\mathcal{A}) - \mathbf{Adv}_{\mathsf{Game}_2}(\mathcal{A})| \le d^2 \cdot \mathbf{Adv}_{\mathsf{RLWR}}(\mathcal{C})$ based on the analysis presented above.

Game$_4$: This game is similar to Game_3 except that the chanllenger changes the generation of the item $(\boldsymbol{a}, s)$.

- **The item** $(\boldsymbol{a}, s)$: This pair is generated according to Game_4, as depicted in Figure 4.

The only difference between Game_4 and Game_3 lies in the generation of the pair $(\boldsymbol{a}, s)$. In Game_4, $(\boldsymbol{a}, s)$ is sampled uniformly at random from $\mathcal{R}_q^m \times \mathcal{R}_{2^{u_s}}$, whereas in Game_3, it is indirectly derived from the values t_i, where $i \in \{0, \ldots, d-1\}$ and each t_i is sampled uniformly from $\mathcal{R}_p$. The transformation from $t_0, \cdots, t_{d-1}$ to the $a_0, \cdots, a_{m-1}$ is performed using the same technique as in Algorithm 2. Consequently, the distribution of $\boldsymbol{a}$ remains uniform in both Game_3 and Game_4. Since the distribution of s is also unchanged, it follows that the view of the adversary is identical in both games. Therefore, $\mathbf{Adv}_{\mathsf{Game}_4}(\mathcal{A}) = \mathbf{Adv}_{\mathsf{Game}_3}(\mathcal{A})$. Moreover, in Game_4, the pair $(\boldsymbol{a}, s)$ is sampled uniformly at random from $\mathcal{R}_q^m \times \mathcal{R}_{2^{u_s}}$. Hence, we conclude that $\mathbf{Adv}_{\mathsf{Game}_4}(\mathcal{A}) = 0$.

Lemma 1. *Define a function as* $\mathcal{P}(n_0, n_1, q_0, q_1) := \sum_{j=n_0-n_1}^{n_0} \binom{n_0}{j} \left(1 - \frac{q_1}{q_0}\right)^j \left(\frac{q_1}{q_0}\right)^{n_0-j}$. *The probability that the algorithm* LEAP.GEN *(Algorithm 3) fails to produce output due to*

the rejection sampling is $\delta_1 \leq \sum_{i=0}^{kN} \left(\delta_{1,i}^{\mathsf{VNTRU}} \cdot \delta_{1,i}^{a'}\right) + \left(1-(1-\delta_1^{s'})^{mn}\right) + (1-(1-\delta_1^{a})^{mn})$, *where*

$$\delta_{1,i}^{\mathsf{VNTRU}} = \mathcal{P}(kN, i, Q, Q'), \quad \delta_1^{\mathsf{s}'} = \mathcal{P}(n, \frac{u_{s'}n}{\log_2(q')}, q, q')$$

$$\delta_{1,i}^{\mathsf{a}'} = \mathcal{P}(N_i, n, 2^{\lceil \log_2(q) \rceil}, q), \quad N_i = \frac{i\log_2(Q') - u_s n}{\log_2(q')}$$

$$\delta_1^a = \sum_{i=0}^{n} \left(\mathcal{P}(n, i, q, q') \cdot \mathcal{P}(\frac{i\log_2(q') - u_{s'}n}{\lceil \log_2(q) \rceil}, 1, 2^{\lceil \log_2(q) \rceil}, q)\right)$$

Proof. The probability δ_1 comprises three components: the rejection sampling from Q to Q' in VNTRU, the rejection sampling from $2^{\lceil \log_2(q) \rceil}$ to q in the generation of a', and the rejection sampling from q to q' in RLWR. We calculate the probability that the VNTRU component generates i elements over $\mathbb{Z}_{Q'}$ as $\delta_{1,i}^{\mathsf{VNTRU}}$. Since we require $u_s n$ bits for generating s, the remaining elements are used to generate a'. We calculate the probability $\delta_{1,i}^{\mathsf{a}'}$ that fails to generate the a' with $\frac{i\log_2(Q')-u_s n}{\log_2(q')}$ elements over $\mathbb{Z}_{q'}$ from the VNTRU component. Then we calculate the probability that RLWR part fails to update s' in each iteration as $\delta_1^{\mathsf{s}'}$.

For a conservative bound, let δ_1^a denote the probability that each iteration fails to generate one coefficient a_i. Since the RLWR part requires for mn iterations to generate the complete key in this case, we compute the complete failure probability as:

$$\delta_1 \leq \sum_{i=0}^{kN} \left(\delta_{1,i}^{\mathsf{VNTRU}} \cdot \delta_{1,i}^{a'}\right) + \left(1-(1-\delta_1^{s'})^{mn}\right) + (1-(1-\delta_1^{a})^{mn})$$

Our parameter sets ensure that δ_1 is negligible. The concrete values are presented in Table 3. For clarity, we can view the first term of δ_1 as the probability that LEAP.GEN cannot start, the second term is the probability that LEAP.GEN cannot run mn iterations consecutively, and the third term is the probability that LEAP.GEN cannot generate the complete key after running mn iterations. □

Based on the analysis above, we obtain the following bound:

$$\mathbf{Adv}_{\Pi_{\mathsf{GEN}}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathsf{VNTRU}}(\mathcal{B}) + d^2 \cdot \mathbf{Adv}_{\mathsf{RLWR}}(\mathcal{C}) + \delta_1.$$

□

4.1.2 Security Reduction of LEAP.PRNG

The security of LEAP.PRNG can be reduced to the RLWR assumption. Through the reduction, we ensure that the distribution of the generated pseudorandom numbers is indistinguishable from the uniform distribution, thereby demonstrating that LEAP satisfies the security definition of PRNGs.

Theorem 2. *Let* Π_{PRNG} *be the pseudorandom number generator scheme* LEAP.PRNG *defined by Algorithm 4. Let* δ_2 *be the probability that* LEAP.PRNG *fails to produce output due to the rejection sampling. Suppose that there exists a probabilistic polynomial time (PPT) adversary* $\mathcal{A}$ *against the algorithm* Π_{PRNG}*. In that case, there exists a PPT adversary* $\mathcal{B}$ *against the decisional* RLWR *problem with roughly the same running time as that of* $\mathcal{A}$ *such that* $\mathbf{Adv}_{\Pi_{\mathsf{PRNG}}}(\mathcal{A}) \leq d^2 \cdot \mathbf{Adv}_{\mathsf{RLWR}}(\mathcal{B}) + \delta_2$*, where* d *is the number of iterations in the scheme.*

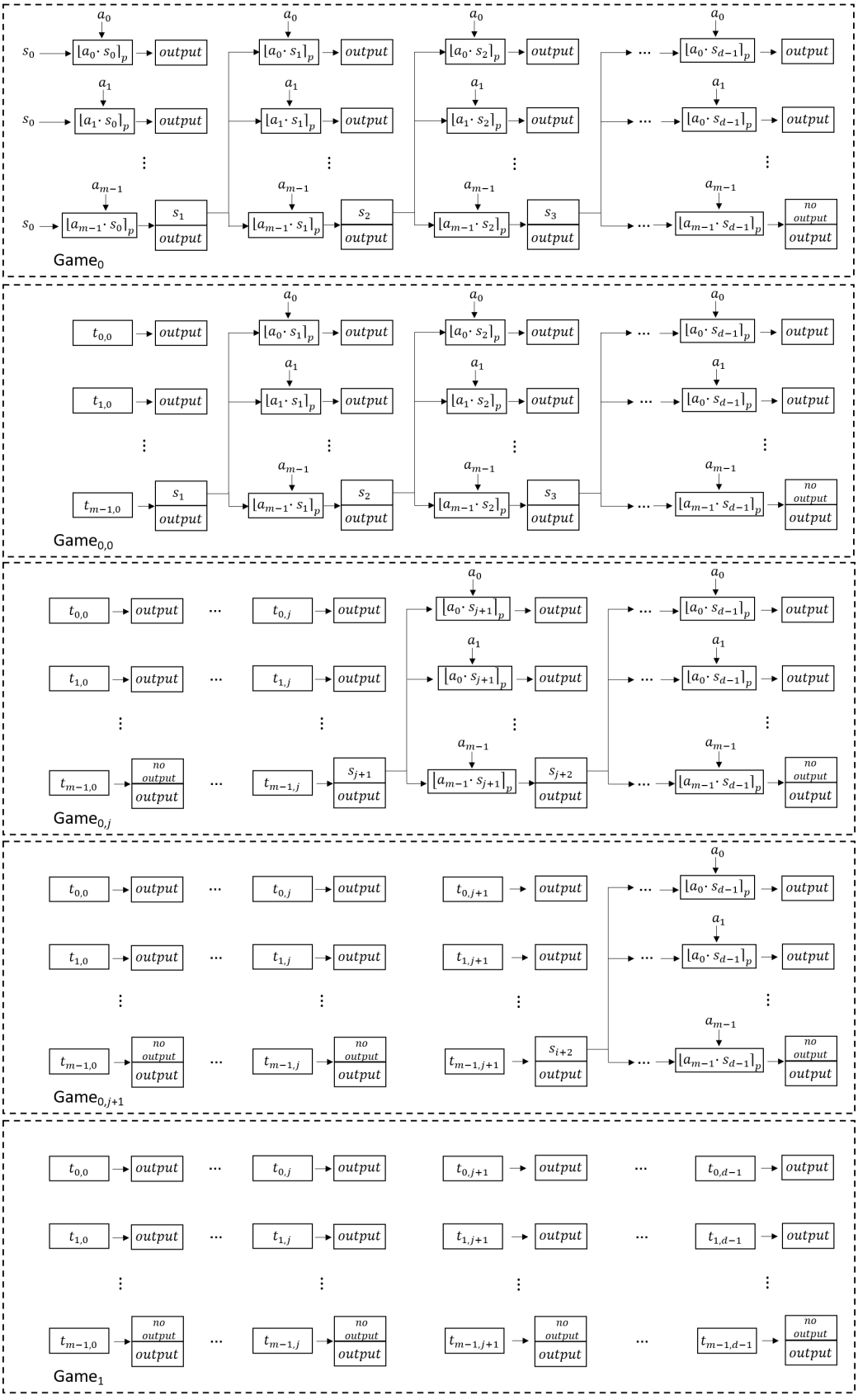

Figure 5: Games for Theorem 2

Proof. For the pseudorandom number generator algorithm Π_{PRNG}, where the output $\boldsymbol{b}^l \in \mathbf{B}^l$, we define the Π_{PRNG} security game as follows. The challenger begins by randomly selecting a bit $b \in \{0,1\}$. If $b = 0$, the challenger generates $\boldsymbol{b}^l$ using the algorithm Π_{PRNG} and sends it to the adversary $\mathcal{A}$. If $b = 1$, the challenger instead samples $\hat{\boldsymbol{b}}^l$ uniformly at random from $\mathbf{B}^l$ and sends it to $\mathcal{A}$. The adversary $\mathcal{A}$ then outputs a guess $b' \in \{0,1\}$ for the value of b. The game concludes by checking whether $b = b'$. Let $\mathbf{Adv}_{\Pi_{\mathsf{PRNG}}}(\mathcal{A})$ denote the advantage of $\mathcal{A}$ in distinguishing the real output of Π_{PRNG} from a uniformly random one.

To prove Theorem 2, we introduce two games: Game_0 and Game_1. The game Game_0 corresponds to the Π_{PRNG} security game as defined for the LEAP.PRNG. In contrast, Game_1 is constructed by replacing each term $\lfloor a_i \cdot s_j \rceil_p$ with a randomly selected value $t_{i,j} \in \mathcal{R}_p$, where $(i,j) \in \{0,\ldots,m-1\} \times \{0,\ldots,d-1\}$. The security of the LEAP.GEN scheme is established by demonstrating that Game_0 and Game_1 are computationally indistinguishable from the perspective of the adversary $\mathcal{A}$. Let $\mathbf{Adv}_{\mathsf{Game}_i}(\mathcal{A})$ denote the advantage of $\mathcal{A}$ in distinguishing game Game_i, where $i \in \{0,1\}$. Note that due to rejection sampling, the algorithm Π_{PRNG} may fail to produce the output. In the security reduction to the assumption, it is assumed that the scheme always produces the output. The increase in the advantage of adversary, denoted by δ_2, resulting from this potential failure is analyzed in Lemma 2.

Game$_0$: This game is the real Π_{PRNG} security game described above.

- **The item $\boldsymbol{b}^l$**: This array is generated according to Game_0, as depicted in Figure 5.
- **The item $\hat{\boldsymbol{b}}^l$**: The array $\hat{\boldsymbol{b}}^l$ is generated by sampling uniformly at random from the set $\mathbf{B}^l$.
- **Challenge:** The challenger randomly selects $b \in \{0,1\}$. If $b = 0$, the challenger sends $\boldsymbol{b}^l$ to $\mathcal{A}$; otherwise, it sends $\hat{\boldsymbol{b}}^l$. The adversary $\mathcal{A}$ then outputs a guess b' for the value of b.

By definition, we have $\mathbf{Adv}_{\mathsf{Game}_0}(\mathcal{A}) = \mathbf{Adv}_{\mathsf{LEAP.PRNG}}(\mathcal{A})$

Game$_1$: This game is similar to Game_0 except that the chanllenger changes the generation of the item $\boldsymbol{b}^l$.

- **The item $\boldsymbol{b}^l$**: This array is generated according to Game_1, as depicted in Figure 5.

Let $\mathbf{Adv}^{distin}_{0\text{-}1}(\mathcal{A})$ denote the distinguish advantage of the adversary $\mathcal{A}$ between Game_0 and Game_1. Then we have $|\mathbf{Adv}_{\mathsf{Game}_1}(\mathcal{A}) - \mathbf{Adv}_{\mathsf{Game}_0}(\mathcal{A})| \leq \mathbf{Adv}^{distin}_{0\text{-}1}(\mathcal{A})$. By definition, the output of Game_1 is perfectly uniform. Therefore, $\mathbf{Adv}_{\mathsf{Game}_1}(\mathcal{A}) = 0$, which implies $\mathbf{Adv}_{\mathsf{Game}_0}(\mathcal{A}) \leq \mathbf{Adv}^{distin}_{0\text{-}1}(\mathcal{A})$. Consequently, we have $\mathbf{Adv}_{\mathsf{LEAP.PRNG}}(\mathcal{A}) \leq \mathbf{Adv}^{distin}_{0\text{-}1}(\mathcal{A})$. We now proceed to analyze $\mathbf{Adv}^{\mathsf{distin}}_{0\text{-}1}(\mathcal{A})$. To do this, we introduce a sequence of intermediate games $\mathsf{Game}_{0,0}$ to $\mathsf{Game}_{0,d-1}$ (as illustrated in Figure 5). In $\mathsf{Game}_{0,j}$, the 0 to j columns are generated randomly as in Game_1, while the remaining $d-j-1$ columns are generated as in Game_0. The only difference between $\mathsf{Game}_{0,j}$ and $\mathsf{Game}_{0,j+1}$ is the $(j+1)$-th column. If the adversary $\mathcal{A}$ can distinguish between Game_0 and Game_1 with advantage $\mathbf{Adv}^{\mathsf{distin}}_{0\text{-}1}(\mathcal{A})$, then by a standard hybrid argument, there must exist an adversary $\mathcal{A}''$ can distinguish the $\mathsf{Game}_{0,j}$ and $\mathsf{Game}_{0,j+1}$ with the advantage $\mathbf{Adv}^{distin}_{(0,j)\text{-}(0,j+1)}(\mathcal{A}'') \geq 1/d \cdot \mathbf{Adv}^{\mathsf{distin}}_{0\text{-}1}(\mathcal{A})$, where $0 \leq j < d-1$. We claim that $\mathbf{Adv}^{distin}_{(0,j)\text{-}(0,j+1)}(\mathcal{A}'') \leq \mathbf{Adv}_{\mathsf{RLWR}}(\mathcal{B})$, and we prove this by constructing an adversary $\mathcal{B}$ against the RLWR assumption using $\mathcal{A}''$.

Given a challenge $(a_0,h_0),\ldots,(a_{m-1},h_{m-1})$ from the RLWR challenger, where each pair is either an RLWR instance of the form $(a_i, \lfloor a_i \cdot s \rceil_p)$ or a uniformly random pair (a_i,t_i) sampled from $\mathcal{R}_q \times \mathcal{R}_p$, we proceed as follows: We replace the $(w+1)$-th column $(\lfloor a_0 \cdot s_{w+1} \rceil_p,\ldots,\lfloor a_{m-1} \cdot s_{w+1} \rceil_p)$ in Game_0 with $(h_0,\ldots,h_{m-1})$. Since we have $(a_0,\ldots,a_{m-1})$,

we can generate $(\lfloor a_0 \cdot s_{w+2} \rceil_p, \ldots, \lfloor a_{m-1} \cdot s_{w+2} \rceil_p)$ through $(\lfloor a_0 \cdot s_{d-1} \rceil_p, \ldots, \lfloor a_{m-1} \cdot s_{d-1} \rceil_p)$ as specified in LEAP.PRNG. The columns from 0 to w are generated using the method from Game_1 by replacing all terms $\lfloor a_i \cdot s_j \rceil_p$ with $t_{i,j}$, where $(i,j) \in \{0,\ldots,m-1\} \times \{0,\ldots,d-1\}$. If the given $(h_0,\ldots,h_{m-1})$ is RLWR instances, then the resulting output corresponds to $\mathsf{Game}_{0,j}$; otherwise, it corresponds to $\mathsf{Game}_{0,j+1}$. We then send this output to $\mathcal{A}''$ and observe its response. If $\mathcal{A}''$ succeeds in distinguishing the output, then the adversary $\mathcal{B}$ successfully breaks the RLWR assumption. Since the correct value of j is unknown to the adversary $\mathcal{B}$, it must guess the w uniformly at random from the set $\{0,\ldots,d-1\}$. The guess is correct, i.e., $w = j$, with probability $1/d$. Therefore, the overall distinguishing advantage satisfies: $\mathbf{Adv}^{distin}_{0\text{-}1}(\mathcal{A}) \leq d^2 \cdot \mathbf{Adv}_{\mathsf{RLWR}}(\mathcal{B})$.

Lemma 2. *Define a function as* $\mathcal{P}(n_0,n_1,q_0,q_1) := \sum_{j=n_0-n_1}^{n_0} \binom{n_0}{j} \left(1-\frac{q_1}{q_0}\right)^j \left(\frac{q_1}{q_0}\right)^{n_0-j}$. *The probability that the algorithm* LEAP.PRNG *(Algorithm 4) fails to produce output due to the rejection sampling is* $\delta_2 \leq \left(1-(1-\delta_2^s)^l\right) + \left(1-(1-\delta_2^{byte})^l\right)$, *where* l *is the upper bound of the number of output bytes, and*

$$\delta_2^s = \mathcal{P}(mn, \frac{u_s n}{\log_2(q')}, q, q'), \quad \delta_2^{byte} = \mathcal{P}(mn, \frac{u_s n}{\log_2(q')} + \frac{8}{\log_2(q')}, q, q')$$

Proof. For LEAP.PRNG, we require $u_s n$ bits to update s in each iteration. We calculate the corresponding probability as δ_2^s. Then, we calculate the probability that each iteration generates at least one byte as δ_2^{byte}. Therefore, we compute the complete failure probability as:

$$\delta_2 \leq \left(1-(1-\delta_2^s)^l\right) + \left(1-(1-\delta_2^{byte})^l\right)$$

We set the upper bound of the number of output bytes as $l = 2^{256}$, which is further enough for the usage, and our parameter sets ensure that δ_2 is negligible for this l. The concrete values are presented in Table 3. □

Based on the analysis above, we obtain the following bound:

$$\mathbf{Adv}_{\Pi_{\mathsf{PRNG}}}(\mathcal{A}) \leq d^2 \mathbf{Adv}_{\mathsf{RLWR}}(\mathcal{B}) + \delta_2$$

. □

4.2 Concrete Security of LEAP

We estimate the concrete security of LEAP using state-of-the-art estimation methods and cost models. LEAP is divided into two components: LEAP.GEN and LEAP.PRNG. The security levels of these two components are based on VNTRU and RLWR with hidden $\boldsymbol{a}$. We first analyze the concrete security of LWE, then transform the concrete security of these two instances into different LWE instances, estimating the associated costs separately.

4.2.1 Concrete Security of LWE

The concrete security of LWE has been extensively studied recently. We can broadly categorize the attacks into three types: algebraic, lattice-based, and combination. Additionally, some attacks incorporate elements from more than one category. The following sections will demonstrate the standard analysis method for LWE. For convenience, we briefly explain the form of LWE: $\mathbf{As} + \mathbf{e} \equiv \mathbf{b} \mod q$, where $\mathbf{A} \in \mathbb{Z}^{m \times n}$, $\mathbf{s} \in \mathbb{Z}^n$, $\mathbf{e} \in \mathbb{Z}^m$, and $\mathbf{s}[i] \leftarrow D_s$, $\mathbf{e}[i] \leftarrow D_e$. The standard deviations of the distributions D_s and D_e are σ_s and σ_e, respectively. We denote this instance as $\mathsf{LWE}_{m,n,q,\sigma_s,\sigma_e}$.

Algebraic Attacks Algebraic attacks were first proposed in [AG11] and have been well studied in [ACF+15a]. The idea is to construct equations over the variables $\mathbf{s}[i]$ based on the small size of the set of $\mathbf{e}[i]$. For example, from the equation $< \mathbf{A}[i], \mathbf{s} > + \mathbf{e}[i] \equiv \mathbf{b}[i] \mod q$, if the set of possible $\mathbf{e}[i]$ is $\{-E, \ldots, 0, \ldots, E\}$, we can derive an equation $\prod_{j=-E}^{E}(< \mathbf{A}[i], \mathbf{s} > + j - \mathbf{b}[i])$ on $\mathbb{Z}_q$. Ultimately, we obtain m equations of degree $2E+1$ over n variables and then use a Gröbner basis algorithm to find the roots. The time complexity of the Gröbner basis algorithm depends on the set of possible $\mathbf{e}[i]$ values, as well as the parameters m and n. Consequently, this attack is not optimal unless m is large and the error distribution is extremely sparse.

Lattice Attacks Lattice attacks are the mainstream method for estimating the concrete security of lattice-based schemes today. They encompass two approaches: the primal attack and the dual attack. Both methods are based on lattice reduction algorithms.

The primal attack constructs a lattice for LWE as follows:

$$\mathcal{L}^{\mathbf{A},q} = \begin{bmatrix} \mathbf{A}^T & \mathbf{I}_n \\ q\mathbf{I}_m & \mathbf{0} \end{bmatrix}$$

The parameter settings of an LWE instance ensure that the closest vector in $\mathcal{L}^{A,q}$ to $(\mathbf{b}, 0)$ is $(\mathbf{s}\mathbf{A}^T, \mathbf{s})$ with high probability. The distance between these vectors is $||(-\mathbf{e}, \mathbf{s})||$, which is relatively small. Therefore, once we solve the above Closest Vector Problem, we solve the LWE. We transform the Closest Vector Problem to the Shortest Vector Problem using the embedding technique ([Kan87], [BG14]):

$$\mathcal{L}^{\mathbf{A},\mathbf{b},q} = \begin{bmatrix} \mathbf{A}^T & \mathbf{I}_n & \mathbf{0} \\ q\mathbf{I}_m & \mathbf{0} & \mathbf{0} \\ \mathbf{b} & \mathbf{0} & 1 \end{bmatrix}$$

We apply a lattice reduction algorithm to the lattice $\mathcal{L}^{\mathbf{A},\mathbf{b},q}$ to obtain the shortest vector $(-\mathbf{e}, \mathbf{s}, -1)$. The complexity of solving LWE is equivalent to the cost of the lattice reduction algorithm. Currently, the state-of-the-art approach employs the BKZ algorithm [SE91], and specifically, we use the lattice-estimator script[8] from [APS15] to determine the security level.

The dual attack first rewrites $\mathbf{A}\mathbf{s} + \mathbf{e} \equiv \mathbf{b} \mod q$ to $\mathbf{A}_0\mathbf{s}_0 + \mathbf{A}_1\mathbf{s}_1 + \mathbf{e} \equiv \mathbf{b} \mod q$, where the dimensions of $\mathbf{A}_0$ and $\mathbf{A}_1$ are n_0 and n_1 respectively, with $n_0 + n_1 = n$. Then, we build a lattice $\mathcal{L}^{\mathbf{A}_0,q} = \{(\mathbf{w}, \mathbf{v}) \mid \mathbf{w} \equiv \mathbf{v}\mathbf{A}_0 \mod q\}$ as follows:

$$\mathcal{L}^{\mathbf{A}_0,q} = \begin{bmatrix} \mathbf{A}_0 & \mathbf{I}_m \\ q\mathbf{I}_{n_0} & \mathbf{0} \end{bmatrix}$$

After applying a lattice reduction algorithm to the above lattice, we obtain many short vectors $(\mathbf{w}_i, \mathbf{v}_i)$. This leads to the following relation:

$$\mathbf{v}_i\mathbf{b} - \mathbf{v}_i\mathbf{A}_1\mathbf{s}_1 \equiv \mathbf{w}_i\mathbf{s}_0 + \mathbf{v}_i\mathbf{e} \mod q$$

Thus, we can transform the LWE-solving problem into a distinguishing problem. First, we fix a guess for $\mathbf{s}_1$. If the guess is correct, we obtain $\mathbf{w}_i\mathbf{s}_0 + \mathbf{v}_i\mathbf{e}$; otherwise, we get an element from the uniform distribution over $\mathbb{Z}_q$. Therefore, if we can distinguish the distribution of $\mathbf{w}_i\mathbf{s}_0 + \mathbf{v}_i\mathbf{e}$ from the uniform distribution, the correct $\mathbf{s}_1$ can be identified by traversing all possible values. A distinguisher can be constructed to differentiate between the distribution of $\mathbf{w}_i\mathbf{s}_0 + \mathbf{v}_i\mathbf{e}$ and the uniform distribution over $\mathbb{Z}_q$. By traversing all possible $\mathbf{s}_1$ values, we eventually find the correct one that passes the distinguisher, thus solving LWE. The time complexity of the dual attack mainly involves lattice reduction

[8] https://github.com/malb/lattice-estimator

and traversal. We can estimate this complexity using the lattice estimator mentioned previously.

Hybrid Attacks Due to some special distributions chosen in practical schemes, such as distributions with fixed Hamming weight being super sparse, hybrid attacks combine lattice attacks and the Meet-in-the-Middle (MITM) strategy. At a high level, hybrid attacks guess some variables using MITM, thereby reducing the dimension of the lattice to be reduced. There is a balance between MITM and the lattice reduction algorithm; sometimes hybrid attacks are more effective.

Combination Attacks The combination attack commonly considered is BKW [BKW00], which is similar to the dual attack but replaces the lattice reduction step with combinatorial computation. BKW tends to perform better for specific parameter settings, and its complexity can be estimated using the method described in [ACF+15b], which has been incorporated into the lattice estimator.

4.2.2 Concrete Security of VNTRU

For a VNTRU instance formed as $f \cdot (h_0, \ldots, h_{k-1}) \equiv (g_0, \ldots, g_{k-1}) \pmod q$, where $f[i] \leftarrow D_f$, $g[i] \leftarrow D_g$, and the standard deviations of the distributions D_f and D_g are σ_f and σ_g, respectively. From an adversarial perspective, VNTRU can be viewed as a special case of LWE, where $h_i f - g_i \equiv 0 \pmod q$ is equivalent to $as + e \equiv b \pmod q$, except that $b = 0$. When considering a lattice attack on VNTRU, we transform each polynomial into an anti-circular matrix corresponding to ring multiplication and construct a lattice as:

$$\mathcal{L}^{h_0,\ldots,h_{k-1},q} = \begin{bmatrix} \mathbf{Mat}(h_0) & \cdots & \mathbf{Mat}(h_{k-1}) & \mathbf{I}_n \\ q\mathbf{I}_n & \cdots & \mathbf{0} & \mathbf{0} \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \cdots & q\mathbf{I}_n & \mathbf{0} \end{bmatrix}$$

Similar to LWE, we aim to perform lattice reduction to obtain the short vector $(\mathbf{Vec}(g_0), \ldots, \mathbf{Vec}(g_{k-1}), \mathbf{Vec}(f))$. Research indicates that VNTRU can be solved more easily than LWE with the same parameter scale under a large modulus q. [DvW21] identified the reason behind this as the dense sublattice $(\mathbf{Mat}(g_0), \ldots, \mathbf{Mat}(g_{k-1}), \mathbf{Mat}(f))$. In the case of LWE, there is only one relatively short vector in the lattice, but in VNTRU, there are n linearly independent relatively short vectors caused by the anti-circular matrix. However, [DvW21] provided a refined bound analysis, stating that $q \approx 0.004 \cdot n^{2.484}$. The time complexity for VNTRU instances with q values below this bound would differ when treated as LWE instances. The parameter settings in our scheme are not affected by this overstretched attack, so our estimation of the concrete security of VNTRU aligns with that of LWE in this context. The concrete security of VNTRU part in LEAP.GEN is estimated as the $\mathsf{LWE}_{kn,n,q,\sigma_f,\sigma_g}$ instance.

4.2.3 Concrete Security of RLWR with Hidden a

For a RLWR instance with hidden $\boldsymbol{a}$, we can write it as a form of matrix-like equation system:

$$\begin{matrix} \lfloor a_0 \cdot s_0 \rceil_p = b_{0,0} & \cdots & \lfloor a_0 \cdot s_{k-1} \rceil_p = b_{0,t-1} \\ \vdots & \ddots & \vdots \\ \lfloor a_{m-1} \cdot s_0 \rceil_p = b_{m-1,0} & \cdots & \lfloor a_{m-1} \cdot s_{k-1} \rceil_p = b_{m-1,t-1} \end{matrix}$$

a_i, s_j are reused in rows or columns, but we only know $b_{i,j}$, thus we have the transformation:

$$\begin{cases} a_i \cdot s_{j_1} + e_{i,j_1} \equiv b'_{i,j_1} \mod q \\ a_i \cdot s_{j_2} + e_{i,j_2} \equiv b'_{i,j_2} \mod q \end{cases}$$

where $b'_{i,j} = \frac{q}{p} \cdot b_{i,j}$, we get the following equation to erase a_i:

$$b'_{i,j_1} s_{j_2} - b'_{i,j_2} s_{j_1} \equiv e_{i,j_1} s_{j_2} - e_{i,j_2} s_{j_1} (\mathrm{mod} q)$$

we can build a lattice from it:

$$\mathcal{L}^{j_1,j_2,q} = \begin{bmatrix} \mathbf{Mat}(b'_{0,j_1}) & \cdots & \mathbf{Mat}(b'_{m-1,j_1}) & \mathbf{I}_n & \mathbf{0} \\ \mathbf{Mat}(b'_{0,j_2}) & \cdots & \mathbf{Mat}(b'_{m-1,j_2}) & \mathbf{0} & \mathbf{I}_n \\ q\mathbf{I}_n & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & \cdots & q\mathbf{I}_n & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

where $(\mathbf{Vec}(e_{0,j_1} s_{j_2} - e_{0,j_2} s_{j_1}), \ldots, \mathbf{Vec}(e_{m-1,j_1} s_{j_2} - e_{m-1,j_2} s_{j_1}), \mathbf{Vec}(s_{j_2}), \mathbf{Vec}(-s_{j_1}))$ is a short vector over this lattice, therefore, the lattice attack on RLWR with hidden $\boldsymbol{a}$ is equal to an attack on $\mathsf{LWE}_{mn,2n,q,\sigma_s,\sigma_s\sigma_e\sqrt{2n}}$.

4.2.4 Concrete Security of LEAP.GEN and LEAP.PRNG

After analyzing VNTRU and RLWR with hidden $\boldsymbol{a}$, we can clearly present the concrete security levels of LEAP.GEN and LEAP.PRNG, as evaluated by the lattice estimator. Since algebraic attacks do not benefit from quantum acceleration, we provide separate complexities for classical, quantum, and algebraic cases. The term 'block' refers to the β parameter for the BKZ algorithm, while 'cost' represents the logarithm of the time complexity. This value reflects the most optimal attack, as determined using the lattice estimator with our parameters. The notation '-' indicates that the corresponding indicator is meaningless under the given parameter setting. Specifically, for 'Algebraic', it means that the complexity is too large to estimate, and for 'block,' it signifies that the most optimal attack is the pure combination attack BKW without lattice reduction.

The concrete security of LEAP.GEN is presented in Table 4, where we separate the VNTRU and RLWR components for clarity. Table 5 presents the concrete security of LEAP.PRNG, with the meaning of the indicators being the same as in Table 4.

Table 4: Concrete Security of LEAP.GEN

Scheme	NTRU					RLWR				
	Classical		Quantum		Algebraic	Classical		Quantum		Algebraic
	block	cost	block	cost		block	cost	block	cost	
LEAP-128	414	146.9	419	141.8	348.5	366	143.1	373	139.2	-
LEAP-192	610	220.4	610	205.7	-	641	206.5	641	194.4	-
LEAP-256	955	287.9	955	266.4	-	933	281.7	933	260.8	-

Table 5: Concrete Security of LEAP.PRNG

Scheme	Classical				Quantum				Algebraic
	hybrid primal		hybrid dual		hybrid primal		hybrid dual		
	block	cost	block	cost	block	cost	block	cost	
LEAP-128	432	150.6	365	142.3	432	143.7	370	138.3	-
LEAP-192	779	246.7	817	258.5	779	230.2	818	241.3	-
LEAP-256	-	340.9	991	427.7	-	340.9	999	402.2	-

4.3 Randomness Test

NIST SP 800-22 [BRS+10] is a commonly used suite for testing the randomness of pseudorandom number generators. We conducted all 15 tests of NIST SP 800-22 on the LEAP scheme, using 10^6 bits for each of the 128 samples as input. Table 6 shows the proportion of accepted bit sequences, and all three security levels of the LEAP scheme passed the tests. The scheme surpasses the minimum probability threshold in each test category, meeting all statistical characteristics of randomness satisfactorily.

Table 6: Test Results of NIST SP 800-22

* Detailed test results are provided in Appendix A.

Test	Proportion					
	LEAP-128		LEAP-192		LEAP-256	
Frequency	100.00%	Passed	100.00%	Passed	100.00%	Passed
BlockFrequency	97.66%	Passed	99.22%	Passed	100.00%	Passed
CumulativeSums	100.00%	Passed	97.66%	Passed	99.22%	Passed
Runs	99.22%	Passed	100.00%	Passed	100.00%	Passed
LongestRun	99.22%	Passed	99.22%	Passed	99.22%	Passed
Rank	99.22%	Passed	97.66%	Passed	99.22%	Passed
FFT	100.00%	Passed	100.00%	Passed	100.00%	Passed
NonOverlappingTemplate	99.22%	Passed	99.22%	Passed	99.22%	Passed
OverlappingTemplate	99.22%	Passed	100.00%	Passed	99.22%	Passed
Universal	96.88%	Passed	98.44%	Passed	98.44%	Passed
ApproximateEntropy	99.22%	Passed	97.66%	Passed	99.22%	Passed
RandomExcursions	98.73%	Passed	100.00%	Passed	100.00%	Passed
RandomExcursionsVariant	100.00%	Passed	100.00%	Passed	98.63%	Passed
Serial	100.00%	Passed	100.00%	Passed	98.63%	Passed
LinearComplexity	99.22%	Passed	99.22%	Passed	99.22%	Passed

5 Implementations and Performance

We implementated LEAP using plain `C`, `AVX2` and `AVX512`. This section presents the implementation details and comprehensive performance results of LEAP. The results offer detailed data on the scheme under various security levels, enabling adaptable parameter selection according to specific needs.

5.1 Implementations Details

Polynomial Multiplication and Inversion Polynomial multiplication and inversion are the most computationally expensive operations in the LEAP scheme. Therefore, optimizing the efficiency of polynomial multiplication and inversion is crucial for improving the overall efficiency of the scheme. In the LEAP, we use the NTT algorithm for fast polynomial multiplication and inversion. Since $257 \bmod (2 \cdot 512) \neq 1$, LEAP.GEN for all three security levels perform an incomplete NTT which contains 7 layers of NTTs. The BaseMul algorithm employs 4×4 schoolbook polynomial multiplication, while the BaseInv algorithm employs 4×4 schoolbook polynomial inversion. As for LEAP.PRNG, it also uses an incomplete NTT algorithm, where polynomial multiplication involves 7 layers of NTTs, using a 2×2 schoolbook polynomial multiplication as the BaseMul algorithm. In the implementation of the NTTs, we employ both Montgomery reduction and Barrett reduction for fast modular operation. Furthermore, we incorporate the Lazy Reduction technique, as mentioned in [Sei18], to further optimize the efficiency of NTT computations. Specifically, for the different parameters, we optimize the modular operation according to the magnitude of data growth during the NTT calculation process which depends on the size of the modulus and the word size boundary. If the data growth remains within the word size boundary,

we optimize the calculations by omitting modular arithmetic, resulting in computational savings.

Rounding In Section 3.2, we discuss the parameter selection strategy for HRLWR instances. Specifically, we set q' as a power-of-2 multiple of p. This choice facilitates fast rounding calculations through bit shift operation. Through a simple analysis, we can conclude that the distribution obtained from the right-shift operation remains unchanged compared to the distribution obtained from the original rounding operation. As a result, this approach does not compromise the security of the schemes.

Vectorization with AVX Instructions The LEAP scheme is designed with the goal of achieving efficient implementation by leveraging `SIMD` instructions. The majority of processes in the scheme can be optimized through the use of `AVX` instructions. In addition to the implementation provided in the plain `C`, we also implement schemes using `AVX2` and `AVX512` instruction sets. These instruction sets offer wider vector registers, enabling parallel processing of multiple elements in a single instruction. `AVX2` supports 256-bit wide `SIMD` registers, while `AVX512` extends this further to 512-bit. With our chosen parameters requiring a 16-bit word size for storage, `AVX2` and `AVX512` implementations enable loading 16 and 32 coefficients, respectively, into a single vector register to facilitate parallel computation. The implementation of the NTT process is highly amenable to vectorization, making the utilization of `AVX2` and `AVX512` instructions highly advantageous, leading to significant improvements in computational efficiency. Furthermore, within the LEAP, several key components, such as rejection sampling and bit shift for fast rounding computation, can also be accelerated through the use of `AVX` instructions. A comprehensive performance comparison of different implementations is provided in Section 5.2, offering detailed insights into the efficiency gains achieved by leveraging `AVX` instructions in the LEAP.

5.2 Performance

We provide three implementations of the LEAP: a plain `C` version, an `AVX2` version, and an `AVX512` version. We measure the CPU cycles and throughput on an Intel Core i9-11900K processor at a frequency of 3.5GHz while disabling TurboBoost and hyperthreading. The average counts of CPU cycles are measured for 100,000 iterations. The compiler used is clang version 16.0.6 with the following compilation flags: `-O3, -march=native, -mtune=native, -fomit-frame-pointer`.

Table 7, 8, and 9 respectively provide detailed performance test results (single thread only) for the three security levels of LEAP, showcasing various metrics such as the CPU cycles required to generate per byte, the throughput in generating gigabits per second, and the CPU cycles needed by the LEAP.GEN algorithm. To evaluate the performance of LEAP in detail, we test different values of m, incrementing by 16 until reaching optimal performance on our test machine. As m grows, performance does not improve because the time required for the NTT on s in line 3 of Algorithm 4 is evenly distributed across m parts. However, as m increases, the time allocated to each part becomes significantly smaller, resulting in diminishing performance improvements. It is important to note that the optimal value of m varies depending on the software and hardware conditions of the specific test machine. Small variations in parameter selection across the three security levels lead to similar performance as m increases. The data presented in tables shows that, after our optimal implementation, the `AVX2` and `AVX512` versions can achieve performance improvements of about 9 and 13 times, respectively, compared with the plain `C` implementation.

Table 7: Performance of LEAP-128

m	Cycles/byte			Gbps			LEAP.GEN (Kcycles)		
	C	AVX2	AVX512	C	AVX2	AVX512	C	AVX2	AVX512
1	20.13	3.08	2.30	1.11	4.17	4.96	81.34	8.56	6.62
16	13.82	1.68	1.20	1.85	14.08	19.02	248.08	33.40	24.49
32	13.50	1.65	1.17	1.91	15.08	20.83	415.33	60.23	44.17
48	13.41	1.63	1.16	1.92	15.46	21.47	588.94	85.95	64.01
64	13.34	1.62	1.15	1.94	15.69	21.79	762.42	113.61	83.67
80	13.30	1.62	1.15	1.94	15.77	21.98	935.78	140.54	103.49
96	13.28	1.62	1.15	1.95	15.88	22.15	1109.52	167.08	123.14
112	13.27	1.61	1.15	1.95	15.95	22.27	1283.71	194.22	142.83
128	13.25	1.61	1.15	1.95	16.00	22.37	1464.89	220.97	162.42
144	13.24	1.61	1.14	1.95	16.02	22.42	1628.48	247.65	182.18
160	13.24	1.61	1.14	1.95	16.05	22.47	1801.39	274.52	201.81
176	13.22	1.61	1.14	1.96	16.07	22.54	1974.95	301.20	221.60
192	13.22	1.61	1.14	1.96	16.09	22.56	2157.38	327.86	241.35
208	13.21	1.61	1.14	1.96	16.12	22.60	2311.81	354.51	260.37

Table 8: Performance of LEAP-192

m	Cycles/byte			Gbps			LEAP.GEN (Kcycles)		
	C	AVX2	AVX512	C	AVX2	AVX512	C	AVX2	AVX512
1	30.56	4.51	3.34	0.73	2.87	3.30	82.01	8.79	6.73
16	13.91	1.72	1.22	1.85	13.80	18.72	260.22	36.98	26.87
32	13.52	1.66	1.18	1.90	14.86	20.66	451.83	67.39	48.73
48	13.43	1.64	1.17	1.92	15.42	21.39	646.47	97.75	70.60
64	13.38	1.63	1.17	1.93	15.63	21.71	836.73	128.06	92.46
80	13.33	1.62	1.15	1.94	15.76	21.95	1032.78	158.35	114.37
96	13.31	1.62	1.15	1.94	15.83	22.08	1225.73	188.64	136.49
112	13.30	1.62	1.15	1.94	15.91	22.24	1418.78	218.86	158.09
128	13.29	1.61	1.15	1.95	15.97	22.31	1609.91	249.31	180.11
144	13.27	1.61	1.15	1.95	16.00	22.39	1804.22	280.33	201.64
160	13.26	1.61	1.14	1.95	16.01	22.39	1998.13	310.01	223.64
176	13.26	1.61	1.14	1.95	16.05	22.51	2191.59	340.27	245.60
192	13.25	1.61	1.14	1.96	16.06	22.52	2386.04	370.31	267.29
208	13.25	1.61	1.14	1.96	16.10	22.56	2573.71	400.62	289.63

Table 9: Performance of LEAP-256

m	Cycles/byte			Gbps			LEAP.GEN (Kcycles)		
	C	AVX2	AVX512	C	AVX2	AVX512	C	AVX2	AVX512
1	37.90	5.83	3.35	0.58	2.22	3.30	82.17	9.16	7.00
16	13.97	1.73	1.23	1.84	13.72	17.90	284.58	43.58	31.84
32	13.62	1.66	1.18	1.89	14.94	20.65	500.26	79.90	58.24
48	13.60	1.64	1.19	1.90	15.19	20.97	718.36	116.45	84.31
64	13.50	1.63	1.17	1.91	15.59	21.70	938.62	153.41	111.20
80	13.49	1.62	1.16	1.92	15.74	21.68	1155.65	189.76	137.55
96	13.47	1.62	1.16	1.92	15.82	21.98	1372.26	226.22	164.06
112	13.46	1.62	1.15	1.92	15.89	22.12	1592.92	263.19	190.71
128	13.46	1.62	1.15	1.92	15.91	22.26	1809.57	299.63	217.21
144	13.46	1.62	1.15	1.92	15.98	22.38	2022.56	336.63	243.93
160	13.45	1.61	1.15	1.92	16.00	22.44	2239.02	372.19	269.87
176	13.45	1.61	1.15	1.92	16.05	22.49	2457.97	410.61	296.84
192	13.43	1.61	1.14	1.93	16.05	22.52	2675.97	447.04	323.42
208	13.42	1.61	1.14	1.93	16.09	22.55	2890.64	482.01	349.29

6 Conclusion

Pseudorandom number generators play a crucial role in cryptography. However, research on lattice-based pseudorandom number generators remains relatively limited. Based on the NTRU variant and the RLWR assumption, we propose a novel lattice-based pseudorandom number generator scheme. This scheme significantly enhances the performance of lattice-based generators while reducing the key size. We provide guidelines for parameter selection at different security levels and conduct a thorough security reduction, along with a concrete security evaluation of the scheme. Furthermore, the proposed scheme offers flexible parameter adjustment capabilities, allowing it to be tailored to meet various practical application requirements.

References

[ABB10a] Shweta Agrawal, Dan Boneh, and Xavier Boyen. Efficient lattice (H)IBE in the standard model. In Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29, pages 553–572. Springer, 2010.

[ABB10b] Shweta Agrawal, Dan Boneh, and Xavier Boyen. Lattice basis delegation in fixed dimension and shorter-ciphertext hierarchical IBE. In Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30, pages 98–115. Springer, 2010.

[ACF+15a] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. Algebraic algorithms for LWE problems. ACM Commun. Comput. Algebra, 49(2):62, aug 2015.

[ACF+15b] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. On the complexity of the bkw algorithm on lwe. In L. Budach, editor, Designs, Codes and Cryptography, pages 325–354, 2015.

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jiří Sgall, editors, Automata, Languages and Programming, pages 403–415, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[AKPW13] Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. Learning with rounding, revisited. In Ran Canetti and Juan A. Garay, editors, Advances in Cryptology – CRYPTO 2013, pages 57–74, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Journal of Mathematical Cryptology, 9(3):169–203, 2015.

[ASA16] Jacob Alperin-Sheriff and Daniel Apon. Dimension-preserving reductions from LWE to LWR. IACR Cryptol. ePrint Arch., 2016:589, 2016.

[BBL+15] Abhishek Banerjee, Hai Brenner, Gaëtan Leurent, Chris Peikert, and Alon Rosen. SPRING: Fast pseudorandom functions from rounded ring products. In Carlos Cid and Christian Rechberger, editors, Fast Software Encryption, pages 38–57, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[BCD+16] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pages 1006–1018, 2016.

[BDFK17] Charles Bouillaguet, Claire Delaplace, Pierre-Alain Fouque, and Paul Kirchner. Fast lattice-based encryption: Stretching SPRING. In Tanja Lange and Tsuyoshi Takagi, editors, Post-Quantum Cryptography, pages 125–142, Cham, 2017. Springer International Publishing.

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-kyber: a CCA-secure module-lattice-based KEM. In 2018 IEEE European Symposium on Security and Privacy (EuroS&P), pages 353–367. IEEE, 2018.

[BG14] Shi Bai and Steven D. Galbraith. Lattice decoding attacks on binary LWE. In Willy Susilo and Yi Mu, editors, Information Security and Privacy, pages 322–337, Cham, 2014. Springer International Publishing.

[BGM+16] Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the hardness of learning with rounding over small modulus. In Eyal Kushilevitz and Tal Malkin, editors, Theory of Cryptography, pages 209–224, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT), 6(3):1–36, 2014.

[BK15] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators, 2015-06-24 2015.

[BKW00] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model, 2000.

[BM82] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo random bits. 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982), pages 112–117, 1982.

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, Advances in Cryptology – EUROCRYPT 2012, pages 719–737, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BRS+10] Lawrence Bassham, Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Stefan Leigh, M Levenson, M Vangel, Nathanael Heckert, and D Banks. A statistical test suite for random and pseudorandom number generators for cryptographic applications, 2010-09-16 2010.

[BV14] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. SIAM Journal on computing, 43(2):831–871, 2014.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Tfhe: fast fully homomorphic encryption over the torus. Journal of Cryptology, 33(1):34–91, 2020.

[CHKP12] David Cash, Dennis Hofheinz, Eike Kiltz, and Chris Peikert. Bonsai trees, or how to delegate a lattice basis. Journal of cryptology, 25:601–639, 2012.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, Advances in Cryptology – ASIACRYPT 2017, pages 409–437, Cham, 2017. Springer International Publishing.

[CS20] Chitchanok Chuengsatiansup and Damien Stehlé. Towards practical GGM-based PRF from (Module-)Learning-with-Rounding. In Kenneth G. Paterson and Douglas Stebila, editors, Selected Areas in Cryptography – SAC 2019, pages 693–713, Cham, 2020. Springer International Publishing.

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. volume 19, pages 297–301, 1965.

[DLP14] Léo Ducas, Vadim Lyubashevsky, and Thomas Prest. Efficient identity-based encryption over NTRU lattices. In Palash Sarkar and Tetsu Iwata, editors, Advances in Cryptology – ASIACRYPT 2014, pages 22–41, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[DvW21] Léo Ducas and Wessel van Woerden. NTRU fatigue: How stretched is overstretched? In Mehdi Tibouchi and Huaxiong Wang, editors, Advances in Cryptology – ASIACRYPT 2021, pages 3–32, Cham, 2021. Springer International Publishing.

[FHK+19] 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. 2019.

[FSK15] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. Generating randomness. Cryptography Engineering: Design Principles and Practical Applications, pages 135–161, 2015.

[GGH+19] Nicholas Genise, Craig Gentry, Shai Halevi, Baiyu Li, and Daniele Micciancio. Homomorphic encryption for finite automata. In Steven D. Galbraith and Shiho Moriai, editors, Advances in Cryptology – ASIACRYPT 2019, pages 473–502, Cham, 2019. Springer International Publishing.

[GGM84] O. Goldreich, S. Goldwasser, and S. Micali. How to construct randolli functions. In 25th Annual Symposium onFoundations of Computer Science, 1984., pages 464–479, 1984.

[GS66] W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In Proceedings of the November 7-10, 1966, fall joint computer conference, pages 563–578, 1966.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe P. Buhler, editor, Algorithmic Number Theory, pages 267–288, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[HW03] Peter Hellekalek and Stefan Wegenkittl. Empirical evidence concerning AES. ACM Transactions on Modeling and Computer Simulation (TOMACS), 13(4):322–333, 2003.

[Kan87] Ravi Kannan. Minkowski's convex body theorem and integer programming. Mathematics of Operations Research, 12(3):415–440, 1987.

[KSF99] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In International Workshop on Selected Areas in Cryptography, pages 13–33. Springer, 1999.

[LW20] Feng-Hao Liu and Zhedong Wang. Rounding in the rings. In Daniele Micciancio and Thomas Ristenpart, editors, Advances in Cryptology – CRYPTO 2020, pages 296–326, Cham, 2020. Springer International Publishing.

[NR95] M. Naor and O. Reingold. Synthesizers and their application to the parallel construction of pseudo-random functions. In Proceedings of IEEE 36th Annual Foundations of Computer Science, pages 170–181, 1995.

[PMS21] Alice Pellet-Mary and Damien Stehlé. On the hardness of the NTRU problem. In Mehdi Tibouchi and Huaxiong Wang, editors, Advances in Cryptology – ASIACRYPT 2021, pages 3–35, Cham, 2021. Springer International Publishing.

[PW08] Chris Peikert and Brent Waters. Lossy trapdoor functions and their applications. In Proceedings of the fortieth annual ACM symposium on Theory of computing, pages 187–196, 2008.

[SE91] C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In L. Budach, editor, Fundamentals of Computation Theory, pages 68–85, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[Sei18] Gregor Seiler. Faster AVX2 optimized ntt multiplication for ring-LWE lattice cryptography, 2018. Report 2018/039.

[XZD+23] Binwu Xiang, Jiang Zhang, Yi Deng, Yiran Dai, and Dengguo Feng. Fast blind rotation for bootstrapping FHEs. In Helena Handschuh and Anna Lysyanskaya, editors, Advances in Cryptology – CRYPTO 2023, pages 3–36, Cham, 2023. Springer Nature Switzerland.

[Yao82] Andrew Chi-Chih Yao. Theory and applications of trapdoor functions (extended abstract). In IEEE Annual Symposium on Foundations of Computer Science, 1982.

A The Test Results of NIST SP 800-22

Each test relies on a calculated test statistic, derived from the data, to compute a P-value that summarizes the strength of evidence opposing the null hypothesis. A P-value of one suggests the sequence exhibits perfect randomness, while zero indicates complete non-randomness. These tests allow for the selection of a significance level (α), set at 0.01 in this case. If the P-value is greater than or equal to α, the null hypothesis is accepted, indicating randomness in the sequence. Conversely, if the P-value is less than α, the null hypothesis is rejected, signifying non-randomness. The 'proportion' denotes the ratio of successful sequences for a specific statistical test.

Table 10: Test Results of LEAP-128

Test	P-value	Proportion	Accepted/Rejected
Frequency	0.568055	128/128	Accepted
BlockFrequency	0.534146	125/128	Accepted
CumulativeSums	0.834308	128/128	Accepted
Runs	0.242986	127/128	Accepted
LongestRun	0.484646	127/128	Accepted
Rank	0.500934	127/128	Accepted
FFT	0.484646	128/128	Accepted
NonOverlappingTemplate	0.232760	127/128	Accepted
OverlappingTemplate	0.232760	127/128	Accepted
Universal	0.534146	124/128	Accepted
ApproximateEntropy	0.110952	127/128	Accepted
RandomExcursions	0.386280	78/79	Accepted
RandomExcursionsVariant	0.256632	79/79	Accepted
Serial	0.392456	128/128	Accepted
LinearComplexity	0.739918	127/128	Accepted

Table 11: Test Results of LEAP-192

Test	P-value	Proportion	Accepted/Rejected
Frequency	0.706149	128/128	Accepted
BlockFrequency	0.991468	127/128	Accepted
CumulativeSums	0.534146	125/128	Accepted
Runs	0.105618	128/128	Accepted
LongestRun	0.148094	127/128	Accepted
Rank	0.105618	125/128	Accepted
FFT	0.834308	128/128	Accepted
NonOverlappingTemplate	0.178278	127/128	Accepted
OverlappingTemplate	0.957319	128/128	Accepted
Universal	0.437274	126/128	Accepted
ApproximateEntropy	0.551026	125/128	Accepted
RandomExcursions	0.788728	128/128	Accepted
RandomExcursionsVariant	0.414525	82/82	Accepted
Serial	0.848588	82/82	Accepted
LinearComplexity	0.723129	127/128	Accepted

Table 12: Test Results of LEAP-256

Test	**P-value**	**Proportion**	**Accepted/Rejected**
Frequency	0.862344	128/128	Accepted
BlockFrequency	0.654467	128/128	Accepted
CumulativeSums	0.392456	127/128	Accepted
Runs	0.941144	128/128	Accepted
LongestRun	0.264458	127/128	Accepted
Rank	0.637119	127/128	Accepted
FFT	0.311542	128/128	Accepted
NonOverlappingTemplate	0.756476	127/128	Accepted
OverlappingTemplate	0.804337	127/128	Accepted
Universal	0.654467	126/128	Accepted
ApproximateEntropy	0.568055	127/128	Accepted
RandomExcursions	0.491599	128/128	Accepted
RandomExcursionsVariant	0.316916	72/73	Accepted
Serial	0.637119	72/73	Accepted
LinearComplexity	0.931952	127/128	Accepted