Towards practical GGM-based PRF from (Module-)Learning-with-Rounding

Chitchanok Chuengsatiansup[1] and Damien Stehlé[2]

[1] Inria and ENS de Lyon, France
[2] ENS de Lyon, Laboratoire LIP (U. Lyon, CNRS, ENSL, INRIA, UCBL), France

Abstract. We investigate the efficiency of a (module-)LWR-based PRF built using the GGM design. Our construction enjoys the security proof of the GGM construction and the (module-)LWR hardness assumption which is believed to be post-quantum secure. We propose GGM-based PRFs from PRGs with larger ratio of output to input. This reduces the number of PRG invocations which improves the PRF performance and reduces the security loss in the GGM security reduction. Our construction bridges the gap between practical and provably secure PRFs. We demonstrate the efficiency of our construction by providing parameters achieving at least 128-bit post-quantum security and optimized implementations utilizing AVX2 vector instructions. Our PRF requires, on average, only 39.4 cycles per output byte.

Keywords: pseudorandom function, (module-)learning-with-rounding, post-quantum security, efficient implementation, Karatsuba multiplication

1 Introduction

A pseudorandom function (PRF) is a keyed function whose outputs are pseudorandom, i.e., no probabilistic polynomial-time adversary can distinguish, with non-negligible advantage, between the outputs of the pseudorandom function for chosen inputs and those of a truly random function. The pseudorandom function should also be efficiently and deterministically computable.

Pseudorandom functions are essential tools in designing protocols. PRFs are used as building blocks in many cryptographic primitives especially in symmetric-key cryptography, for example, message authentication codes and block ciphers.

Banerjee, Peikert and Rosen (BPR) [7] proposed theoretical provably-secure constructions of pseudorandom functions based on conjectured hard learning-with-error lattice problems and "rounded-subset products". They also introduced a "derandomization" technique in order to generate the error terms deterministically. Despite interesting proofs and new techniques, the main drawback of their constructions is that the parameters are too large for practicality.

This work was supported in part by BPI-France in the context of the national project RISQ (P141580), by the European Union PROMETHEUS project (Horizon 2020 Research and Innovation Program, grant 780701).

Later on, Banerjee, Brenner, Leurent, Peikert and Rosen [6] proposed concrete and practical instantiations of PRFs called "SPRING" (for subset-product with rounding over a ring), which are based on the BPR design [7]. However, SPRING instantiations [6] do not adhere to the parameters suggested by the security proof of the BPR design, namely, they relax the condition requiring the modulus q to be exponentially large in the PRF input length ℓ. More precisely, they use a very small modulus q to obtain good performance.

Even though it has been stated in [6] (and also referred to the original comments in BPR [7]) that the exponentially large q might be a proof artifact, SPRING instantiations do not inherit the security guarantees from BPR [7]. Those instantiations can be viewed as heuristic. If SPRING [6] had set parameters according to the proofs in [7], it would have resulted in a huge performance penalty.

In this work, we investigate the practicality of the pseudorandom function construction of Goldreich, Goldwasser, and Micali (GGM) [22] and show that the GGM-based PRF can lead to a reasonably efficient scheme enjoying a proof under a reasonable assumption. Note that the GGM construction is elementary; it can be constructed from any length-doubling pseudorandom generator (PRG) and can be depicted as a binary tree. However, there is a significant disadvantage with this construction, namely, it requires ℓ *sequential* invocations to the underlying PRG for input length ℓ which corresponds to the number of levels in the binary tree. The reason why it is not possible to parallelize between different level of the tree is that one needs the output of the PRG at level i to be used as the input of the PRG at level $i+1$. For many applications (in particular when used as building components of advanced cryptographic primitives), this can be a significant drawback. For this reason, the BPR construction and follow-up works do not rely on the GGM framework.

Even though the sequential invocation can be viewed as a crucial disadvantage, we can use this construction in applications which evaluate PRF using consecutive inputs such as in a counter mode. In such a case, the average number of invocations to the underlying PRG is approximately only 2 (as opposed to ℓ in the worst case). The reason why the number of invocations can be very low is that we do not always need to traverse the tree from the root to one of the leaves; we may backtrack only one level up. Recall the GGM construction: we use only half of the PRG's output for the next input. If the current evaluation uses the left half, then the next consecutive evaluation would use the right half which has already been computed, thus involving no PRG invocation at all.

To construct an efficient and post-quantum secure PRG, we rely on the hardness assumption of the (module-)learning-with-rounding ((module-)LWR) problem. We demonstrate the practicality of our lattice-based PRF by deriving parameters achieving at least 128-bit security and providing optimized implementations together with timing results.

1.1 Tools and techniques

Our intention is to construct an efficient post-quantum secure PRF based on a provably secure construction and broadly studied conjectures which are believed to be hard problems. The main tools we used are the GGM construction and the hardness assumption of the (module-)LWR problem. Other tools that we used to obtain fast implementations are Karatsuba's multiplication and the vector instructions (AVX2). In the following, we briefly explain how we utilized and combined these tools.

GGM construction. One of the main reasons that we chose the GGM construction is its simplicity: the construction requires only a length-doubling PRG and can be depicted using a binary tree. Moreover the GGM construction has a rigorous security proof, namely, so long as the underlying PRG is secure, the GGM-based PRF construction is secure. This means that we can restrict to building a secure length-doubling PRG.

We further observed that the *binary* tree in the GGM construction corresponds to the length-*doubling* requirement for the PRG. However, there is no restriction why we should use a *binary* tree. For example, we could use a *ternary* tree and use a length-*tripling* PRG. We could even push further and use a *4-ary* tree with a length-*quadrupling* PRG. In this work, we generalize to an ω-*ary* tree with a length-ω-fold PRG. Using higher expansion rate is the main technique in our construction to obtain a lower-depth tree, fewer number of calls to the underlying PRG, and improvements in the efficiency.

(M)LWR. Note that the security of the GGM-based PRFs relies on the security of the underlying PRG. Therefore, the next question that we need to answer is what to use for the secure expansion-rate-ω PRG. Note also that computing this PRG should be efficient because the performance of the PRF heavily depends on the performance of the underlying PRG.

Since we aim at constructing a post-quantum secure PRF, lattice-based constructions seem to be the most appealing especially in terms of efficiency. For this reason, we decided to rely on the (module-)LWR hardness assumption. Informally, the small-secret variant of the (module-)LWR problem states that multiplying a uniformly random matrix $\mathbf{A}$ with a "small" uniformly random vector $\mathbf{s}$ then performing rounding operations leads to a resulting vector that looks uniformly random, i.e., indistinguishable, with non-negligible probability, from a truly uniform vector.

Note that there also exist other variants such as ring-LWR, learning-with-errors (LWE), ring-LWE, and module-LWE. The main reason that we chose to focus on (module-)LWR variants instead of the ring variants is the flexibility in the parameter setting. Note that with the plain LWR, it allows even better adjustment of parameters and involves no polynomial reduction. However, one drawback of the plain LWR is that it requires rather huge memory for the public parameter, i.e., $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$. For the completeness, we include the plain LWR variant in Appendix A.

We could have decided to construct the PRG relying on the hardness assumption of the (module-)LWE problem (instead of the (module-)LWR one). However, our analysis suggested that using learning-with-rounding provides more efficiency in terms of parameter selection and implementation. Since the input to the PRG (in our context) is the secret vector $\mathbf{s}$, if we were using (module-)LWE we would also need the error vector $\mathbf{e}$ as part of the input. This means that the output of the matrix-vector multiplication $\mathbf{As}$ plus an error vector $\mathbf{e}$ needs to provide enough bits to be used as the next input. In case we use (module-)LWR (which we do), we only need that the output has enough bits for the secret vector $\mathbf{s}$. We do not need the error vector $\mathbf{e}$ because we use deterministic rounding instead. This also means that we can set our parameters slightly smaller. Moreover, the rounding operation is simply cutting bits; this operation is arguably easier to implement than generating and adding errors.

Karatsuba's multiplication. Our construction with module-LWR consists of matrix-vector multiplication and rounding operations. Since rounding is simply ignoring bits and can be done relatively fast, the bottleneck operation is the matrix-vector multiplication which is basically polynomial multiplication. Since we also aim at having efficient implementations, we wish to use a polynomial multiplication algorithm which gives better time complexity than that of the schoolbook multiplication. One of commonly used multiplications is the number-theoretic transform (NTT). However, this multiplication requires an NTT-friendly prime which we view as quite a strong restriction. Therefore, we opt for more flexible multiplication algorithms such as Karatsuba and Toom-Cook.

The reason we choose Karatsuba's multiplication is that it outperforms Toom-Cook's multiplication for low to medium degree polynomials. This suggests that we should use Karatsuba's multiplication up to certain degrees and then switch to using Toom-Cook's multiplication. In other words, Toom-Cook should be used in combination with Karatsuba, but we may not need to switch to using Toom-Cook. To keep our implementations simple and flexible, we decided to ignore Toom-Cook and use only Karatsuba's multiplication.

Table 1 compares estimated multiplication costs of our instantiations, where we use Karatsuba's multiplication, with (extrapolations of) previous work, where we assume using NTT for polynomial multiplication. In this cost estimation for our Karatsuba's multiplication, we decompose polynomial degree n down to 16 then we switch to the schoolbook method for the actual multiplication.

1.2 Optimized implementations

We analyzed the parameters of the (module-)LWR problem together with the expansion rate ω. We derived parameters achieving at least 128-bit security in the quantum core model [3]. We provide optimized implementations [3] of our PRF construction in which we also use vector instructions to obtain better performance. The timing results (see Section 5 for more details) show that evaluating

[3] available at https://github.com/BeJade/mlwr-prf

PRF in a counter mode using our construction requires, on average, only 39.4 cycles per output byte.

Table 1: Extrapolated cost estimation of lattice-based PRFs with different underlying hardness assumptions for 128-bit security

schemes	assumptions	mod	key	pp.	mul
SPRING-BCH [6]	R $\lfloor$sub prod$\rceil$	257	16396	129	115
SPRING-CRT [6]	R $\lfloor$sub prod$\rceil$	514	18444	145	115
Our scheme, M	MLWR, GGM	2^{16}	192	24576	1991
Our scheme	LWR, GGM	2^{16}	200	2944000	2944
[7] + AKPW13,R [5]	RLWR, GGM	$> 2^{40}$	655360	5120	5243*
BPR12,syn,R [7]	RLWR, syn	$> 2^{128}$	163840	-	10486*
BPR12,direct,R [7]	R $\lfloor$sub prod$\rceil$	$> 2^{128}$	2097152	16384	10486*
[7] + AKPW13 [5]	LWR, GGM	$> 2^{40}$	313600000	3500	250880*
BPR12,direct [7]	$\lfloor$sub prod$\rceil$	$> 2^{128}$	1003520000	11200	501760*
BPR12,GGM [7]	LWR, GGM	$> 2^{128}$	11200	31360000	1003520*
BPR12,GGM,R [7]	RLWR, GGM	$> 2^{128}$	16384	67108864	42949673*
BPR12,syn [7]	LWR, syn	$> 2^{128}$	74097496	-	351232000*

In the first two columns, abbreviations 'syn', 'R', 'M', '$\lfloor$sub prod$\rceil$', 'MLWR' mean 'synthesizer', 'ring', 'module', 'rounded subset product' and 'module-LWR' respectively. Key and public parameter (pp.) are of size in bytes. Multiplication (mul) in the last column considers only the main matrix-matrix or matrix-vector product; the cost is shown in terms of thousand 16-bit-by-16-bit multiplication; asterisk ($*$) means extrapolated cost. Since we focus on constructions which provide security level at least 2^{128}, we extrapolate the cost of BPR-based schemes using dimension $n = 1024$ for RLWR (to allow NTT) and $n = 700$ for LWR. Note that in this comparison table we provide BPR-based schemes two advantages: (1) we consider modulus $q = 2^{128}$ (resp. 2^{40}) while it should be much more than these values for the proofs to be applicable; (2) for the synthesizer variant, we consider that all moduli are the same and take the smallest one. We do not include [12] in the comparison because it is a PRG not a PRF.

2 Preliminaries

We write vectors in bold lowercase letters and matrices in bold uppercase letters. We write log for $\log_2$ unless specified otherwise. In the context of distributions, we denote U a uniform distribution. We define the rounding function $\lfloor \cdot \rceil_p : \mathbb{Z}_q \longrightarrow \mathbb{Z}_p$ as $\lfloor x \rceil_p = \lfloor x \cdot (p/q) \rceil$ where $q > p \in \mathbb{Z}$. When applied to vectors, it means applied componentwise.

2.1 GGM construction

Goldreich, Goldwasser and Micali [22] proposed a method to construct a PRF from any length-doubling pseudorandom generator, i.e., expansion rate 2. To

describe their construction, let G be a length-doubling pseudorandom generator where G_0 and G_1 denote the first half and the second half of G's output respectively, i.e, $G(k) = G_0(k)||G_1(k)$ where $|G_0(k)| = |G_1(k)| = \ell$. PRF with a key k and ℓ-bit input x, $F_k : \{0,1\}^\ell \to \{0,1\}^\ell$ is defined as:

$$F_k(x_1 x_2 \cdots x_\ell) = G_{x_\ell}(\cdots(G_{x_2}(G_{x_1}(k)))\cdots).$$

This construction can be depicted as a binary tree whose depth corresponds to the input length ℓ. The root of the tree gets the input key k as an input for the PRG evaluation whereas each left-child node gets G_0 from its parent to use as an input and each right-child node gets G_1. This process continues as many times as the input length. Evaluating PRF with an input x is done by looking at each bit of x and traversing the tree from the root to one of the leaves by going down to the left subtree if the bit is 0 and going down to the right subtree if the bit is 1. Note that we do not have to compute and store the entire binary tree; we only need to compute PRG along the path from the root to the corresponding leaf. Figure 1 depicts the above construction for $\ell = 3$. Solid lines show the path to compute $F_k(010)$.

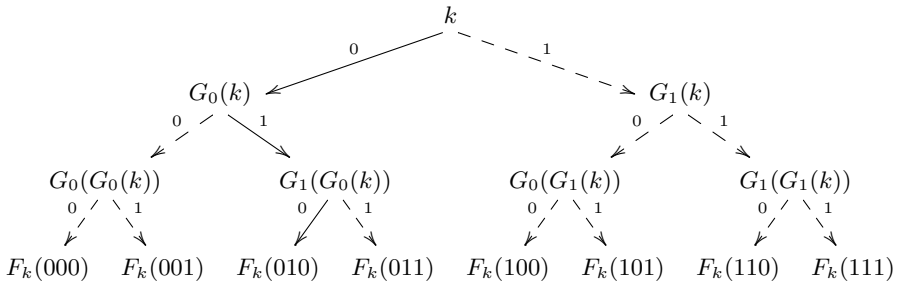

Fig. 1: GGM construction

Theorem 1 (GGM construction, adapted from [22,25]). *If G is a length-doubling ϵ-indistinguishable pseudorandom generator, then the construction outlined above is a $qd\epsilon$-indistinguishable pseudorandom function where q denotes the number of queries and d denotes the tree's depth.*

2.2 Hardness assumptions.

The security of cryptographic primitives usually relates to hardness assumptions that certain mathematical problems are difficult to solves. In this subsection, we recall definitions of hardness assumptions that we use in our constructions.

Definition 1 (Learning-with-errors (LWE)). *Let $q \geq 2$ be a modulus, m and n be dimensions such that $m \geq n \geq 1$, and χ be a distribution over $\mathbb{Z}$.*

The learning-with-errors *problem is to distinguish, with non-negligible probability, between the following two distributions:* $(\mathbf{A}, \mathbf{As} + \mathbf{e})$ *and* $U(\mathbb{Z}_q^{m \times n} \times \mathbb{Z}_q^m)$*, where* $\mathbf{A} \sim U(\mathbb{Z}_q^{m \times n})$*,* $\mathbf{s} \sim U(\mathbb{Z}_q^n)$ *and* $\mathbf{e} \sim \chi^m$*.*

The LWE assumption states that the LWE problem for certain choices of parameters (q, m, n, χ) is hard for all probabilistic polynomial time algorithms. The LWE problem has been extended to ring-LWE by Lyubashevsky, Peikert and Regev [34] where the integer ring $\mathbb{Z}$ has been replaced by a polynomial ring $\mathcal{R}$. Since LWE and ring-LWE are very similar except for using different rings and vector dimensions, these problems were later on generalized by Brakerski, Gentry and Vaikuntanathan [13] to what they called a *general learning with errors problem*. Langlois and Stehlé [32] analyzed the hardness and explained the usefulness of this variant where they called it module-LWE. In this paper, we also use the term *module*.

Definition 2 (Module-LWE, adapted from [13]). *Let* m *and* k *be integer dimensions, let modulus* $q \geq 2$ *be a prime integer, let* $\mathcal{R} = \mathbb{Z}[x]/(x^d + 1)$ *where* d *is a power of two, let* $\mathcal{R}_q = R/qR$*, and let* χ *be a distribution over* $\mathcal{R}$*. The* module-LWE *problem is to distinguish, with non-negligible probability, between the following two distributions:* $(\mathbf{A}, \mathbf{As} + \mathbf{e})$ *and* $U(\mathcal{R}_q^{m \times k} \times \mathcal{R}_q^m)$*, where* $\mathbf{A} \sim U(\mathcal{R}_q^{m \times k})$*,* $\mathbf{s} \sim U(\mathcal{R}_q^k)$ *and* $\mathbf{e} \sim \chi^m$*.*

Banerjee, Peikert and Rosen [7] introduce the learning-with-rounding (LWR) problem whose error terms in the LWE problem are replaced by a deterministic rounding process. With some conditions on the modulus and the dimensions, they also proved that the LWR problem is as hard as the LWE problem. There also exist ring and module versions of LWR where the problems are defined analogously.

Definition 3 (Module-LWR, adapted from [7,13,32]). *Let* m *and* k *be integer dimensions, let moduli* $q \geq p \geq 2$ *be integers, let* $\mathcal{R} = \mathbb{Z}[x]/(x^d + 1)$ *where* d *is a power of two, and let* $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$*. The* module-LWR *problem is to distinguish, with non-negligible probability, between the following two distributions:* $(\mathbf{A}, \lfloor \mathbf{As} \rfloor_p)$ *and* $U(\mathcal{R}_q^{m \times k}) \times \lfloor U(\mathcal{R}_q^m) \rfloor_p$*, where* $\mathbf{A} \sim U(\mathcal{R}_q^{m \times k})$ *and* $\mathbf{s} \sim U(\mathcal{R}_q^k)$*.*

Note that the distribution $\lfloor U(\mathcal{R}_q^m) \rfloor_p$ may seem inconvenient. However, by taking $p|q$, this distribution becomes identical to $U(\mathcal{R}_p^m)$ which is easier to work with.

We would like to make a remark that there also exist different (module-)LWR variants depending on the distributions of the secret $\mathbf{s}$ and the round-off bits. The variant that we use is the one with small secret where the distribution of the secret $\mathbf{s}$ is consistent with the round-off bits.

3 Our construction

We aim at constructing an efficient and secure lattice-based PRF. Our construction is based on two main building blocks, namely, the GGM construction and

the (module-)learning-with-rounding problem [7]. The former is used for constructing pseudorandom functions while the latter is used for the underlying hardness assumption of pseudorandom generators.

This section explains how we combine the aforementioned two building blocks to achieve a competitive performance PRF. We first start by explaining how we cope with the sequential nature of the GGM construction. Then we explain choices for the PRG and give justifications why we decided to work with the (module-)LWR. Finally, we give the outline of our construction.

3.1 Lower-depth GGM-based PRF

Recall that the GGM-based pseudorandom function construction requires a length-doubling pseudorandom generator, and the construction can be depicted using a binary tree. Each node (except for the leaves) of the tree corresponds to an evaluation of a PRG whose input is from half of its parent's PRG output, i.e., front half for a left-child node, and back half for a right-child node. For the root of the tree, its input is the key k of the PRF.

Evaluating the GGM-based PRF is done by looking at each bit of the input x and traversing the tree from the root to one of the leaves by, say, if the current bit is 0 then go to the left subtree, if the current bit is 1 then go to the right subtree. One thing to be noticed is that the depth of the tree corresponds to the length of the PRF input, i.e., an ℓ-bit-input PRF implies a binary tree of depth ℓ.

Observe that the performance of the GGM construction heavily relies on the number of calls to the PRG which corresponds to the input length ℓ and, thus, the depth of the *binary* tree. The ℓ-sequential call is a major drawback of this construction, and it is unavoidable due to the nature of the construction, namely, one needs to traverse the tree level by level since the output from the previous level is used as the input to the next level. Our goal is to address this drawback.

Our approach is to generalize to using an *ω-ary tree* instead of a binary tree. This affects the original construction in three ways. First, we do not merely look at a single bit when we traverse the tree; instead, we examine $\log \omega$ bits at a time. Second, the tree's depth decreases to $\ell / \log \omega$ levels, meaning that the number of invocations to the underlying PRG also decreases to $\ell / \log \omega$ calls. By reducing the depth of the tree, this also reduces the security loss in the security reduction of the security proof of the GGM construction. Third, the construction now requires the output of the PRG to be ω times longer than the input length (not just length-doubling as in the original construction). Note however that this should not be viewed as a severe penalty because it is achievable in practice; we give a list of possible parameters, explain a concrete implementation and show timing results in Section 5.

In general, we may set ω to be any positive integer greater than 1, and if we set $\omega = 2$ then we obtain the original GGM construction. The construction works best when ω is a power of two, i.e., when $\log \omega$ is an integer. Note that we do not impose on a condition that ℓ is divisible by $\log \omega$. If ℓ is divisible by $\log \omega$, then it results in a complete ω-ary tree and every call to PRG has a full

ω choices of which part of the output to be used as the next input. Otherwise, the tree is not complete where nodes at the level before the leaves have about $2^{\ell}/\omega^{\lceil \ell/\log\omega \rceil - 1}$ child nodes.

Even though our construction does not remove the nature of the sequential invocations to the underlying PRG, we significantly decrease the number of calls (the tree's depth). That is, instead of requiring ℓ sequential calls for length ℓ input, our construction requires only $\ell/\log\omega$ calls. Notice that the asymptotic bound remains the same for the original GGM and our construction. However, our construction, indeed, helps improve the performance in practice (see Section 5 for more details on concrete implementations and timing results) since the number of calls to PRG decreases at least by a factor of 1.5.

3.2 (M)LWR-based PRG

Another component that we need in our construction is a secure PRG with the output length ω times larger than its input length. Since we target an efficient and post-quantum secure PRF construction, one natural candidate is the lattice-based cryptography. Therefore, we consider constructing PRG based on the learning-with-errors (LWE) [43] and the learning-with-rounding (LWR) [7] problems.

Recall that the LWE problem involves computing $(\mathbf{A}, \mathbf{b} = \langle \mathbf{A}, \mathbf{s} \rangle + \mathbf{e})$ where $\mathbf{A}$ is public, $\mathbf{s}$ is secret, and $\mathbf{e}$ is small error generated randomly. In case of the LWR problem, the process of adding small error $\mathbf{e}$ is replaced by deterministic rounding. The LWR problem was introduced in [7] where the authors also introduced a lattice-based PRF whose construction is based on a pseudorandom synthesizer [37,38,39] which requires a deterministic function. This requirement is opposite to the nature of LWE where errors are generated randomly. For this reason, LWR was used in [7]. In case of our construction, both LWE and LWR are possible. We use LWR for efficiency, i.e., to decrease the amount of randomness required (see below and Section 5). Therefore, we want to emphasize that the reason we use LWR slightly differs from that of [7].

To reduce the memory required for the secret vector $\mathbf{s} \in \mathbb{Z}_q^n$ and the public matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$, we consider using module-LWR [13,32] instead of LWR, i.e., using $\mathbf{s} \in \mathcal{R}_{q'}^{n'}$ and $\mathbf{A} \in \mathcal{R}_q^{m' \times n'}$. We could have used ring-LWR [35], but we chose to work with modules because of the flexibility. As explained in [11] and [18]; there is only one ring to implement, and adjusting security is done by adjusting the dimension of the matrix. We explain our construction using LWR in Appendix A. In this section, we focus on our construction using module-LWR.

Until now, we have not yet mentioned how to obtain the length-ω-fold-output PRG from module-LWR. We recall our module-LWR-based construction then we explain how to satisfy the required length condition. In our construction, except for the initial input for the first call to the PRG, the subsequent calls use the output from previous calls. The output length condition requires that the number of bits of $\lfloor \mathbf{As} \rceil_p$ must be at least ω times larger than $\mathbf{s}$ which is the input for the next call to the module-LWR-based PRG. We claim that this requirement is indeed achievable.

To see this, recall that the expansion rate r is the ratio between the output and the input length of the PRG (e.g., length-doubling PRG has $r = 2$) and we want $r \geq \omega$ for our construction. Let $q \geq p \geq 2$ be integers, let $\mathcal{R} = (\mathbb{Z}[x]/(x^d + 1))$, let $\mathbf{A} \in \mathcal{R}_q^{m \times n}$, and let $\mathbf{s} \in \mathcal{R}^n_{\log q - \log p}$ (i.e., $q' = \log q - \log p$). Note that each coefficient of $\mathbf{s}$ has the same magnitude as the round-off bits. The input $\mathbf{s}$ has size $nd(\log q - \log p)$, and the output $\lfloor \mathbf{As} \rfloor_p$ has size $md \log p$. Thus, we need to ensure that $\frac{md \log p}{nd(\log q - \log p)} \geq r$. One concrete example (which we also use in our implementation, see Section 5) to achieve $r \geq 16$ is to set parameters as follows: $q{=}2^{16}, d{=}256, n{=}3, m{=}16$ and $p{=}2^{12}$. This gives $\frac{16 \cdot 256 \cdot \log 2^{12}}{3 \cdot 256(\log 2^{16} - \log 2^{12})} \geq 16 = r$. We refer to Table 2 and Table 4 for other possible parameters.

3.3 Construction outline

In this subsection, we explain our efficient lattice-based PRF (see Appendix A for the LWR variant). Recall that the input to the PRF is a key k and a string x of length ℓ while the output is a pseudorandom string z. A public parameter in our construction is $\mathbf{A} \in \mathcal{R}_q^{m \times n}$.

Since our construction is more efficient when $\log \omega = \tilde{\omega}$ is an integer, we assume from now on that it is the case. Construction 1 outlines our PRF. We divide the construction into three parts, namely, pre-computation, main loop, and outputting. In the following, we explain each part in more details.

Construction 1 PRF from module-LWR

Input: PRF's key $k = \{0,1\}^*$ and a bit string $x = \{0,1\}^\ell$
Output: Pseudorandom string $z = \{0,1\}^{\ell'}$

1: $\mathbf{t} = \mathsf{Random}(k)$
2: $\mathbf{s} = \mathsf{Reformat}(\mathbf{t})$ } Pre-computation
3: **for** $i = 0$ **to** $\ell - 1$ **step by** $\tilde{\omega}$ **do**
4: $\quad c = x[\tilde{\omega}i] \;\; x[\tilde{\omega}i{+}1] \;\ldots\; x[\tilde{\omega}i{+}\tilde{\omega}{-}1]$ ▷ View $\tilde{\omega}$ bits as number in $[0, \omega{-}1]$
5: $\quad \mathbf{t} = \mathbf{A}_c \cdot \mathbf{s}$ ▷ Multiply row c of $\mathbf{A}$ to $\mathbf{s}$
6: $\quad \mathbf{t} = \lfloor \mathbf{t} \rfloor$
7: $\quad \mathbf{s} = \mathsf{Reformat}(\mathbf{t})$ ▷ Reformat into $\mathbf{s}$
8: **end for**
9: **for** $i = 0$ **to** $m - 1$ **do**
10: $\quad \mathbf{t} = \mathbf{A}_i \cdot \mathbf{s}$
11: $\quad \mathbf{t} = \lfloor \mathbf{t} \rfloor$
12: $\quad z = z || \mathbf{t}$ ▷ Concatenate output extracting from $\mathbf{t}$
13: **end for**
14: **return** z

Pre-computation. This part handles (1) generating randomness from the key k, i.e., $\mathsf{Random}(k)$, for example, using extendable output functions, and (2) reformatting into the vector $\mathbf{s}$, i.e., $\mathsf{Reformat}(\mathbf{t})$, for example, setting each 4 bits of $\mathbf{t}$

to each coefficient of $\mathbf{s}$. Note that these can be pre-computed per secret key k. For instance, evaluating the same function $F_k(\cdot)$ at different points, say, x and y, e.g., $F_k(x)$ and $F_k(y)$. Note that we could reformat k directly into $\mathbf{s}$ without feeding it as an input to generate random bits, i.e., calling $\mathsf{Reformat}(\mathbf{t})$ directly without calling $\mathsf{Random}(k)$. However, we have $\mathsf{Random}(k)$ for two reasons: first, in case the input k is not random, which would affect the security analysis of the underlying module-LWR problem; second, in case the input k does not have the preferred length, which would be problematic if it is too short. If the input k is already random and has long enough length, then $\mathsf{Random}(k)$ can be skipped. Nevertheless, we still perform $\mathsf{Reformat}(\mathbf{t})$ to convert random bits into a right format for the following steps.

Main loop. This part handles the main loop which involves calling the underlying module-LWR-based PRG. The number of calls depends on the width of the tree (ω) which is one of several parameters to be optimized. In the construction outline, we present a general construction that is not specific to any particular width. In Section 5, we give a concrete example where we define all parameters including the tree's width ω.

One optimization in our construction is that each iteration involves a vector-vector multiplication and not a matrix-vector multiplication. We observed that we only need m/ω rows of the matrix $\mathbf{A}$ to multiply with the vector $\mathbf{s}$. This value m/ω can be as small as one, which is the case for our concrete implementation. Therefore, we assume that we select only one row of the matrix $\mathbf{A}$. Choosing which row is done by viewing $\tilde{\omega}$ bits of the input x as a number between 0 and $\omega - 1$ (line 4 in Construction 1) then selecting the corresponding row c (line 5 in Construction 1).

After having multiplied the selected row of the matrix $\mathbf{A}$ with the vector $\mathbf{s}$, before moving to the next $\tilde{\omega}$ bits of ℓ, we call $\mathsf{Reformat}(\mathbf{t})$ to extract bits from $\mathbf{t}$ and assign to $\mathbf{s}$ to be used in the following iteration. These steps are analogous to selecting front half ($G_0(\cdot)$) or back half ($G_1(\cdot)$) of the PRG for the next PRG's call in the original GGM construction.

Outputting. This part handles the output of the PRF. Before we reformat the output, we perform the final m iterations of vector-vector multiplications. In contrast to previous iterations where we multiply only m/ω rows of $\mathbf{A}$ with $\mathbf{s}$, for the output we multiply all m rows, i.e., the entire matrix $\mathbf{A}$. This has the advantage of significantly increasing the length of the PRF output. The final task of this part is to concatenate all the results and turn them into the desire format.

4 Security analysis

The high-level structure of our PRF construction follows the GGM construction whose security relies on the security of the underlying PRG. Therefore, we mainly analyze the security of the (module-)LWR problem which we use as the expansion-rate-ω PRG in our lattice-based PRF construction. Note that in our generalized

GGM construction, it is possible that the tree's depth is lower than in the original one. For this reason, we also examine if this may incur any loss in the security reduction of the GGM construction.

4.1 GGM

Since our construction is essentially the generalization of the GGM construction, Theorem 1 also applies to our setting. Note that in the original GGM construction, the tree's depth corresponds to the bitlength ℓ of the input string to the PRF. In our generalized GGM construction, the tree's depth corresponds to $\ell/\tilde{\omega}$ where $2^{\tilde{\omega}}$ is the tree's width. If $\tilde{\omega} = 1$ then we recover the original GGM construction. This means that the tree's depth in the generalized GGM construction is no greater than in the original construction.

4.2 LWE and (M)LWR

Our construction relies on the hardness assumption of the (module-)LWR problem where each sample is a vector whose elements are from a polynomial ring (or an integer ring in case of the plain LWR variant). It can be seen that the module-LWR problem provides more structure than LWR. However, since there are no known attacks that exploit such additional structures (of either ring or module), we did not make use of the module structure in our security analysis.

In fact, we analyzed the hardness of the (module-)LWR problem as the LWE one where we consider the process of adding errors as being replaced by the process of rounding off bits. A reduction from the LWE to the LWR problem was first given by Banerjee, Peikert and Rosen [7], then further improved by Alwen, Krenn, Pietrzak and Wichs [5], Bogdanov, Guo, Masny, Richelson and Rosen [10], and Alperin-Sheriff and Apon [4]. We refer to those articles for more details.

To solve the LWE problem (see [3] for more details), the state-of-the-art algorithm is BKZ [44,16] and there are two variants, namely, the primal and the dual attacks. We estimate security for both variants and take the minimum between the two. The BKZ algorithm reduces a lattice basis by calling an SVP oracle with a smaller dimension b where there are two different approaches, namely, enumeration and sieving. We explore both approaches and take the minimum between the two. We estimate the complexity of solving SVP by using $2^{0.292b}$ as the classical and $2^{0.265b}$ as the quantum cost estimations [40,36,29,8,30,28,11]. We use Kyber's script [11] as a main tool to perform security analysis where our notion of bit security reflects security of the underlying LWE problem.

5 Implementation

To illustrate the practicality of our construction, we implemented two variants of our lower-depth GGM-based PRF, one with module-LWR and another one with LWR. In this section, we focus on the module-LWR variant and explain how

we designed our implementation, derived parameters achieving at least 128-bit security, and optimized our implementation. (The script to estimate the security level and our implementation are publicly available at https://github.com/BeJade/mlwr-prf.) We also show the timing results of the two variants of our implementations and compared with previous work. Note that details on the LWR variant can be found in Appendix A.

5.1 Design

The goal of this subsection is to describe how we chose to implement each component in our construction. We give justifications of choices made and explain why we think they best suit our optimized implementation.

Polynomial Multiplication. In addition to the fundamental schoolbook multiplication, there exist other (polynomial) multiplication algorithms which achieve faster time complexity, for example, NTT, Toom-Cook, Karatsuba. The main reason that we cannot use NTT is that it requires modulus q to be of the form $q \equiv 1 \bmod 2n$ where n is the degree of the polynomial. With this condition, it rules out even moduli q.

We could have used Toom-Cook (to split degree-n polynomial into k degree-n/k polynomials) which can be considered as a generalization of Karatsuba ($k = 2$, i.e., splitting degree-n polynomial into 2 degree-$n/2$ polynomials). However, our analysis shows that Karatsuba's algorithm works better for low to medium degree polynomials where the switching point between Karatsuba and Toom-Cook is unclear. To keep our implementation simple and reusable in case of updating parameters, we decided to use Karatsuba's algorithm.

To multiply degree-n polynomials f and g, we start by splitting f into f_0 and f_1 each of degree $n/2$ (similarly for g). We repeat this splitting until we reach polynomials of degree 16, then we switch to using the schoolbook multiplication. We chose the cut-off at degree 16 because AVX2 instructions allow 16-way vectorization.

Karatsuba's algorithm is very well suited for vectorization. To see this, recall (refined) Karatsuba's identity [9]:

$$(f_0 + x^{n/2} f_1)(g_0 + x^{n/2} g_1) = (1 - x^{n/2})(f_0 g_0 - x^{n/2} f_1 g_1) + x^{n/2}(f_0 + f_1)(g_0 + g_1).$$

Computing $f_0 + f_1$ is done by loading the first, say 16, coefficients of f_0 into one 16-way vector and the first 16 coefficients of f_1 into another vector then adding them together. Next, we load the next 16 coefficients of f_0 and f_1 and add them together. We repeat these steps for all coefficients of f_0 and f_1 (also for $g_0 + g_1$). Multiplying by $x^{n/2}$ corresponds to working with coefficients at index $+n/2$. Note that when computing $f_i g_j$, the actual multiplication is performed once the degree of f_i and g_i reaches 16 and is computed using the schoolbook multiplication. We also vectorize this schoolbook multiplication by multiplying 16 pairs at once since there are enough independent multiplications to choose from. For example, for $d = 256, n = 3$, there are $3^4 = 81$ 16-by-16 multiplications.

Randomness generation. Our construction requires randomness generation for deriving the initial vector $\mathbf{s}$. We decided to use hash functions with the extendable output function SHAKE-128 (standardized in FIPS 202 [41]). The reason why we chose SHAKE-128 is that it has received a lot of cryptanalysis at least in the SHA-3 competition. Another advantage in using SHAKE-128 is that we can adapt some parts of the publicly available optimized AVX2 implementation from Kyber [11].

Note that deriving the vector $\mathbf{s}$ can be pre-computed *per key* k. In many scenarios, the same key k is used for different evaluations of input string x. This means that once the key k is known, deriving $\mathbf{s}$ can be pre-computed for each key k and stored it even before knowing x.

Rounding operation. To round numbers from $\log q$ bits to $\log p$ bits, we simply remove $(\log q - \log p)$ least significant bits. In the implementations, we use logical operation `and` with `1` for the bits that we want to keep and with `0` for the bits that we want to discard. Since q and p are fixed, the pattern of ones and zeros used for this logical operation is also fixed. Therefore, we keep this masking pattern as a constant.

Value extraction. We need to extract from PRG output only the part to be used as an input for the next PRG call. We begin with masking the corresponding part of the vector $\mathbf{t}$, namely, to round from $\log q$ bits to $\log p$ bits. This is done by the masking technique similar to the rounding operation. Then we rearrange the rounded values into new vectors $\mathbf{s}_i$. That is, we extract each $\log q - \log p$ bits from $\mathbf{t}$ and assign to each $s_{i,j}$ for $0 \leq i < n$ and $0 \leq j < d$.

5.2 Parameters

We recall notations and parameters in our construction. Let q be a modulus and p be a rounding parameter such that $q > p$ and $\log q = \log p + \log B$ where q, p, and B are power of two. Let $\mathcal{R} = \mathbb{Z}_q[x]/(x^d + 1)$. Let $\mathbf{A} \in \mathcal{R}_q^{m \times n}$ be a random public matrix. Let $\mathbf{s} \in \mathcal{R}_{\log B}^n$ be a small secret vector. We want parameter sets satisfying the expansion-rate-ω output length requirement and achieving at least 128-bit security for the underlying module-LWR problem.

Consider the length condition for the module-LWR-based PRG, the input is the vector $\mathbf{s} \in \mathcal{R}_{\log B}^n$ and the output is the rounded vector $\lfloor \mathbf{As} \rfloor_p \in \mathcal{R}_p^m$. This means that the output size is $m \cdot d \cdot \log p$ and the input size is $n \cdot d \cdot (\log B)$. Therefore, the equations that we consider is:

$$\begin{aligned} m \cdot d \cdot \log p &\geq \omega \cdot (n \cdot d \cdot (\log B)) \\ m \cdot d \cdot (\log q - (\log B)) &\geq \omega \cdot (n \cdot d \cdot (\log B)) \\ m \cdot (\log q - (\log B)) &\geq \omega \cdot (n \cdot (\log B)). \end{aligned}$$

Note that we want $\omega \geq 2$, and the larger ω is the shallower the tree becomes.

We modified Kyber's script [11] to suit our setting and use it to estimate the security of the underlying module-LWR problem. In this security analysis,

the major modification is that we use a uniform distribution instead of a binomial one. The conditions in which we search for good parameter sets for the module-LWR variant are as follows:

- $q = 2^{16}$, fixed;
- $\log B \in \{1, 2, 3, 4\}$;
- $d \in \{128, 256\}$;
- $n \in \{5, 6\}$ for $d = 128$ and $n = 3$ for $d = 256$;
- $m \in \{4, 8, 16, 32\}$.

Good parameters that achieve at least 128-bit security and $\omega \geq 2$ are listed in Table 2.

Table 2: Good parameters achieving at least 128-bit module-LWR security and the expansion-rate-ω condition

d	m	n	$\log B$	security	ω	mul	mem	depth	cost
128	4	5	4	145	2	69120	5120	128	8294400
128	4	6	2	131	4	41472	6144	64	2654208
128	16	5	4	133	8	69120	20480	43	2972160
128	32	5	4	133	16	69120	40960	32	2211840
256	4	3	4	167	4	62208	6144	64	3981312
256	8	3	4	167	8	62208	12288	43	2674944
256	16	3	4	167	16	62208	24576	32	1990656
256	32	3	4	167	32	62208	49152	26	1617408

We fix the modulus $q = 2^{16}$. Abbreviations 'mem' denotes the size in bytes of the matrix $\mathbf{A}$; 'depth' denotes the tree's depth; 'mul' denotes the multiplication cost per depth; and 'cost' = mul $\times$ depth denotes the total multiplication cost for the full input length ℓ.

From Table 2, we observe that the module-LWR variant with $d = 256$ provides higher security (with larger gap) than what we target, i.e., 128 vs. 167. Using $d = 128$ allows better adjustment in terms of security level, i.e., 131–145 is closer to 128, and less memory usage to store the matrix $\mathbf{A}$. However, it comes with a trade-off of increasing the multiplication cost.

According to the possible parameter sets, it is debatable which one should be selected for the implementation. Smaller m allows less memory usage at the price of increasing multiplication cost and higher tree's depth. On the other hand, larger m allows larger expansion rate ω. We decided that we prioritize the runtime (multiplication cost) and chose the parameter set with the tree's depth power of two to aid implementation. Concretely, we chose the second to last row of Table 2 with the following parameters: $q = 2^{16}$, $d = 256$, $m = 16$, $n = 3$ $\log B = 4$, and $\omega = 16$. This gives the (M)LWR security estimation of 167. If we

consider the security loss due to multiple PRF evaluations, this parameter set allows up to 2^{34} PRF queries to still preserve 128-bit PRF security.

As for comparison to the LWR variant, we state here the parameter set that we use in our implementation and refer to Appendix A for more details. Since we want to be able to compare to our module-LWR implementation, we decided to use similar parameters, namely, same q and ω. Concretely, we use the following parameters: $q = 2^{16}$, $\omega = 16$, $n = 800$, $m = 1840$, and $\log B = 2$. This gives the (M)LWR security estimation of 131.

5.3 Performance

To demonstrate the efficiency of our lattice-based PRF, we implemented Construction 1 and Construction 2 (see Appendix A) with the parameters mentioned in the previous subsection. We measured the run time of our implementations in cycle counts which are the medians of 10000 executions obtained from an Intel Core i7-6600U (Skylake) with TurboBoost turned off running at 2.6 GHz. Generating the public parameter $\mathbf{A}$ (setup phase) costs 78815 cycles. The precomputation part (generating randomness, i.e., $\mathsf{Random}(k)$ and reformatting the vector, i.e., $\mathsf{Reformat}(\mathbf{t})$) all together costs 4630 cycles. The main loop costs 711374 cycles. The outputting costs 236124 cycles. Note that these cycle counts are for 128-bit input and 6144-byte output.

We would like to emphasize that if we consider evaluating PRF with consecutive x in a counter mode, we do not need to compute the tree from its root to one of its leaves; we can reuse the computation from previous iterations. As a result, it takes only 242347 cycles for the main loop, and 236390 cycles for the outputting. Note that the cost of the main loop significantly decreases due to shorter paths to reach the leaves. However, the outputting costs stays the same because we always do one matrix-vector multiplication.

Table 3 below compares our timing results with SPRING [6] (in terms of numbers of cycles per output byte). Recall that SPRING has its security based on the rounded subset product hardness assumption while the security of our PRF relies on the more standard (module-)LWR hardness assumption.

References

1. *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*. IEEE Computer Society, 1984. 18
2. *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 2018. 17
3. Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In *USENIX 2016 [24]*, pages 327–343, 2016. 4, 12
4. Jacob Alperin-Sheriff and Daniel Apon. Dimension-preserving reductions from LWE to LWR. *IACR Cryptology ePrint Archive*, 2016:589, 2016. 12
5. Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. Learning with rounding, revisited - new reduction, properties and applications. In *CRYPTO 2013 [14]*, pages 57–74, 2013. 5, 12

Table 3: Actual performance comparison of PRFs

Schemes	i7 Ivy Bridge	i5 Haswell	i7 Skylake
SPRING-BCH	46.0	19.5	-
SPRING-CRT	23.5	-	-
Ours, MLWR	-	-	39.4
Ours, LWR	-	-	64.2

Performance is measured in terms of the number of cycles per output byte. Cycle counts for our implementations are the total cost (include the pre-computation) on average to evaluate 100 inputs in a counter mode. Note that Haswell and Skylake have AVX2 instructions. Note also that the highest CPU frequency of Haswell is actually higher than that of Skylake. This suggests that if we were running on a machine with Haswell our implementations should not be slower than on a machine with Skylake.

6. Abhishek Banerjee, Hai Brenner, Gaëtan Leurent, Chris Peikert, and Alon Rosen. SPRING: fast pseudorandom functions from rounded ring products. In *FSE 2014 [17]*, pages 38–57, 2014. 2, 5, 16
7. Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In *EUROCRYPT 2012 [42]*, pages 719–737, 2012. 1, 2, 5, 7, 8, 9, 12
8. Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA 2016 [26]*, pages 10–24, 2016. 12
9. Daniel J. Bernstein. Batch binary edwards. In *CRYPTO [23]*, pages 317–336, 2009. 13
10. Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the hardness of learning with rounding over small modulus. In *TCC 2016 [27]*, pages 209–224, 2016. 12
11. Joppe W. 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 *EuroS&P 2018 [2]*, pages 353–367, 2018. 9, 12, 14, 22
12. Charles Bouillaguet, Claire Delaplace, Pierre-Alain Fouque, and Paul Kirchner. Fast lattice-based encryption: Stretching spring. In *PQCrypto 2017 [31]*, pages 125–142, 2017. 5
13. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *TOCT*, 6(3):13:1–13:36, 2014. 7, 9
14. Ran Canetti and Juan A. Garay, editors. *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*. Springer, 2013. 17
15. Moses Charikar, editor. *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*. SIAM, 2010. 19
16. Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT 2011 [33]*, pages 1–20, 2011. 12

17. Carlos Cid and Christian Rechberger, editors. *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*. Springer, 2015. 17
18. Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - dilithium: Digital signatures from module lattices. *IACR Cryptology ePrint Archive*, 2017:633, 2017. 9
19. Harold N. Gabow and Ronald Fagin, editors. *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*. ACM, 2005. 19
20. Rosario Gennaro and Matthew Robshaw, editors. *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*. Springer, 2015. 18
21. Henri Gilbert, editor. *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*, volume 6110 of *Lecture Notes in Computer Science*. Springer, 2010. 19
22. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *FOCS 1984 [1]*, pages 464–479, 1984. 2, 5, 6
23. Shai Halevi, editor. *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*, volume 5677 of *Lecture Notes in Computer Science*. Springer, 2009. 17
24. Thorsten Holz and Stefan Savage, editors. *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. USENIX Association, 2016. 17
25. Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2007. 6
26. Robert Krauthgamer, editor. *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*. SIAM, 2016. 17
27. Eyal Kushilevitz and Tal Malkin, editors. *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part I*, volume 9562 of *Lecture Notes in Computer Science*. Springer, 2016. 17
28. Thijs Laarhoven. *Search problems in cryptography*. PhD thesis, Eindhoven University of Technology, The Netherlands, 2015. 12
29. Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *CRYPTO 2015 [20]*, pages 3–22, 2015. 12
30. Thijs Laarhoven, Michele Mosca, and Joop van de Pol. Finding shortest lattice vectors faster using quantum search. *Des. Codes Cryptography*, 77(2-3):375–400, 2015. 12
31. Tanja Lange and Tsuyoshi Takagi, editors. *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*, volume 10346 of *Lecture Notes in Computer Science*. Springer, 2017. 17
32. Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptography*, 75(3):565–599, 2015. 7, 9
33. Dong Hoon Lee and Xiaoyun Wang, editors. *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application*

of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings, volume 7073 of *Lecture Notes in Computer Science*. Springer, 2011. 17

34. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT 2010 [21]*, pages 1–23, 2010. 7
35. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, 2013. Preliminary version in Eurocrypt 2010 [21]. 9
36. Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *SODA 2010 [15]*, pages 1468–1480, 2010. 12
37. Moni Naor and Omer Reingold. Synthesizers and Their Application to the Parallel Construction of Pseudo-Random Functions. *J. Comput. Syst. Sci.*, 58(2):336–375, 1999. Preliminary version in FOCS 1995. 9
38. Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *J. ACM*, 51(2):231–262, 2004. Preliminary version in FOCS 1997. 9
39. Moni Naor, Omer Reingold, and Alon Rosen. Pseudorandom functions and factoring. *SIAM J. Comput.*, 31(5):1383–1404, 2002. Preliminary version in STOC 2000. 9
40. Phong Q. Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *J. Mathematical Cryptology*, 2(2):181–207, 2008. 12
41. National Institute of Standards and Technology. SHA-3 standard: Permutation-based hash and extendable-output functions. FIPS PUB 202, 2015. `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`. 14
42. David Pointcheval and Thomas Johansson, editors. *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*. Springer, 2012. 17
43. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC 2005 [19]*, pages 84–93, 2005. 9
44. Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(2):181–199, September 1994. 12

A LWR-based PRG

This Appendix explains our PRF construction based on the LWR hardness assumption. Note that, in general, this construction is very much similar to the one in the main article which is based on the module-LWR hardness assumption. In order not to repeat the same details, we only highlight the differences between the two constructions.

A.1 Construction

Construction 2 outlines our LWR-variant PRF. Recall that the PRF takes two inputs, namely, a key k and a string x of length ℓ, and outputs a pseudorandom string z. The matrix $\mathbf{A}$ is a public parameter. Similar to the module-LWR construction, the LWR construction is also divided into three parts: pre-computation, main loop, and outputting. Below, we emphasize the differences between the two constructions parts by parts.

Construction 2 PRF from LWR

Input: PRF's key $k = \{0,1\}^*$ and a bit string $x = \{0,1\}^\ell$
Output: Pseudorandom string $z = \{0,1\}^{\ell'}$

$\mathbf{t} = \mathsf{Random}(k)$
$\mathbf{s} = \mathsf{Reformat}(\mathbf{t})$ } Pre-computation
for $i = 0$ **to** $\ell - 1$ **step by** $\tilde{\omega}$ **do**
  $c = x[\tilde{\omega}i] \;\; x[\tilde{\omega}i{+}1] \;\ldots\; x[\tilde{\omega}i{+}\tilde{\omega}{-}1]$ ▷ View $\tilde{\omega}$ bits as number in $[0, \omega{-}1]$
  for $j = c{\cdot}(m/\omega)$ **to** $(c+1){\cdot}(m/\omega) - 1$ **do**
    $t_j = \mathbf{A}_j \cdot \mathbf{s}$ ▷ Multiply row j of $\mathbf{A}$ to $\mathbf{s}$
    $t_j = \lfloor t_j \rfloor$
  end for
  $\mathbf{s} = \mathsf{Reformat}(\mathbf{t})$ ▷ Extract bits from $t_{c(m/\omega)}, t_{c(m/\omega)+1}, \ldots, t_{(c+1)(m/\omega)-1}$
end for
for $j = 0$ **to** $m - 1$ **do**
  $t_j = \mathbf{A}_j \cdot \mathbf{s}$
  $t_j = \lfloor t_j \rfloor$
  $z = z || t_j$
end for
return z

Pre-computation. There is no differences in the way we generate randomness, i.e., $\mathsf{Random}(k)$, and reformat, i.e., $\mathsf{Reformat}(\mathbf{t})$. Note however that $\mathbf{s} \in \mathbb{Z}^n_{\log B}$ (instead of $\mathcal{R}^{n'}_{\log B}$).

The reasons for having $\mathsf{Random}(k)$ are also the same, namely, in case the input k is not random or does not have the preferred length. If this is not the case, then $\mathsf{Random}(k)$ can be skipped and directly proceed to $\mathsf{Reformat}(\mathbf{t})$.

Main loop. Since the underlying PRG is now constructed from LWR, we need to change polynomial multiplications to integer multiplications. Nevertheless, a similar optimization for computing $\mathbf{As}$ is still applied, i.e., each iteration we compute vector-vector multiplications instead of a matrix-vector multiplication. For the LWR case, we need to multiply only m/ω rows of the matrix $\mathbf{A}$ to the vector $\mathbf{s}$. This time, m/ω is certainly more than one. We added an inner loop (line 5–8 in Construction 2) to iterate multiplying those rows. Once we multiply all those m/ω rows and before moving the next iteration of the outer loop, we call $\mathsf{Reformat}(\mathbf{t})$ to extract bits from $\mathbf{t}$ and assign to $\mathbf{s}$. Note that $\mathbf{t}$ contains m/ω elements.

Outputting. This part remains the same, namely, we perform the entire matrix-vector multiplication to increase the output length. Then we change the result into the desire format and output it.

A.2 Implementation

Two major differences between the module-LWR and the LWR implementations are multiplication algorithm and parameter set. In what follows, we explain how we handle multiplications when we cannot use fast polynomial multiplication algorithms, and then we state possible LWR parameter sets.

Multiplication. Since we use the plain LWR, i.e., having neither ring nor module additional structure, our multiplications are simply integer vector multiplications which can be implemented efficiently. Another advantage of using integers instead of polynomials is that we do not need to perform polynomial reductions.

Assume that we can perform w pairs of integer vector multiplications. To multiply row i of matrix $\mathbf{A}$ with vector $\mathbf{s}$, we load the first w elements of $\mathbf{A}_i$ and the first w elements of $\mathbf{s}$, then we perform integer vector-vector multiplication. After that, we load the next w element of $\mathbf{A}_i$ and the next w elements of $\mathbf{s}$, and again we perform another integer vector-vector multiplication. We repeat these steps until we multiply the entire row of $\mathbf{A}$ with $\mathbf{s}$. Since we always multiply $A_{i,j}$ with s_j, i.e, index j of $\mathbf{A}_i$ with index j of $\mathbf{s}$, this means that data are aligned in correct slots and no permutation needed. Therefore, this multiplication step works very well with vectorization.

Next step is to sum all the results of the integer vector-vector multiplications into a single integer. Unlike the multiplication step, this addition step requires permutations because we want to add data which contain in the same vector. To do so, we make use of the instruction `vphaddd` to perform horizontally additions and the instruction `vperm2i128` to perform permutations.

Parameters. We recall parameters in our construction. Let q be a modulus and p be a rounding parameter such that $q > p$ and $\log q = \log p + \log B$ where q, p, and B are power of two. Let $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$ be a random public matrix. Let $\mathbf{s} \in \mathbb{Z}_{\log B}^n$ be a small secret vector. Similar to the module-LWR variant, we want to set these parameters such that they satisfy the expansion-rate-ω output length requirement and achieve at least 128-bit security for the underlying LWR problem.

Recall the length condition. In our LWR-based PRG, the input is the vector $\mathbf{s}$ and the output is the rounded vector $\lfloor \mathbf{As} \rfloor_p$. That is, we have the output size $m \cdot \log p$ and the input size $n \cdot (\log B)$. Therefore, the equation we consider is:

$$m \cdot \log p \geq \omega \cdot (n \cdot (\log B))$$
$$m \cdot (\log q - (\log B)) \geq \omega \cdot (n \cdot (\log B)).$$

Similar remarks of requiring $\omega \geq 2$ and larger the ω shallower the tree also apply to the LWR variant.

Now, we consider the security of the underlying LWR problem. We modified Kyber's script [11] to suit for our security estimation. Two main modifications are: (1) setting the ring dimension to 1 (because we use the plain LWR); and (2) using a uniform distribution instead of a binomial one. The conditions in which we search for good parameter sets are as follows:

- $q = 2^{16}$, fixed
- $\log B \in \{1, 2, 3, 4\}$
- n multiple of 32 in range [640, 800]
- m among 4,8,16 and 32 multiples of $\lceil n \cdot \log B / \log p \rceil$

Table 4 shows possible parameter sets for the LWR construction. To select which parameters to implement, we prioritize the runtime and choose the last row, namely, $m = 1840, n = 800, \log B = 2$ which gives $m/\omega = 115$, the tree's depth $= 32$, and the estimated security of 131.

Table 4: Good parameters achieving at least 128-bit LWR security and the expansion-rate-ω condition

m	n	$\log B$	security	ω	m/ω	mem	depth	mul
368	640	3	135	2	184	471040	128	15073280
384	672	3	143	2	192	516096	128	16515072
404	704	3	152	2	202	568832	128	18202624
424	736	2	131	4	106	624128	64	4993024
440	768	2	137	4	110	675840	64	5406720
460	800	1	129	8	115	736000	43	1995200
1840	800	2	131	16	115	2944000	32	2944000

We fix the modulus $q = 2^{16}$. Abbreviations 'mem' denotes size in bytes of matrix $\mathbf{A}$; 'depth' denotes tree's depth; and 'mul' $= m/\omega \times$ depth denotes the numbers of multiplications.