Efficient generation of the public parameter $\boldsymbol{A}$ in unstructured lattice-based schemes

Hayo Baan, Sauvik Bhattacharya, Oscar Garcia-Morchon, and Ludo Tolhuizen

Royal Philips N.V., Netherlands.

Abstract. Key encapsulation mechanisms and public-key encryption schemes based on the Learning with Errors or Learning with Rounding problems use a matrix $\boldsymbol{A}$ as public parameter, that defines the underlying lattice controlling the security of the scheme. A common technique to avoid threats such as pre-computation attacks is to refresh $\boldsymbol{A}$ with each protocol instantiation. However, $\boldsymbol{A}$ is a large matrix of dimension $d \times d$ with elements in $\mathbb{Z}_q$. Thus, the generation of $\boldsymbol{A}$ forms a very significant fraction of the computational overhead of these schemes.
In this manuscript, we analyze different methods for the generation of $\boldsymbol{A}$ from Round5 and Frodo, unstructured lattice-based submissions to the NIST PQC project. We discuss how these methods influence the resistance of the resulting scheme to pre-computation and backdoor attacks, formal proofs of security, exploitable structure in the resulting underlying lattice, and performance of the scheme. We encourage the community to study the methods proposed in Round5 since they lead to a considerable performance speed-up.

Keywords: Lattice cryptography · Public-key encryption · Key-generation

1 Introduction

The NIST Post-Quantum Cryptography (PQC) project [18] is evaluating a number of post-quantum key encapsulation mechanisms (KEM) and public-key encryption (PKE) schemes for standardization. In the second round of the project there are nine schemes remaining in the category of lattice-based schemes , namely FrodoKEM [2], CRYSTALS-Kyber [4], LAC [16], NewHope [1], NTRU [10], NTRUPrime [8], Round5 [5], Saber [13], and Threebears [14]. Most of those schemes belong to the category of *structured lattices* used to achieve higher efficiency [17]. Only FrodoKEM (henceforth referred to as "Frodo") and Round5 in its non-ring configuration ("n1" parameter sets) offer parameters or instantiations where the underlying hardness assumptions are based on unstructured lattice-based problems [21,6]. Although such solutions offer more conservative choices in the case that advances in cryptanalysis lead to attacks on the structured alternatives [12,11,20], they also involve a higher overhead due to the absence of structure in the lattice.

In more detail, unstructured lattices bring two main downsides: they lead to *larger key sizes* and *slower computational performance*. Table 1 shows that

Table 1: Bandwidth requirements (in Bytes) in second round NIST PQC candidates. Frodo and Round5 n1 are the unstructured solutions analyzed in this paper and require around 10x more bandwidth than structured solutions.

			IND-CPA KEM					IND-CCA KEM				
Category	Scheme	Configuration	1	2	3	4	5	1	2	3	4	5
Unstructured	Frodo							19336	-	31376	-	43152
		n1	10450	-	17700	-	28552	11544	-	19393	-	29360
Structured	Round5	nd_5	994	-	1639	-	2035	1097	-	1730	-	2279
		nd_0	1316	-	1890	-	2452	1432	-	2102	-	2874
	Threebears							1721	-	2501	-	3281
	Saber							1408	-	2074	-	2784
	Kyber							1536	-	2272	-	3136
	NewHope		1797				4000	2048	-	-	-	4032
	LAC							1256	-	2244	-	2480
	NTRUPrime	st						-	1891	2197	2506	-
		lpr						-	1922	2206	2496	-
	NTRU	hps						1398	-	1862	-	2460
		hrss						-	-	2276	-	-

Frodo and Round5(n1) have keys that are around 10 times larger than those of the structured lattice-based alternatives. Computational or CPU performance is slower because of two reasons: more random data is needed to generate $\boldsymbol{A}$, and more operations are needed to compute the public-key and ciphertext components. Unstructured lattice-based schemes use a public $d \times d$ matrix $\boldsymbol{A}$. This is a factor d larger compared with the structured case. For instance, Round5 in its ring configurations only requires a polynomial $\boldsymbol{a}$ containing d elements. For Frodo and Round5, each coefficient of $\boldsymbol{A}$ fits in two bytes. This results in matrices $\boldsymbol{A}$ with sizes between 0.75 and 3.6 MBytes. The generation process of the overall public-key, which has more components in the unstructured case, is also obviously slower than in the structured case, because more vector-matrix multiplications are required.

This paper focuses on the step in the protocol of KEM and PKE schemes that generates the public-parameter $\boldsymbol{A}$. Specifically, we consider different ways of carrying out this generation to achieve specific performance and security goals. We specify the following design goals:

- **Formal proofs of security**: Public-key encryption schemes are typically required to show a proof of semantic security of the ciphertext against (at least) passive adversaries. The common technique of proving indistinguishability (IND) of lattice-based schemes against a chosen plaintext adversary (CPA) requires the chosen public parameter, i.e., including the matrix $\boldsymbol{A}$, to be indistinguishable from a uniformly generated one [15].
- **Avoiding structure in the lattice**: A fundamental reason to design schemes based on unstructured lattice-based problems is to make sure that no ob-

vious structure in the underlying lattice remains that can be exploited in attacks, such as those considered in [12,11,20].
- **Resistance to pre-computation attacks**: In the protocols described in [19] and [9], the public parameter $\boldsymbol{A}$ remains fixed for all protocol instantiations. However, such a design might make the scheme prone to pre-computation attacks, as discussed in [3].
- **Resistance to backdoor attacks**: Similar to the previous point, the generation of the public parameter $\boldsymbol{A}$ should make it possible to claim that any exploitable (unknown) structure in the underlying lattice has been avoided.
- **Fast generation of $\boldsymbol{A}$**: This is a fundamental goal to facilitate the integration of schemes into real-world protocols such as TLS or IKEv2. This is important on the responder side, e.g., a server that has to handle many connections (and must instantiate a fresh $\boldsymbol{A}$ for each) and also on the initiator side, e.g., a resource-constrained client, since fast generation implies fewer CPU cycles and hence less power usage.
- **Low storage needs**: This is also important for both parties of the communication. A server having to handle many connections should keep memory needs for storing $\boldsymbol{A}$ as low as possible. Further, resource-constrained clients have limited memory capabilities, making it also an important goal for them.

The rest of the paper is organized as follows: Section 2 introduces basic notation. Section 3 describes methods to generate the public-parameter $\boldsymbol{A}$. Section 4 discusses advantages and disadvantages of the different methods. Section 5 concludes this paper.

2 Notation

A public-key encryption (PKE) scheme is defined as a triple of functions PKE = $(\texttt{Keygen}, \texttt{Enc}, \texttt{Dec})$ with message space $\mathcal{M}$, where given a security parameter λ `Keygen` returns a secret key sk and public key pk, `Enc` encrypts a message $m \in \mathcal{M}$ using pk to produce a ciphertext ct, and `Dec` returns an estimate m' of m given ct and sk.

The decisional Learning with Errors (LWE) [21] problem involves distinguishing the uniform sample $(\boldsymbol{A}, \boldsymbol{U}) \leftarrow \mathcal{U}(\mathbb{Z}_q^{k_1 \times k_2} \times \mathbb{Z}_q^{k_1 \times m})$ from the LWE sample $(\boldsymbol{A}, \boldsymbol{B} = \langle \boldsymbol{AS} + \boldsymbol{E} \rangle_q)$ where $\boldsymbol{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{k_1 \times k_2})$ and where the secret key $\boldsymbol{S}$ and error $\boldsymbol{E}$ are generated from the secret and error distributions χ_s and χ_e respectively. The search problem is to recover $\boldsymbol{S}$ from the LWE sample.

The decisional generalized Learning with Rounding (LWR) [6] problem involves distinguishing the uniform sample $(\boldsymbol{A}, \lfloor p/q \cdot \boldsymbol{U} \rceil)$ where $\boldsymbol{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{k_1 \times k_2})$ and $\boldsymbol{U} \leftarrow \mathcal{U}(\mathbb{Z}_q^{k_1 \times m})$ from the generalized LWR sample $(\boldsymbol{A}, \boldsymbol{B} = \left\lfloor p/q \cdot \langle \boldsymbol{AS} \rangle_q \right\rceil)$ where $\boldsymbol{A} \leftarrow \mathcal{U}(\mathbb{Z}_q^{k_1 \times k_2})$, and $\boldsymbol{S} \leftarrow \chi_s(\mathbb{Z}_q^{k_2 \times m})$. Analogous to the LWE case, the search problem is to recover $\boldsymbol{S}$ from the generalized LWR sample.

Algorithm 1 shows the steps required in the key generation when LWR is used as the underlying problem. Line 1 shows that the first step consists in obtaining matrix $\boldsymbol{A}$. Line 2 includes the steps of sampling the secret and error

vectors. Line 3 computes public-key component $\boldsymbol{B}$. Line 4 returns both public- and private-keys.

Algorithm 1: Keygen()

1 $\boldsymbol{A} \leftarrow$ Sample_A()
2 $\boldsymbol{S}_A \leftarrow \chi_s$, $\boldsymbol{E}_A \leftarrow \chi_e$
3 $\boldsymbol{B} = \left\lfloor p/q \cdot \langle \boldsymbol{A}\boldsymbol{S}_A + \boldsymbol{E}_A \rangle_q \right\rceil$
4 **return** $(pk = (\boldsymbol{A}, \boldsymbol{B}), sk = \boldsymbol{S}_A)$

3 Methods

We describe four methods that can be used to determine the public-parameter $\boldsymbol{A}$, i.e., to instantiate function `Sample_A()` in Line 1 in Algorithm 1. We assume that each entry of $\boldsymbol{A}$ can be represented in two bytes.

3.1 Method 0: $\boldsymbol{A}$ defined per protocol instance

This is the method used in all second round NIST candidates based on (R/M)LW(E/R). The idea is to generate the entries of $\boldsymbol{A}$ by applying a deterministic random bit generator (DRBG) on a random seed σ unique to and freshly chosen for each protocol exchange. This means that the initiator generates random seed σ and sends it to the responder. Both initiator and responder use σ to obtain $\boldsymbol{A}$ by means of the DRBG, which must be called to obtain $2 * d^2$ bytes in each protocol exchange, where $d \in \mathbb{Z}$ is the main security parameter of the scheme. Typically, $d \geq 700$.

Both Frodo and Round5 make use of this approach, in particular, Round5 denotes this option as $f^{(0)}_{d,n}$. Both schemes rely on instantiations of the DRBG based on AES or SHAKE. For the first, AES is used in a "counter-mode" approach, basically applying AES to a set of pre-defined values dependent on the row and column of $\boldsymbol{A}$.

In the case of SHAKE, Round5 generates the elements sequentially; in contrast, Frodo can generate several rows in parallel since for each row SHAKE is called with seed σ prefixed with a counter that depends on the row [2].

3.2 Method 1: Permuting a publicly pre-defined $\boldsymbol{A_{master}}$

This approach consists of generating $\boldsymbol{A}$ by permuting a public and system-wide matrix $\boldsymbol{A_{master}}$. This means that $\boldsymbol{A_{master}}$ is defined a priori, e.g., following Method 0 discussed in Section 3.1. The permutation is derived from a random seed σ that is unique to and freshly chosen for each protocol exchange. This means that the initiator generates a random seed and sends it to the responder. Both initiator and responder use this seed to obtain the same permutation.

This method is an optional approach in Round5 denoted as $f^{(1)}_{d,n}$. Round5 describes the permutation as a vector $\boldsymbol{p}_1 \in Z^d_d$ and the fresh $\boldsymbol{A}$ is defined as

$\boldsymbol{A}[i,j] = \boldsymbol{A_{master}}[i, \langle \boldsymbol{p}_1[i] + j \rangle_d]$. Here, $\boldsymbol{A}[i,j]$ refers to the j^{th} entry of the i^{th} row in $\boldsymbol{A}$ and $\langle a \rangle_d$ means *a modulo d*. This means that row i in $\boldsymbol{A}$ consists of a rotation – determined by $\boldsymbol{p}_1[i]$ – of row i in $\boldsymbol{A_{master}}$. The entries of $\boldsymbol{p}_1$ are obtained from a DRBG initialized with σ and with uniform output in $\mathbb{Z}_d$.

With this approach, the DRBG only needs to deliver $2 * d$ bytes, a factor d less compared with Method 0.

3.3 Method 2: Permuting an a_{master} defined per protocol instance

This approach involves four steps: (1) the initiator obtains a random seed σ specific to each protocol exchange and sends it to the responder; (2) both initiator and responder generate a vector $\boldsymbol{a_{master}} \in \mathbb{Z}_q^{len\_a\_master}$ by applying a DRBG on seed σ; (3) both initiator and responder obtain a permutation $\boldsymbol{p}_2$ by applying a DRBG on the seed σ, and finally (4) both initiator and responder compute $\boldsymbol{A}$ by permuting $\boldsymbol{a_{master}}$ according to $\boldsymbol{p}_2$.

This is the default approach in Round5 for its non-ring parameters and it is denoted as $f^{(2)}_{d,n}$. The entries of the permutation vector $\boldsymbol{p}_2$ are obtained from a DRBG initialized with σ and with uniform output in $\mathbb{Z}_{len\_a\_master}$; rejection sampling is used to ensure that the entries of $\boldsymbol{p}_2$ are distinct. The matrix $\boldsymbol{A}$ is defined as $A[i,j] = \boldsymbol{a}_{master}[\langle \boldsymbol{p}_2[i] + j \rangle_{len\_a\_master}]$. With this approach, it is possible to efficiently obtain a fresh $\boldsymbol{A}$ without having to compute a significant amount of pseudorandom data. In particular, this approach only requires extracting $2 * len\_a\_master + d * \log_2(len\_a\_master)/8$ bytes from the DRBG. We ignored potential additional generation due to rejection sampling. This is justified if, as in the implementation of Round5, *len_a_master* is much larger than *d*. Furthermore, this approach does not require the storage of a large matrix $\boldsymbol{A_{master}}$.

3.4 Method 3: Standard defined A_{master}

This method has not been proposed in any NIST PQC submission, but its equivalent has appeared in the literature [19]. This option would consist of fixing a global public-parameter $\boldsymbol{A_{master}}$ that is identical for all protocol instantiations. In order to avoid backdoor attacks, the entries of $\boldsymbol{A_{master}}$ could be generated by applying a DRBG on a random seed σ as done in Method 0. In order to avoid any possibility of backdoors, the value of σ could be chosen from different randomness sources managed by different organizations.

4 Comparison

The following subsections describe how well the different methods described in Section 3 fulfill the design goals specified in the introduction of this paper, i.e., formal (indistinguishability) proofs of security, avoidance of lattice structure, prevention of backdoor and pre-computation attacks, and performance requirements regarding CPU and memory. Regarding the last point, it is important

to realize that the generation of $\boldsymbol{A}$ and the computations are very interlinked. This means that Lines 1 and 3 in Algorithm 1 are often mixed together in the implementations. Thus, it is difficult to give stand-alone values of the different methods and we present performance values for `Keygen`, `Enc`, and `Dec`. Similarly, implementations might trade CPU and memory needs depending on the target application.

4.1 Method 0: $\boldsymbol{A}$ defined per protocol instance

This is the standard approach followed by many NIST proposals, being initially proposed by NewHope [3] in the context of a structured lattice-based key-exchange scheme.

Security. Extending to the case of unstructured lattice-based schemes, the main advantage of this method is that security proofs can be phrased in terms of indistinguishability of a uniformly generated matrix $\boldsymbol{A}$ and a matrix $\boldsymbol{A}$ generated by a applying a DRBG on a seed. Hence, security proofs can be provided under the assumption that the DRBG outputs a uniform symbol string.

However, as recently pointed out in [7, Section 4], since the seed for generating $\boldsymbol{A}$ is public, the possibility of an attack exploiting the DBRG cannot be ruled out. Finally, this method defeats both pre-computation and backdoor attacks – as mentioned in NewHope [3] – since $\boldsymbol{A}$ is specific to each protocol instantiation and it is generated by means of a DRBG.

Performance. This method requires generating d^2 elements in $\mathbb{Z}_q$. This is the main reason of this method incurring a high performance overhead. Tables 2 and 3 summarize the cost in the case of FrodoKEM-640, FrodoKEM-976 and FrodoKEM-1344. These two tables correspond to the second entries in Table 5 and Table 6 in [2], respectively. The first table includes the performance when both AVX2 instructions and hardware-specific AES instructions are available. The second table shows the performance when using a 4-way implementation of SHAKE and AVX2 instructions. This means that the operations in 4 rows of $\boldsymbol{A}$ can be performed in parallel.

For comparison, we see the performance of R5N1_1PKE_0d using this same method in the first column of Table 4. The dimensions of the matrices $\boldsymbol{A}$ instantiated in Frodo640 (640×640) and in R5N1_1PKE_0d (636×636) are almost equal. The reason why R5N1_1PKE_0d performs worse is that its specific implementation that is considered here does not use a 4-way implementation of SHAKE.

We note that this method can be implemented using very low memory requirements, as low as a single element of $\boldsymbol{A}$ that needs to be kept in memory if $\boldsymbol{A}$ is generated on the fly. However, this usually comes at the price of a higher CPU overhead since more calls to the DRBG are required.

Table 2: Optimized (Fast AES) $\boldsymbol{A}$ generation in Frodo using Method 0. Results obtained when compiling https://github.com/Microsoft/PQCrypto-LWEKE with *make CC=gcc ARCH=x64 OPT_LEVEL=FAST GENERATION_A=AES128 USE_OPENSSL=TRUE OPENSSL_INCLUDE_DIR=/path/to/openssl/include OPENSSL_LIB_DIR=/path/to/openssl/lib* on a MacBookPro15.1 with an Intel Core i7 2.6GHz, running macOS 10.13.6.

	Variants		
	FrodoKEM-640	FrodoKEM-976	FrodoKEM-1344
Performance: Average elapsed time (ms)			
`Keygen`	0.4	0.8	1.3
`Enc`	0.6	1.2	1.8
`Dec`	0.6	1.1	1.8
Total	1.6	3.1	4.9
Performance: Average CPU Clock Cycles			
`Keygen`	1054.9k	2087.1k	3465.2k
`Enc`	1481.7k	3020.6k	4764.8k
`Dec`	1471.8k	2898.0k	4680.4k
Total	4008.4k	8985.7k	12910.4k

4.2 Method 1: Permuting a publicly pre-defined $\boldsymbol{A_{master}}$

With this method, the matrix $\boldsymbol{A}$ used in the protocol is derived from a fixed long-term system-wide matrix $\boldsymbol{A_{master}} \in \mathbb{Z}_q^{d \times d}$. This is done by applying to $\boldsymbol{A_{master}}$ a fresh permutation chosen by the initiator of the protocol at the start of each protocol exchange.

Security. The permutation consists of random, cyclic shifts of the rows of $\boldsymbol{A_{master}}$. This prevents any pre-computation attacks since the possible number of permuted versions $\boldsymbol{A}$ of $\boldsymbol{A_{master}}$ obtained in this way is d^d, a number high enough to make guessing the actual chosen $\boldsymbol{A}$ an infeasible task. Back-doors are avoided since $\boldsymbol{A_{master}}$ itself is derived by means of a pseudo-random function from a randomly generated seed (Method 0). Since $\boldsymbol{A}$ is created by simply permuting the row elements of $\boldsymbol{A_{master}}$, which itself was randomly generated, the underlying lattice resulting from $\boldsymbol{A}$ does not have any structure as in ideal lattices, for example. Proving formal security in terms of indistinguishability is not feasible if $(\boldsymbol{A}, \boldsymbol{B})$ is jointly considered as the public key. This is because $\boldsymbol{A_{master}}$ is publicly known and $\boldsymbol{A}$ is just a permuted version of it.

Performance. Computing this permutation is cheap, and thus Method 1 is efficient in terms of CPU overhead. The computational advantage is clear when looking at the first and second columns in Table 4 that shows at least a $8x$ factor speed up compared with Method 0. Memory-wise, $\boldsymbol{A_{master}}$ needs to

Table 3: Optimized (AVX2) $\boldsymbol{A}$ generation in Frodo using Method 0. Results obtained when compiling https://github.com/Microsoft/PQCrypto-LWEKE with *make CC=gcc ARCH=x64 OPT_LEVEL=FAST GENERATION_A=SHAKE128 USE_OPENSSL=FALSE* on a MacBookPro15.1 with an Intel Core i7 2.6GHz, running macOS 10.13.6.

	Variants		
	FrodoKEM-640	FrodoKEM-976	FrodoKEM-1344
Performance: Average elapsed time (ms)			
`Keygen`	1.1	2.4	4.4
`Enc`	1.2	2.6	4.7
`Dec`	1.2	2.5	4.6
Total	3.6	7.5	13.7
Performance: Average CPU Clock Cycles			
`Keygen`	2956.0k	6381.6k	11447.0k
`Enc`	3182.4k	6685.6k	12252.6k
`Dec`	3170.9k	6561.0k	11952.3k
Total	9309.3k	19628.2k	35651.9k

be kept in memory and simple permutations of it can be used for connecting to multiple clients. If keeping $\boldsymbol{A_{master}}$ in memory is not feasible, e.g., in a resource-constrained client, $\boldsymbol{A_{master}}$ can be generated on the fly row by row (following Method 0). However, this will deteriorate the computational performance making such an implementation slower than Method 0. This is the reason that motivated the design of Method 2.

4.3 Method 2: Permuting an a_{master} defined per protocol instance

The public parameter $\boldsymbol{A}$ is obtained from a vector $\boldsymbol{a}_{\text{master}} \in \mathbb{Z}_q^{len\_a\_master}$ that is randomly generated by means of a DBRG using a seed determined by the initiator in each protocol interaction. Each row in $\boldsymbol{A}$ is obtained from $\boldsymbol{a}_{\text{master}}$ by means of a random permutation that is also determined by the initiator and is specific to each protocol interaction, and ensures that $\boldsymbol{A}$ has distinct rows. In particular, each row of $\boldsymbol{A}$ consists of d consecutive entries of $\boldsymbol{a}_{\text{master}}$.

Security. The latter is the reason why an indistinguishability proof is not feasible in Method 2: elements in two different rows of $\boldsymbol{A}$ might have some common entries of $\boldsymbol{a}_{\text{master}}$, thus it is easy to distinguish $\boldsymbol{A}$ from random. We argue that pre-computation and back-door attacks are avoided since the seed that determines $\boldsymbol{A}$ is new in each protocol interaction, and this approach allows computing many $\boldsymbol{A}$'s: $\boldsymbol{A}$ is obtained by permuting – there are a total of $\binom{len\_a\_master}{d}$ ways of picking up d $d-$element vectors from $\boldsymbol{a}_{\text{master}}$. $\boldsymbol{a}_{\text{master}}$ is a fresh vector for which there are $q^{len\_a\_master}$ choices.

Table 4: Optimized (AVX2) R5N1_1PKE_0d – $\boldsymbol{A}$ generation variants with cSHAKE128. Results obtained running the code at https://github.com/round5/code on a MacBookPro15.1 with an Intel Core i7 2.6GHz, running macOS 10.13.6 and using the runSpeedTests application with command *./runSpeedTests -api-set 15 -avx2 1 -tau x* with *x*={*0,1,2*} corresponding to Method 0, 1, and 2, respectively.

	Variants		
	Method 0	**Method 1**	**Method 2**
Performance: Elapsed time (ms)			
`Keygen`	1.6	0.1	0.1
`Enc`	1.7	0.2	0.2
`Dec`	1.8	0.2	0.2
Total	5.1	0.6	0.5
Performance: CPU Clock Cycles			
`Keygen`	4306.1k	361.7k	360.2k
`Enc`	4474.0k	529.9k	500.1k
`Dec`	4549.9k	613.9k	579.6k
Total	13330.8k	1505.4k	1440k

We now analyze this method and how it destroys the circulant-like structure in the underlying lattice. If $len\_a\_master = d$, then this method, in particular its definition $f^{(2)}_{d,n}$ in Round5, results in a matrix $\boldsymbol{A}$ in which each row is obtained by a cyclic shift of the top row over some positions. Thus, Method 2 with $len\_a\_master = d$ essentially results in a cyclic matrix.

However, len_a_master is to be chosen greater than d. The most extreme case occurs when $len\_a\_master = d + 1$. In this case, we can visualize the structure of $\boldsymbol{A}$ by rotating $\boldsymbol{a_{master}}$ as in a circular matrix and removing (in red) the row that is not corresponding to any entry in $\boldsymbol{p2}$, and the last column.

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & \dots & a_{d-1} & a_d \\ a_d & a_0 & a_1 & a_2 & a_3 & \dots & a_{d-2} & a_{d-1} \\ a_{d-1} & a_d & a_0 & a_1 & a_2 & \dots & a_{d-3} & a_{d-2} \\ a_{d-2} & a_{d-1} & a_d & a_0 & a_1 & \dots & a_{d-4} & a_{d-3} \\ & & & & & \dots & & \\ a_2 & a_3 & a_4 & a_5 & a_6 & \dots & a_0 & a_1 \\ a_1 & a_2 & a_3 & a_4 & a_5 & \dots & a_d & a_0 \end{bmatrix} \rightarrow \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & \dots & a_{d-1} \\ a_{d-1} & a_d & a_0 & a_1 & a_2 & \dots & a_{d-3} \\ a_{d-2} & a_{d-1} & a_d & a_0 & a_1 & \dots & a_{d-4} \\ & & & & & \dots & \\ a_2 & a_3 & a_4 & a_5 & a_6 & \dots & a_0 \\ a_1 & a_2 & a_3 & a_4 & a_5 & \dots & a_d \end{bmatrix}_{d \times d}$$

For an arbitrary value of $len\_a\_master > d$, matrix $\boldsymbol{A}$ is obtained by first constructing the circular matrix using a_{master} (dimension $len\_a\_master \times len\_a\_master$ and then keeping the rows that correspond to the entries in $\boldsymbol{p2}$, and removing the last $len\_a\_master - d$ columns.

We now take a different perspective. We say that two rows of $\boldsymbol{A}$ overlap if they have at least one entry originating from the same entry of $\boldsymbol{a}_{\text{master}}$. If a row does

not overlap with any other row in $\boldsymbol{A}$, then $f_{d,n}^{(2)}$ is equivalent to $f_{d,n}^{(0)}$ regarding that row since all its entries have been obtained by applying a DRBG to a seed. Now we consider two overlapping rows of $\boldsymbol{A}$ that share $d-k$ elements, namely $a^{(0)} = \langle a_0, a_1, a_2, \ldots, a_{d-1}\rangle$ and $a^{(k)} = \langle a_L, a_{L-k+1}, \ldots, a_{L-1}, a_0, a_1, a_2, \ldots, a_{d-k-1}\rangle$, where we have denoted *len_a_master* as L for conciseness. If we define $a^{(0)}(x) \coloneqq a_0+a_1x+a_2x^2+\cdots+a_{d-1}x^{d-1}$ and $a^{(k)}(x) \coloneqq a_{L-k}+a_{L-k+1}x+\cdots+a_{L-1}x^{k-1}+a_0x^k + a_1x^{k+1} + \cdots + a_{d-k-1}x^{d-1}$, then we have $a^{(k)}(x) = a(x)x^k + (a_{L-k} - a_{d-k}) + (a_{L-k+1} - a_{d-k+1})x + \cdots + (a_{L-1} - a_{d-1})x^{k-1} \bmod x^d - 1$. Effectively, that is, each row can be seen as using the $x^d - 1$ ring with a random shifting (due to k) and additional random noises, those corresponding to $(a_{L-k}-a_{d-k})+(a_{L-k+1}-a_{n-k+1})x+\cdots+(a_{L-1}-a_{d-1})x^{k-1}$. The random shift is due to the the random permutation and the random noises are due to the random generation of $\boldsymbol{a}_{\text{master}}$.

When *len_a_master* is small, in the extreme case equal to d, there are no random-shifts since all rows are just shifted a single position and there are no noises since there are only d elements. When *len_a_master* increases, there are many ways of choosing d elements (shifts) out of *len_a_master* and there are up to *len_a_master* $- d$ random noises. This is the reason why this construction removes the circular structure in the resulting $\boldsymbol{A}$ and makes it harder to apply any potential attacks that might exploit this structure in the lattice.

Performance. Since only a few – *len_a_master* – elements need to be generated and kept in memory, $f_{d,n}^{(2)}$ is efficient both in terms of memory and CPU consumption. In particular, Round5 sets 2^{11} as the default value for *len_a_master*, the smallest power of two greater than the value of the parameter d for any of the Round5 configurations.

The computational advantage is clear when looking at the first and third columns in Table 4 that show a $9x$ factor speed up compared with Method 0. Comparing Tables 3 and 5 we observe that this optimization allows Round5 to achieve a factor 6x faster performance compared with Frodo when SHAKE is used as the underlying DRBG. This method reduces memory needs for both initiator and responder, which can be especially useful on small devices, since only $\boldsymbol{a_{master}}$ needs to be kept in memory requiring $2 * len\_a\_master$ bytes . This equals to only 4 KB in the default Round5 parameter choice.

We observe that the choice of *len_a_master*, i.e., the length of *a_master* plays a fundamental role in the security/performance trade-off of this method. A very small value, say approximately d, leads to a very efficient scheme but preserves some structure in the lattice that may or may not lead to exploits in future. A very high value, say close to d^2, can lead to a scheme equivalent to Method 0 if the permutation does not allow any overlap in the rows of $\boldsymbol{A}$. Intermediate values provide a trade-off between performance and the capability to remove any (anti-)circulant structure in the underlying lattice.

Table 5: Optimized (AVX2) R5N1_1PKE_0d – $\boldsymbol{A}$ generation variants with cSHAKE128. Results obtained running the code at https://github.com/round5/code on a MacBookPro15.1 with an Intel Core i7 2.6GHz, running macOS 10.13.6 and using the runSpeedTests application with command *./runSpeedTests -api-set x -avx2 1 -tau 2* with *x*={*15,16,17*} corresponding to R5N1_1PKE_0d, R5N1_3PKE_0d, and R5N1_5PKE_0d, respectively.

	Variants		
	R5N1_1PKE_0d	R5N1_3PKE_0d	R5N1_5PKE_0d
Performance: Elapsed time (ms)			
`Keygen`	0.1	0.3	0.4
`Enc`	0.2	0.4	0.5
`Dec`	0.2	0.6	0.6
Total	0.5	1.3	1.5
Performance: CPU Clock Cycles			
`Keygen`	360.2k	821.0k	1006.1k
`Enc`	500.1k	1064.8k	1310.3k
`Dec`	579.6k	1440.4k	1624.7k
Total	1440k	3326.1k	3941.1k

4.4 Method 3: Standard defined $\boldsymbol{A_{master}}$.

Works prior to NewHope [3], like [9] and [19], defined key exchange protocols or key encapsulation protocols with a fixed public parameter $\boldsymbol{A_{master}}$.

Security. A proof of formal security for such a construction is possible under the assumption that the matrix $\boldsymbol{A_{master}}$ is chosen such that it is indistinguishable from uniform. Further, this approach is prone to a pre-computation attack since an attacker is free to perform lattice reduction on the basis computed from $\boldsymbol{A_{master}}$ over an indefinite period of time. A back-door attack cannot be mounted *if* the fixed $\boldsymbol{A_{master}}$ is computed by means of a DRBG from a random seed (e.g., using Method 0). Further assurances can be provided if multiple entities independently generate the input random seed, or otherwise provide independent contributions to the generation of $\boldsymbol{A}$ in a publicly verifiable manner.

Performance. This method allows for the fastest performance since $\boldsymbol{A_{master}}$ could be stored in memory so no computations are required to generate it on the fly. Alternatively, it could of course be generated on the fly to reduce memory requirements, but then the CPU performance is reduced to that in Method 0.

5 Conclusions

Table 6 provides a final comparison of the four methods discussed in this paper. Most NIST proposals use Method 0, however, the performance penalty paid for

Table 6: Qualitative comparison of the different $\boldsymbol{A}$ generation methods. With respect to a design goal or criteria, ++ denotes the most desired comparative performance demonstrated by a method, followed by + and -, with - - denoting the least.

Design goal		**Method**			
		0	**1**	**2**	**3**
Formal proofs of security		+	-	-	+
Avoidance of structure in the lattice		++	++	+	++
Backdoor attack resistance		++	++	++	++
Pre-computation attack resistance		++	++	++	- -
CPU requirements	Client	- -	++	++	++
	Server	- -	++	++	++
Memory requirements	Client	++	+	++	++
	Server	++	++	++	++

the generation of $\boldsymbol{A}$ is high in unstructured lattices, especially if the DRBG used is not optimized in hardware. Methods 1 and 2 provide a significant performance benefit delivering much faster performance. For instance, NIST level 1 configuration of Round5 R5N1_1PKE_0d using Method 2 with SHAKE128 is 3x (resp. 7x) faster than the optimized implementation Frodo640 using Method 0 with AES128-specific hardware instructions (resp. AVX2-optimized SHAKE128). Furthermore, these methods remove (anti-) circulant-like structure in the underlying lattice and also prevent pre-computation and backdoor attacks.

The choice between Method 1 and Method 2 depends on the usage scenario. Method 1 is good for configurations in which a server has to handle many connections; Method 2 allows for a smaller memory footprint, and thus, it is more suitable if clients are constrained in their storage capabilities, i.e., have more stringent memory requirements. The main issue is that Methods 1 and 2 lack an distinguishability proof and in particular, Method 2, keeps some structure. Still, the performance benefit offered by Methods 1 and 2 motivated their submission in Round2, the default usage of Method 2 in the unstructured Round5 parameter sets, and definitively encourages further cryptanalysis of these constructions.

An important factor to consider with regard to Method 0 is the possibility of efficient attacks against the DBRG used to expand the seed into the matrix $\boldsymbol{A}$ [7]. Regarding Method 3, the difficulty of verifying that entries of $\boldsymbol{A}$ are indeed sampled at random should be taken into account. Furthermore, the risk of pre-computation attacks in Method 3 seems to be too high compared to the potential performance benefits.

References

1. Erdem Alkim, Roberto Avanzi, Joppe W. Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, and Douglas Stebila.

NewHope, 2019. Version 1.02; Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.
2. Erdem Alkim, Joppe W. Bos, Léo Ducas, Patrick Longa, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Chris Peikert, Ananth Raghunathan, and Douglas Stebila. FrodoKEM: Learning With Errors Key Encapsulation, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.
3. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - a new hope. In *USENIX Security Symposium*, pages 327–343, 2016.
4. Roberto Avanzi, Joppe W. Bos, Léo Ducas, Eike Kiltz, Trancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber, Algorithmic Specification And Supporting Documentation (version 2.0), 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.
5. Hayo Baan, Sauvik Bhattacharya, Scott Fluhrer, Oscar Garcia-Morchon, Thijs Laarhoven, Rachel Player, Ronald Rietman, Markku-Juhani O. Saarinen, Ludo Tolhuizen, Jose-Luis Torre-Arce, and Zhenfei Zhang. Round5: KEM and PKE based on (Ring) Learning with Rounding, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.
6. Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *EUROCRYPT*, pages 719–737, 2011.
7. Daniel J. Bernstein. Comparing proofs of security for lattice-based encryption, 2019. Available at https://cr.yp.to/papers/latticeproofs-20190608.pdf.
8. Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU Prime: round 2, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.
9. Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *IEEE S&P*, pages 553–570, 2015.
10. Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. NTRU, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.
11. Ronald Cramer, Léo Ducas, Chris Peikert, and Oded Regev. Recovering short generators of principal ideals in cyclotomic rings. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT (II)*, pages 559–585, 2016.
12. Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short Stickelberger class relations and application to ideal-SVP. In *EUROCRYPT*, pages 324–348, 2016.
13. Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. SABER: Mod-LWR based KEM (Round 2 Submission), 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.
14. Mike Hamburg. Three Bears, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.
15. Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, pages 319–339, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. https://eprint.iacr.org/2010/613.
16. Xianhui Lu, Yamin Liu, Dingding Jia, Haiyang Xue, Jingnan He, Zhenfei Zhang, Zhe Liu, Hao Yang, Bao Li, and Kupeng Wang. LAC:

Lattice-based Cryptosystems, 2019. Available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.
17. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, pages 1–23, 2010.
18. NIST. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. POST-QUANTUM CRYPTO STANDARDIZATION. Call For Proposals Announcement, 2016.
19. Chris Peikert. Lattice cryptography for the internet. In *PQCrypto*, pages 197–219, 2014.
20. Alice Pellet-Mary, Guillaume Hanrot, and Damien Stehlé. Approx-SVP in ideal lattices with pre-processing. In *EUROCRYPT*, 2019.
21. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC*, pages 84–93, 2005.