Fast Lattice-Based Encryption: Stretching Spring

Charles Bouillaguet, Claire Delaplace, Pierre-Alain Fouque, Paul Kirchner

Fast Lattice-Based Encryption: Stretching SPRING

Charles Bouillaguet[1], Claire Delaplace[1,2], Pierre-Alain Fouque[2], and Paul Kirchner[3]

[1] Université de Lille, CRIStAL
[2] Université de Rennes 1, IRISA
[3] Ecole Normale Supérieure

Abstract. The SPRING pseudo-random function (PRF) has been described by Banerjee, Brenner, Leurent, Peikert and Rosen at FSE 2014. It is quite fast, only 4.5 times slower than the AES (without hardware acceleration) when used in counter mode. SPRING is similar to the PRF of Banerjee, Peikert and Rosen from EUROCRYPT 2012, whose security relies on the hardness of the Learning With Rounding (LWR) problem, which can itself be reduced to hard lattice problems. However, there is no such chain of reductions relating SPRING to lattice problems, because it uses small parameters for efficiency reasons.
Consequently, the heuristic security of SPRING is evaluated using known attacks and the complexity of the best known algorithms for breaking the underlying hard problem.
In this paper, we revisit the efficiency and security of SPRING when used as a pseudo-random generator. We propose a new variant which is competitive with the AES in counter mode without hardware AES acceleration, and about four times slower than AES with hardware acceleration. In terms of security, we improve some previous analysis of SPRING and we estimate the security of our variant against classical algorithms and attacks. Finally, we implement our variant using AVX2 instructions, resulting in high performances on high-end desktop computers.

Keywords. Pseudo-random generator. Stream cipher. Ring-LWR. Rejection sampling.

1 Introduction

Lattice-based cryptography is arguably one of the most mature proposal for post-quantum cryptography. One of the most important issue in this research direction is the size of the parameters. Indeed, when designers propose a cryptosystem whose security is related to hard lattice problems, theoretical work gives asymptotic security guarantees which are hard to assess in practice. If we want to build a practical scheme, security safeguards can be relaxed a little. Many interesting schemes have been proposed in connection with lattice problems, ranging from the SWIFFT hash function [LMPR08] to the SPRING pseudo-random function

family (PRF). Even though there is no security reduction for the latter, we can still estimate the level of security using standard algorithm for the underlying hard lattice problem on which it is based.

Symmetric primitives whose security is related to some hard computational problems have been known for a few decades. For instance, the hash function of Chaum, van Heijst and Pfitzmann is often taught as an example of a provably collision-resistant hash function (under the discrete log assumption). These "provably secure" primitives are very inefficient, and are thus rarely used in practice.

Building efficient and "provably secure" symmetric primitives is therefore an interesting research direction; SWIFFT and VSH [CLS06] are two nice examples thereof. Even if there is no direct security reduction, due to the choice of small parameters, the security of a primitive can be based on the hardness of well-known computational problems. At the very least, this gives an intuition as to why the primitive might be secure (or not). The SPRING PRF introduced at FSE 2014 [BBL+15] belongs to this category: the hardness of the Ring-Learning with Rounding problem is necessary for its security. However, as it was already mentioned in [BBL+15], we need to choose small parameters, if we want the PRF to be efficient. Thus, SPRING is not provably secure. As explained in section 3, it does not seems to undermine the security of the scheme though.

Its performance is not horribly bad, since it is of the same order of magnitude than that of the AES. The main drawback of this primitive is the size of the key, about 8 kilobytes. Such a huge size of key require to use key derivation functions (such as HKDF) to expand a 128-bit or 256-bit key obtained after a standard key exchange protocol.

1.1 Related work

Banerjee, Peikert and Rosen introduced in [BPR12] a new family of pseudorandom functions, that we call "BPR" in this paper, based on *rounded products* in well-chosen polynomial rings. Let n be a power of two (in this paper $n = 128$), and consider the polynomial ring:

$$R_q \stackrel{def}{=} \mathbb{Z}_q[\mathbf{x}] \,/\, \langle \mathbf{x}^n + 1 \rangle \,.$$

This is the ring of polynomials taken modulo $\mathbf{x}^n + 1$ and whose coefficients are taken modulo q. We denote by R_q^* the set of invertible elements in R_q. Given a positive integer k, the BPR family of PRFs is the set of functions $F_{\mathbf{a},\mathbf{s}} : \{0,1\}^k \to \{0,1\}^n$ indexed by a unit $a \in R_q^*$ and by a vector $\mathbf{s} = (\mathbf{s}_1, \ldots, \mathbf{s}_k)$ of units. The functions are defined as :

$$F_{\mathbf{a},\mathbf{s}}(x_1, \ldots, x_k) = S\left(\mathbf{a} \cdot \prod_{i=1}^{k} \mathbf{s}_i^{x_i}\right),$$

where S is a "rounding" function that maps each coefficient of the product polynomial into a single bit. The BPR family enjoys a nice security proof : its security can be reduced to the Ring-LWR problem, which is proved in [BPR12]

to be equivalent to worst-case lattice problems. In [BBL+15], Banerjee *et al.* proposed and implemented an efficient yet unproven variant of the BPR family called SPRING (short for "**s**ubset-**p**roduct with **r**ounding over a **ring**"). They reduced the size of the parameters, yielding a fast implementation. However, unlike BPR, the SPRING family does not enjoy "provable security", because the choice of small parameters prevents the reduction to go through. Furthermore, in some cases, the rounding, which is the core of SPRING and BPR, may produce biased output bits, even when its input is uniformly distributed. For instance, any function rounding a coefficient in $\mathbb{Z}_{257}$ to a single bit is bound to have a very detectable bias of at least $1/257$: at the very minimum, the numbers of inputs yielding the two possible outputs differ by at least 1. In the original BPR construction, this is not a problem because the modulus is exponentially large, and thus the bias is exponentially small.

In SPRING, however, where small moduli ($q = 257$ and $q = 514$) are used for the sake of efficiency, precautions have to be taken to deal with such an eventual bias. To cope with this problem, the designers of SPRING proposed two different instantiations: SPRING-CRT and SPRING-BCH.

The first solution, SPRING-CRT, uses an even modulo $q = 514$ to make the problem disappear : $\mathbb{Z}_{514}$ can be split in two equal halves, and an unbiased "truncated" bit can be produced from $x \in \mathbb{Z}_{514}$ by checking if $x \geq 257$. It is the most efficient of the two original SPRING variants, but it is open to a subexponential attack. This attack is more efficient than trying to break the underlying hard problem using the usual algorithms. This shows that the weakening of the security guarantee provided by a reduction to hard problems can have serious consequences.

The second construction SPRING-BCH uses an odd modulus $q = 257$, which avoids the subexponential attack, but introduces a large rounding bias in the output bits. In this case, a post-processing step is added to reduce the bias using a BCH error-correcting code. The code computes linear combinations of the output bits, so that the final bias is $1/q^d = 2^{-177}$ where $d = 22$ is the minimal distance of the linear code. The downside is that the throughput is divided by two compared to SPRING-CRT. The most efficient attack against SPRING-BCH consists in detecting this small bias in the output.

Brenner *et al.* complemented these results in [BGL+14] by implementing SPRING-BCH on FPGAs and discussed the properties of SPRING in hardware implementations against side-channel attacks.

1.2 Our contributions

We propose a simpler, faster PRG derived from SPRING and revisit the security of all these schemes. On a desktop computer, our variant, called SPRING-RS, is four times faster than SPRING-CRT, and using AVX2 instruction, it is twice more efficient than the AES-128 without AES-NI instructions and 5 times less efficient than the AES-128 with AES-NI instructions on recent CPUs.

New SPRING Variant. Our main idea to improve SPRING deals with the way the "rounding" is performed. Because it is difficult to extract a single unbiased bit from a $\mathbb{Z}_{257}$-coefficient, we use a simple form of rejection sampling:

$$S(x) : \begin{cases} 0 & \text{if } x \in [0; 128) \\ 1 & \text{if } x \in [128; 256) \\ \perp & \text{if } x = 256 \end{cases}$$

This produces 0 and 1 with the same probability; the 0.4% of $\perp$ outputs are simply discarded. Applying this post-processing step with $q = 257$ makes it easy to obtain SPRING-RS, a PRG running at 0.996 times the speed of SPRING-CRT, while providing a higher level of security than SPRING-BCH. In addition, it allows for simpler implementations.

Such a technique has been used before, for instance in [Lyu09], where Luybashevsky proposes a way to generate a value independently of the secret information in an identification scheme.

An obvious downside of this approach is that the number of available output bits is not always the same. This complicates designing a PRF using this approach, because a PRF has to produce a specified number of pseudo-random output bits, regardless of the circumstances. It is nevertheless possible to build a PRF using rejection sampling. Its speed should be intermediate between that of SPRING-CRT and SPRING-BCH.

On the other hand, a PRG might be acceptable even though it produces pseudo-random bits slightly irregularly. This is for instance the case of the self-shrinking generator [MS95]. Usually a PRG is "clocked" until enough pseudo-random bits have been obtained. In this setting, the fact that the number of bits produced each time the PRG is clocked may vary is not problematic.

Rejection sampling produces unbiased outputs, and as such it eliminates the most efficient attack against SPRING-BCH. Thus, it is likely to be *more secure*. In fact, our best attack described in section 3 can be achieved with advantage 2^{-900}. However, this level of security is hardly necessary, hence we decided to trade some security for speed. An obvious way to do so is to "truncate" less, for instance by extracting not one, but two, three or four bits out of a single $\mathbb{Z}_{257}$-coefficient. For each input $\mathbf{s} \in \{0,1\}^k$, SPRING-BCH returns a 64-bit output, while the SPRING-RS function, extracting four bits out of each coefficient, returns on average 510 bits. This increases the throughput of the PRG about 8 times, compared to SPRING-BCH.

While extracting more than one bit from a $\mathbb{Z}_{257}$-coefficient would not be immediate in the previous SPRING variants, it is extremely easy using rejection sampling. Because $\mathbb{Z}_{257}$ is "reduced" to a set of 2^8 elements, truncating to k bits boils down to keeping only the k most significant bits. We claim that we have at least 128 bits of security and our best attack can be achieved with advantage 2^{-900}.

As the main disadvantage of SPRING is its large key size, we also propose several way to reduce it. The first one uses a 128-bit secret key and another PRG in order to forge the secret polynomials of SPRING. The other ones use

"smaller" instantiations of SPRING in a bootstrapping phase to generate the secret polynomials. We distinguish two cases. (1) In the first case, we use a SPRING instantiation with five secret polynomials to forge other secret polynomials that will be the secret key of another SPRING instantiation, and we reiterate the process until we have a $k+1$-polynomial SPRING instantiation. In this cases, all secrets are reset at each step, and we never use the same polynomial in two different instantiations. (2) In the second case, we assume that our SPRING instantiation has circular security, and we use only the three first polynomial $\mathbf{a}, \mathbf{s}_1, \mathbf{s}_2$ to forge the next polynomial, using this partial SPRING instantiation, and we reiterate the process until all $\mathbf{s}_i$ are forged. This reduces the key to 3072 bits.

More Cryptanalysis. We propose new attacks against various SPRING instantiations. The first one we present is a simple attack to distinguish the output of all SPRING variant —including ours— from uniform. This attack is a birthday attack on a small part of the internal state of the cipher, meaning that two different subset-products yield the same result. Assume that we split the input $\mathbf{x} = (x_1, \ldots, x_k)$ into $(\mathbf{w}, \mathbf{z}, x_k)$ with $\mathbf{w}$ is 1 bits and $\mathbf{z}$ is $k-2$ bits and x_k is the last bit of $\mathbf{x}$. If we have a collision $F_{\mathbf{a},\mathbf{s}}(\mathbf{w}, \mathbf{z}, 0) = F_{\mathbf{a},\mathbf{s}}(\mathbf{w}, \mathbf{z}', 0)$, for all $\mathbf{w}$, and if we ask what is the output of $F_{a,\mathbf{s}}(\mathbf{w}, \mathbf{z}, 1)$, then it is also the output of $F_{\mathbf{a},\mathbf{s}}(\mathbf{w}, \mathbf{z}', 1)$ with probability greater than $1/2$ and we can predict the PRG. When attacking the SPRING PRFs, we can choose z, and then use the Floyd cycle detection algorithm to reduce the amount of memory needed to find the collision.

Our second original attack works in a simplified instantiation of the scheme, where the first polynomial a is known and is supposed to be identically equal to 1. In this case, we describe a lattice attack.

The third attack we present is similar to the attack of [BBL+15], where the authors consider that a whole part of the key is known, namely all $(\mathbf{s}_i)_{1 \leq i \leq k}$ and only $\mathbf{a}$ is secret. Here classical algorithms such as BKW or lattice reduction algorithms have to be considered.

Finally, we propose an algebraic attack, with the assumption that an adversary may have learned exactly which coefficients have been rejected using side-channel timing attacks. Then, he will have to solve a polynomial system, using Gröbner bases algorithms, to recover the coefficients of the secret polynomials.

We also revisit the attack presented by [BBL+15] on the SPRING-CRT instantiation and propose to use better algorithms to detect correlation using fast matrix multiplication, in appendix A.

Implementation. Just like the designers of the original SPRING did, we implemented our variant using SIMD instructions available on most desktop CPUs. We borrow most implementation techniques from the designers of SPRING. This is in particular the case of an efficient implementation of polynomial multiplication in R^*_{257} thanks to a vectorized FFT-like algorithm using the SSE2

instruction set available in most desktop CPUs. These instructions operate on 128-bit wide "vector registers", allowing us to perform arithmetic operation on batches of eight $\mathbb{Z}_{257}$ coefficients in a single instruction.

We pushed the implementation boundary a little further by writing a new implementation of this operation using the AVX2 instruction set, providing 256-bit wide vector registers. These instructions are available on recent Intel CPUs based on the "Haswell" microarchitecture or later. Without surprise, this yields a twofold speedup and raises a few interesting programming problems.

Note that the AVX2-optimized FFT algorithm can be back-ported to all the cryptographic constructions relying on the same polynomial multiplication modulo 257, such as [BBL+15,LMPR08,LBF08], yielding the same 2× speedup. Our code is available for others to use at:

`https://github.com/cbouilla/spriiiiiiiing`

Table 1. Implementation results for SPRING variants, in Gray code counter mode (CTR). Speeds are presented in processor cycles per output byte. Starred numbers indicate the use of AES-NI instructions. Daggers indicate the use of AVX2 instructions.

	SPRING-BCH	SPRING-CRT	AES-CTR	SPRING-RS
ARM Cortex A7	445	[not implemented]	41	59
Core i7 "Ivy Bridge"	46	23.5	1.3*	6
Core i5 "Haswell"	19.5†	[not implemented]	0.68*	2.8†

The table 1 give the performance of our SPRING-RS implementation, as well as previous implementation of SPRING, and compare them to the performances of the best AES implementations we could use as benchmark (we choose to compare our performances to those of AES with AES-NI instructions when we could). SPRING-RS is much faster than previous implementations of SPRING, and is about four times slower than AES with AES-NI instructions SPRING-RS is also competitive with AES without AES-NI instructions (and is even twice more efficient while using AVX2 instruction).

2 The SPRING Family of PRFs and PRGs

Ring-LWR Problem. Banerjee, Peikert and Rosen introduced in [BPR12] a derandomized version of LWE [Reg05] called Learning With Rounding (LWR), and its ring analog Ring-LWR (or RLWR), in which we are more interested here. They gave the following definition of a Ring-LWR distribution :

Definition 1. *Let n be an integer greater than 1 and let p and q be moduli such that $q \geq p \geq 2$. For $\mathbf{s} \in R_q$, define the ring-LWR distribution to be the distribution over $R_q \times R_p$ obtained by choosing a polynomial $\mathbf{a} \in R_q$, uniformly at random, computing $\mathbf{b} = \lfloor \mathbf{s} \cdot \mathbf{a} \rceil_p$ and outputting the pair $(\mathbf{a}, \mathbf{b})$. The function*

$\lfloor \cdot \rceil_p : R_q \to R_p$ *is a coefficient-wise rounding that maps the coefficients* $b_i \in \{0, \ldots, q-1\}$ *of* $\sum_{i=0}^{n-1} b_i \mathbf{x}^i$ *to* $\left\lfloor \frac{p \cdot b_i}{q} \right\rceil$.

As it has been noted by the authors of [BPR12], the rounding method $\lfloor \cdot \rceil$ can be replaced by the floor or the ceiling function, without major change to the problem. For implementation purposes, we chose to use the floor function $\lfloor \cdot \rfloor$ instead. Then, in the case of SPRING, for all $\mathbf{a} \in R_q$ computing $\lfloor \mathbf{a} \rfloor_p$ is equivalent to keeping the $\log_2(p)$ most significant bits.

For $\mathbf{s}$ chosen uniformly in R_q, the decision-Ring-LWR problem is to distinguish between independent samples $(\mathbf{a}_i, \mathbf{b}_i)$ drawn in the Ring-LWR distribution, and the same number of samples drawn uniformly at random in $R_q \times R_p$. While there are reductions between LWE and worst-case lattice problems, the reductions between LWR and LWE need q/p to be at least $\sqrt{n}$ [BPR12, AKPW13, BGM+15].

The SPRING Family. In [BPR12], it is proved that when $\mathbf{a}$ is uniform, $\mathbf{s}$ are independent discrete Gaussians, and when q is large enough the $F_{\mathbf{a},\mathbf{s}}$ function family is a secure PRF, assuming that the Ring-LWE problem is hard on R_q. However, as described in [BBL+15], the function family does not necessarily require such a large modulus q to be a secure PRF family. Also, they show that if a weakened Ring-LWR is hard where $\mathbf{s}$ is uniform, one can take a small q. [BBL+15] proposed a version of the SPRING function using the parameters :

$$n = 128,\ q = 257,\ p = 2,\ k = 64.$$

In this paper, we choose the same parameters n, q and k, and we allow $p \in \{2, 4, 8, 16\}$. The choice of a larger "truncation modulus" p allows us to generate more output bits with the same amount of work. However, it reveals more information about each coefficients of the "internal state" polynomial (we return half of the bits of each coefficient when $p = 16$) so the security of the instantiation decreases as p grows, but as discussed in section 3, using such parameters p should not put our system at risk. The choice of the modulus $q = 257$ is the same than in [LMPR08] for the SWIFFT hash-function. As discussed in [LMPR08], choosing q such that $q - 1$ a multiple of $2n$ allows for a fast FFT-like multiplication algorithm in R_q^*. In addition, using a Fermat prime $q = 2^{2^k} + 1$ has multiple advantages, including very efficient reduction modulo q.

2.1 SPRING-RS: Rounding with Rejection-Sampling

We introduce here a new PRG based on SPRING using rejection-sampling to eliminate the bias. Just like [BBL+15] we propose to use a counter-like mode using a Gray-code for efficiency. A Gray code is a simple way to order $\{0,1\}^k$ such that two successive values of the counter differ only by one bit. Then, when running SPRING in counter mode, we can compute the successive subset products with only one polynomial multiplication at each step. To transform the

i-th value j of a Gray counter to the $(i+1)$-th, the ℓ-th bit of the counter has to be flipped, where ℓ is the number of trailing zeroes in the binary expansion of $(i+1)$.

The internal state of the PRG is therefore composed of two k-bit integers i, j and a unit polynomial $\mathbf{P}$ in R_q^*. Each time the PRG is clocked :

- i is incremented
- The ℓ-th bit of j is flipped, where ℓ is the number of trailing zeroes in i.
- The polynomial is $\mathbf{P}$ is multiplied by s_j (resp. s_j^{-1}) when the ℓ-th bit of j is 1 (resp. 0).

The initial value of $\mathbf{P}$ is the secret element $\mathbf{a}$, which is part of the key. Finally, after the internal state has been updated, a variable number of pseudorandom bits are extracted from the new value of $\mathbf{P}$.

Extracting bits from the polynomial is done by the rounding operation. We apply the following rounding function to the n coefficients of $\mathbf{P}$ in parallel:

$$S : \begin{array}{ccc} \mathbb{Z}_q & \longrightarrow & \mathbb{Z}_p \cup \{\perp\} \\ x & \mapsto & \begin{cases} \perp & \text{if } x = -1 \\ \lfloor px/q \rfloor & \text{otherwise} \end{cases} \end{array}$$

This results in a sequence of n symbols. The $\perp$ are then "erased" from the output, yielding a variable number of elements of $\mathbb{Z}_p$, which are appended to the pseudorandom stream.

This produces uniformly distributed outputs in $\mathbb{Z}_p$ when the inputs are uniformly distributed in $\mathbb{Z}_q$. Rejecting one of the possible value (here, -1) effectively restricts the input set to $q-1$ elements. As long as p divides $q-1$, exactly $(q-1)/p$ inputs yield each possible output.

Reducing the Size of the Key. One of the main disadvantage of SPRING is the large size of its key (8 kB). We present here several ways to reduce the size of the key for all instantiations of SPRING. The key is composed of $k+1$ secret polynomials $\mathbf{a}, \mathbf{s}_1, \ldots, \mathbf{s}_k$ over R_q^*. Each such polynomial requires 1024 bits.

The most intuitive and most efficient way to reduce the key size of SPRING is to use a 128-bit master secret key denoted by K_m, and use it to generate pseudorandomly the secret polynomials using... another PRG. This is a bit unsatisfying: why not use the other PRG in the first place? However, this would be beneficial if the other PRG is slow or even not cryptographically secure (consider the Mersenne Twister for instance).

In order to drop the need for another PRG, it would be natural to use SPRING to "bootstrap" itself and generate its own secret polynomials. We propose two ways to do so[4].

[4] As discussed in section 3, SPRING seems to have a quite large level of security, even if the $\mathbf{s}_i$ are known. It has been asked to us why we do not choose to make part of the $\mathbf{s}_i$ polynomials known to reduce the size of the key. However, as shown by table 2, it may undermine the security of SPRING, especially when $p = 16$

One possibility is to consider than the "master" key is composed of the 5 polynomials $\mathbf{a}_0, \mathbf{s}_{0,1}, \mathbf{s}_{0,2}, \mathbf{s}_{0,3}, \mathbf{s}_{0,4}$. Then, we may evaluate the "mini-SPRING" function $F^0_{\mathbf{a}_0,\mathbf{s}_0}$ such that, $F^0_{\mathbf{a}_0,\mathbf{s}_0}(\mathbf{x}) = S\left(\mathbf{a}_0 \cdot \prod_{i=1}^4 \mathbf{s}_{0,i}^{x_i}\right)$, for all $x \in \{0,1\}^4$. Using this small instantiation of SPRING-RS with $p = 16$, we may generate up to 8192 pseudo-random bits. This is large enough to forge seven polynomials with very high probability. We call $\mathbf{a}_1, \mathbf{s}_{1,1}, \ldots, \mathbf{s}_{1,6}$ these new polynomials. We reiterate the process with the mini-SPRING function $F^1_{\mathbf{a}_1,s_1}$, and the output given by this PRG is large enough to forge between thirty and thirty-one polynomials. If we reiterate the process once more with those new polynomials, we will be able to forge the 65 polynomials $\mathbf{a}, \mathbf{s}_1, \ldots, \mathbf{s}_k$ of the full SPRING-RS. Using such a trick, we can substantially reduce the size of the key (from about 8 kB to about 700B).

It is possible to push this idea a bit further assuming the circular security of SPRING-RS. In that case, the "master" key is composed of the three secret polynomials $\mathbf{a}, \mathbf{s}_1, \mathbf{s}_2$, and we define the nano-SPRING function $F^0_{\mathbf{a},\mathbf{s}}$ such that, $F^0_{\mathbf{a},\mathbf{s}}(\mathbf{x}) = S\left(\mathbf{a} \cdot \mathbf{s}_1^{x_1} \cdot \mathbf{s}_2^{x_2}\right)$, for all $\mathbf{x} \in \{0,1\}^2$. Using this small instantiation of SPRING-RS, we can generate an output long enough to forge a new polynomial. This will be the next secret polynomial, $\mathbf{s}_3$. We reiterate the process with the micro-SPRING function $F^1_{\mathbf{a},\mathbf{s}}$ the function such that $F^1_{\mathbf{a},\mathbf{s}}(\mathbf{x}) = S(\mathbf{a} \cdot \prod_{i=1}^3 s_i^{x_i})$. We do not reset the Gray Counter, as long as the previous values will only give the output of $F^0_{\mathbf{a},\mathbf{s}}$. The new output thus generated is long enough to forge $\mathbf{s}_4$. If we reiterate the process once more, we will get an output long enough to generate two more polynomials. We reiterate it over again —two more times should be enough— until all the $\mathbf{s}_i$ are forged. We never reset the Gray Counter, otherwise an adversary may know all the $\mathbf{s}_i$ thus obtained.

Tuning for Vector Instructions. To obtain high performance implementations on modern hardware, it is necessary to be able to exploit vector instructions. In particular, it may be beneficial to tune some aspects of the function to the underlying hardware. Let d and r be integers such that $d \cdot r = n$, and let $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{r-1}$ be d-wide vectors with coefficient in $\mathbb{Z}_q$ such that:

$$\begin{pmatrix} \mathbf{v}_0 \\ \mathbf{v}_1 \\ \vdots \\ \mathbf{v}_{r-1} \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & \ldots & b_{d-1} \\ b_d & b_{d+1} & \ldots & b_{2d-1} \\ & \vdots & & \\ b_{(r-1)d} & b_{(r-1)d+1} & \ldots & b_n \end{pmatrix}. \tag{1}$$

Typically, d should be the width of available hardware vectors. For each vector $\mathbf{v}_i$, we apply the rounding function to all coefficients in parallel. If the resulting vector contains $\perp$, we reject the whole vector. This is illustrated in algorithm 1. Even though this also discards "good" coefficients, it allows a performance gain, because examining individual coefficients inside a hardware vector is often very inefficient. With $d = 1$, there is no wasted output. With $d = 8$ (SSE2 instructions on Intel CPUs, or NEON instructions on ARM CPUs), about 2.7% of the good coefficients are wasted. With $d = 16$ (AVX2 instructions), this goes

up to 5.7%. This loss is a small price to pay compared to the twofold speedup that we get from using twice bigger hardware vectors.

Algorithm 1 PRG based on SPRING using a rejection sampling instantiation

Input: $\ell \geq 0$, d the width of available hardware vectors, the secrets parameters $\tilde{\mathbf{a}} \in R_q$ and $\tilde{\mathbf{s}} \in (R_q)^k$, Fast Fourier evaluation of the secret polynomial $\mathbf{a}$ and $\mathbf{s}$.
Output: An ℓ-bits long sequence of (pseudorandom) $\mathbb{Z}_p$ elements.

```
# Initialization
P̃ ← ã
P ← FFT_128^{-1}(P̃)
L ← ε
i ← 0
j ← 0
size ← 0
while size < ℓ do
    # Extract output
    (v_0, v_1, ..., v_{r-1}) ← Dispatch(P)
    for i = 0 to r − 1 do
        v' ← Rounding(v_i)
        if ⊥ ∉ v' then
            L ← L‖v'
            size ← size + d · log_2(p)
    # Update internal state
    i ← i + 1
    u ← CountTrailingZeroes(i)
    j ← j ⊕ (1 ≪ u)
    if j & (1 ≪ u) ≠ 0 then
        P̃ ← P̃ · s̃_i
    else
        P̃ ← P̃ · s̃_i^{-1}
    P ← FFT_128^{-1}(P̃)
return L
```

We describe the full PRG in algorithm 1. The Dispatch procedure takes a polynomial as input, and dispatch its coefficient as in (1) in $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{r-1}$, for constant parameters r and d.

A Rejection-Sampling based PRF. It is clear that the rejection-sampling process often yields sequence of less than $n \log_2 p$ output bits. This makes it less-than-ideal to implement a PRF, which is always expected to return a specified amount of output bits. However, building a PRF is still possible if we accept a reduction in output size.

With the chosen parameters, we know that at least 96 $\mathbb{Z}_p$ elements survive the erasure process with probability greater than $1 - 2^{-156}$. Therefore, a possible yet inelegant workaround consists in making a PRF that returns the first 96 non-

erased outputs. In the unlikely event that less than 96 truncated coefficients are available, the PRF output is padded with zeroes. The probability of this event is so low that it is undetectable by an adversary.

Implementing such a PRF efficiently is likely to be difficult, because of the amount of bit twiddling and juggling that is involved. Furthermore, unlike the CTR mode, we need to compute the product of many polynomials. To make this implementation efficient, we choose to store the discrete logarithms of the secret polynomials, as it was proposed in [BBL+15] and [BGL+14] so that the subset-product becomes a subset-sum. Each exponentiation by the final summed exponents is computed by a table look-up.

3 Security Analysis

Secure PRF and PRG over a polynomial ring R_q are described in [BPR12], assuming the hardness of Ring-LWE problem. However, for the BPR family to be secure, we need to make two assumptions :

1. The parameter q must be large (exponential in the input length k).
2. The $\mathbf{s}_i$ are drawn from the error distribution of the underlying Ring-LWE instantiation.

In [BBL+15], Banerjee *et al.* show that relaxing those statements do not seem to introduce any concrete attack against the SPRING family, and its security seems to be still very high, even thought SPRING is not provably secure. In our instantiation we slightly weaken SPRING by introducing two changes : (1) the rounding function S we use returns more bits, so more information about the internal state is returned, (2) some coefficients are rejected, so using side-channel timing attacks, an adversary may learn that some coefficients are equal to -1 (the rejected value). However, as we shall discuss in the following part, we do not think that this may undermine the security of SPRING. Finally, we believe that SPRING-RS is actually more secure than the previously proposed variants.

3.1 Birthday-type Attack on SPRING

Let $t < k$ be such that $\log q \simeq 2^t \cdot \log p$. For all $\mathbf{x} \in \{0,1\}^k$, $\mathbf{x}$ can be decomposed in $\mathbf{x} = (\mathbf{w}||\mathbf{z}||x_k)$ with $\mathbf{w}$ t-bit wide and $\mathbf{z}$ $(k-t-1)$-bit wide. We denote by $\mathbf{b_w}$ and $\mathbf{b_z}$ the polynomials $\mathbf{b_w} := \prod_{i=1}^{i=t} \mathbf{s}_i^{x_i}$ and $\mathbf{b_z} := \prod_{i=t+1}^{i=k-1} \mathbf{s}_i^{x_i}$. Then we have:

$$F_{\mathbf{a},\mathbf{s}}(\mathbf{w}||\mathbf{z}||x_k) = S(\mathbf{a} \cdot \mathbf{b_w} \cdot \mathbf{b_z} \cdot \mathbf{s}_k^{x_k}),$$

where $S(\cdot)$ is the rounding function used by the SPRING-RS instantiation. We aim to find two $(k-t-1)$-bit vectors $\mathbf{z}$ and $\mathbf{z}'$ such that $\mathbf{z} \neq \mathbf{z}'$ and $\mathbf{b_z} = \mathbf{b_{z'}}$. Then we will also have $\mathbf{b_z} \cdot \mathbf{s}_k = \mathbf{b_{z'}} \cdot \mathbf{s}_k$.

Notice that $\mathbf{w}$ can take 2^t different values, which are called $\mathbf{w}_0, \ldots \mathbf{w}_{2^t-1}$ according to the Gray Code counter. We denote by G the function that takes as input $(\mathbf{z}||x_k) \in \{0,1\}^{k-t}$, and returns the sequence:

$$G(\mathbf{z}||x_k) := F_{\mathbf{a},\mathbf{s}}(\mathbf{w}_0||\mathbf{z}||x_k)||\dots||F_{\mathbf{a},\mathbf{s}}(\mathbf{w}_{2^t-1}||\mathbf{z}||x_k).$$

If no coefficient is rejected, then the output of G for a given $(\mathbf{z}||x_k)$ is a $(8 \cdot n)$-bit wide sequence ($2^t \log p = \log q = 8$). We want to find a couple $(\mathbf{z}, \mathbf{z}')$ in $\{0,1\}^{k-t-1} \times \{0,1\}^{k-t-1}$ such that $G(\mathbf{z}||x_k) = G(\mathbf{z}'||x_k)$. Then we will have $\mathbf{b}_\mathbf{z} = \mathbf{b}_{\mathbf{z}'}$ with high probability (actually the probability of a false positive is at most $2^{2k}/p^{2^t \cdot n} \simeq 2^{-896}$ for $p = 16$). Then, knowing $G(\mathbf{z}||x_k)$ we are able to predict $G(\mathbf{z}'||x_k)$ with high probability, and this gives us a distinguisher between SPRING-RS and the uniform distribution.

Formally, we first get the sequence generated by 2^{k-1} call to the $F_{\mathbf{a},\mathbf{s}}$ function (i.e. in this case x_k is always 0), and we store all the possible $8 \cdot n$-bit output of G in a hash table, and search for a collision inside it. Knowing that the probability that no rejection has been performed in a given $8 \cdot n$-bit sequence is $(1-1/q)^{2^t \cdot n}$, an adversary can predict that $G(\mathbf{z}'||1)$ would be equal to $G(\mathbf{z}||1)$ for some $\mathbf{z}$ and $\mathbf{z}'$ with advantage about $(1-1/q)^{2 \cdot n} \cdot 2^{2(k-2)}/q^n$ (Probability that no rejection sampling has been performed $\times$ probability of a collision between $G(\mathbf{z}||x_k)$ and $G(\mathbf{z}'||x_k)$), which is around 2^{-900} for the weaker instantiation of SPRING-RS (with $p = 16$).

All in all, this attack require 2^{k-1} call the SPRING function, and if we denote by ℓ the length of the bit-string thus generated. One will need to store $(\ell - 8 \cdot n + 1)$ $8 \cdot n$-bit strings in a hash table. As long as ℓ is bounded by 2^{k+8} (when $p = 16$), the total space required by this attack will be bounded by $\mathcal{O}(n \cdot 2^{k+8})$.

3.2 Lattice Attack

In this attack, we try to find back some of the secrets s_i. We present the attack on a simplified version of SPRING in which $\mathbf{a}$ is known and is equal to 1, then we have :

$$F_{\mathbf{a},s}(x) = S\left(1 \cdot \prod_i \mathbf{s}_i^{x_i}\right),$$

where $\mathbf{x} \neq (0,\dots,0)$. Assume we can apply the SPRING function on any input $\mathbf{x} = (x_1,\dots,x_k)$ which are very sparse, with only two bits i and j are set to one. Consequently, we can write equations by adding an error term e that corresponds to the missing bits.

Let $\sigma := q/p$ and $\mathbf{e}_i$,$\mathbf{b}_i$,$\mathbf{b}_{i,j}$,$\mathbf{e}_{i,j}$ be in R_q. Assume we get $S(\mathbf{s}_i)$ for the input that contains one bit set to one in position i for all i. We call $\mathbf{b}_i$ the output bits and $\mathbf{e}_i$ the missing bits. So, we get the corresponding equations and since we can also write $S(\mathbf{s}_i\mathbf{s}_j)$ for the following equations:

$$\mathbf{s}_i = \mathbf{b}_i + \mathbf{e}_i$$
$$\mathbf{s}_i\mathbf{s}_j = \mathbf{b}_{i,j} + \mathbf{e}_{i,j}$$

where the $\mathbf{e}_i$ values represent the least significant bits and the $\mathbf{b}_i$ values the output of the PRG. We assume here that there is no rejection so that all the $\mathbf{b}_i$ are exactly known. We have the following relation :

$$\mathbf{b}_{i,j} - \mathbf{b}_i\mathbf{b}_j - (\mathbf{s}_i\mathbf{s}_j - \mathbf{s}_i\mathbf{s}_j) = \mathbf{b}_i\mathbf{e}_j + \mathbf{b}_j\mathbf{e}_i + \mathbf{e}_i\mathbf{e}_j - \mathbf{e}_{i,j}. \tag{2}$$

Our goal is to find the $\mathbf{e}_i$ values in order to recover the secret values $\mathbf{s}_i$. We will write in (3) an equation corresponding to a closest vector problem in some lattice. In this equation, we denote by $M_{\mathbf{b}}$ the negacyclic matrix representing the multiplication by $\mathbf{b} = \sum_{i=0}^{n-1} p_i\mathbf{x}^i$ in the polynomial ring, formally it is a linear map defined by the multiplication of $\mathbf{b}$ in R_q in the canonical basis :

$$M_{\mathbf{b}} = \begin{pmatrix} p_0 & p_1 & \cdots & p_{n-1} \\ -p_{n-1} & p_0 & \cdots & p_{n-2} \\ \vdots & \ddots & \ddots & \vdots \\ -p_1 & \cdots & -p_{n-1} & p_0 \end{pmatrix}.$$

Denote by $\mathbf{\Delta}_{i,j}$ the polynomial $\mathbf{\Delta}_{i,j} := \mathbf{b}_{i,j} - \mathbf{b}_i\mathbf{b}_j$ and by $\mathbf{E}_{i,j}$ the polynomial $\mathbf{E}_{i,j} := \mathbf{e}_i\mathbf{e}_j - \mathbf{e}_{i,j}$. Let M be the block-matrix

$$M = \begin{pmatrix} 0 & M_{\mathbf{b}_i} & 0 & M_{\mathbf{b}_j} & 0 \end{pmatrix},$$

such that block i is $M_{\mathbf{b}_i}$, block j is $M_{\mathbf{b}_j}$, and the rest is 0. Then the relation 2 for all $i, j < k$ is equivalent to the system :

$$\begin{pmatrix} q & & & & \\ & \ddots & & M & \\ & & q & & \\ & & & K & \\ & & & & \ddots \\ & & & & & K \end{pmatrix} \cdot \begin{pmatrix} * \\ \mathbf{e}_0 \\ \vdots \\ \mathbf{e}_{k-1} \end{pmatrix} - \begin{pmatrix} \mathbf{\Delta}_{0,1} \\ \vdots \\ \mathbf{\Delta}_{k-2,k-1} \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} \mathbf{E}_{0,1} \\ \vdots \\ \mathbf{E}_{k-2,k-1} \\ K \cdot \mathbf{e}_0 \\ \vdots \\ K \cdot \mathbf{e}_{k-1} \end{pmatrix}, \tag{3}$$

with K chosen so that $\mathbf{E}_{i,j}$ and $K \cdot \mathbf{e}_i$ are about the same size, namely $K \approx \sigma \cdot \sqrt{n}$. To find the $\mathbf{e}_i$ we have to enumerate the lattice points of $\mathcal{L}$ in a ball of radius at least $\sigma^2 n/12\sqrt{k(k+1)/2}$. We estimate the number of false positives using the Gaussian heuristic :

$$(\sigma^2 n/12\sqrt{k(k-1)/2}\sqrt{4\pi \mathrm{e}/n/k/(k+1)})^{k(k+1)n/2}/\det(\mathcal{L}) = \\ (\sigma^2\sqrt{2\pi \mathrm{e}(k-1)/(k+1)n}/12)^{k(k+1)n/2} q^{-k(k-1)n/2} \sigma^{-kn} n^{-kn/2}$$

For our parameters with $p = 16$, the actual value is larger than $\approx 2^{1334}$ for all k and K, which is reached in $k = 2$, $K = 52$.

In fact, it is easy to see that for $\sigma^2\sqrt{n} \leq q$, there exist a k where we expect no false positives ; but for q a constant times smaller, the number of false positives becomes $2^{\Omega(n)}$.

In the case one wants to attack the SPRING-RS PRF, this attack is even weaker, as only 96 coefficients of the $\mathbf{s}_i$ polynomials are returned, whether there is rejection sampling or not.

This attack can be adapted to SPRING-RS with $\mathbf{a}$ unknown, however, there would be a loss of efficiency since we need to consider the products of three terms.

The version of SPRING-RS attacked here is trivially weaker than one with an unknown $\mathbf{a}$, and the attack has not been found to be efficient. Therefore, we do not detail further this case.

3.3 Partially Known Secret Key.

In this part we assume that all $\mathbf{s}_i$ are known or have already been found (which is very unlikely) and we are trying to find back the secret $\mathbf{a}$. Although we do not guarantee the security of SPRING when the $\mathbf{s}_i$ are known, we show here, that the system is resistant against lattice-reduction attacks and BKW attack with a small enough p.

Let $\mathbf{x} = (x_1, \ldots, x_k)$ be in $\{0,1\}^k$, if the $\mathbf{s}_i$ are known (even though they are not supposed to be) then the product $\mathbf{b_x} := \prod_{i=1}^{k} \mathbf{s}_i^{x_i}$ is known. The goal is to find back $\mathbf{a}$ knowing $(\mathbf{b_x}, S(\mathbf{a} \cdot \mathbf{b_x}))$. So we have to solve the search Ring-LWR problem [LPR13]. An attack on SPRING-BCH is described in [BBL+15] using lattice reduction, with complexity greater than 2^{430} in time and greater than 2^{160} in space. In fact, the BKW attack appears to be more efficient. The table 2 gives the complexity of the best attack (which is always BKW, since the noise is large), for each $p \in \{2, 4, 8, 16\}$.

Table 2. Complexity of attacking SPRING assuming the $\mathbf{s}_i$ are known, using the BKW algorithm

p	2	4	8	16
BKW	2^{163}	2^{124}	2^{97}	2^{78}

3.4 Side-channel leaks

An obvious drawback of rejection sampling is the irregular rate at which output is produced. It is conceivable that a timing attack could reveal some information about the internal state of the PRG. If we were optimistic about the attacker's side-channel capabilities, we could assume that she knows exactly what coefficients are rejected. We thus assume that each time a coefficient is rejected, both its location and the value of the counter x leak.

Therefore, the attacker has access to equations of the form $\left(\mathbf{a}\prod_i \mathbf{s}_i^{x_i}\right)_j = -1$ over $\mathbb{Z}_q$ where the unknowns are the coefficients of $\mathbf{a}$ and $\mathbf{s}_i$. Denote by $HW(x)$ the Hamming weight of x, then each of these equations can be converted into a polynomial of degree $1 + HW(x)$ over the coefficients of $\mathbf{a}$ and all $\mathbf{s}_i$.

If the first $2^i n \log_2 p$ key-stream bits are observed, then we expect $n2^i/q$ coefficients to be rejected. This yields this many polynomial equations in $n(i+1)$ variables, of which $n\binom{i}{d-1}/q$ are expected to be of degree d. With the chosen parameters, $i \geq 12$ is needed to obtain more equations than unknowns. With

$i = 12$, we obtain the smallest possible over-determined system, with 2032 polynomial equations in 1664 unknowns, of degree mostly larger than 2. Note that the ideal spanned by these polynomial is not guaranteed to be zero-dimensional. No known technique is capable of solving arbitrary systems of polynomial equations of this size. In particular, the complexity of computing a Gröbner basis of this many equations can be roughly estimated [Fau99, BFS04]: it is about 2^{2466}.

When i grows up to 64, the system becomes more over-determined: with $i = k = 64$, the largest possible value, we obtain 2^{63} equations in 8320 variables, with degrees up to 65. Storing this amount of data is completely unpractical. Neglecting this detail, we argue that a Gröbner basis computation will crunch polynomials of degree larger than 12: there are 2^{128} monomials of degree 12 in 8320 variables and only 2^{114} degree-12 multiples of the input equations (the computation generically stops when these two quantities match).

As such, we expect any Gröbner basis computation to perform, amongst others, the reduction to row echelon form of a sparse matrix of size $2^{114} \times 2^{128}$, a computationally unfeasible task.

4 Implementation details

We implemented our variant of SPRING using SIMD instructions, which enabled us to perform given operations on multiple data in parallel. We propose two implementations of our scheme. The first one uses the 128-bit wide SIMD hardware vectors available in SSE2 or NEON instructions while the second one uses 256-bit wide SIMD hardware vectors and AVX2 instructions.

Most of our implementation tricks are borrowed from [BBL+15], and some of our implementations reuse parts of the code of previous SPRING variants. We refer the reader to [BBL+15] for more details. Our only innovation is the implementation of the rejection-sampling process, which is straightforward, as well as an implementation of fast polynomial multiplication using AVX2 instructions, that we describe next.

The problem comes down to computing a kind of FFT of size 128 modulo 257. We store $\mathbb{Z}_{257}$ in 16-bit words, in zero-centered representation. Using SSE2 instructions, and hardware vector registers holding 8 coefficients, a reasonable strategy is to perform one step of Cooley-Tukey recursive division, after which two size-64 FFTs have to be computed. This is done efficiently by viewing each input as an 8×8 matrix, performing 8 parallel size-8 FFTs on the rows, multiplying by the twiddle factors, transposing the matrix, and finally performing 8 parallel FFTs. The use of vector registers allows to perform the 8 parallel operations efficiently.

When AVX2 instructions are available, we have access to vector registers holding 16 coefficients. Several strategies are possible, and we describe the one we actually implemented. We view the input of a size-128 FFT as a 16×8 matrix. We perform 16 parallel size-8 FFTs on the rows, which is easy using the larger vector registers. Transposing yields a 8×16 matrix, and we need to perform 8 parallel size-16 FFTs on its rows.

This is the non-obvious part. To make full use of the large vector registers, we decided to store two rows in each vector register. Because the first pass of this size-16 FFT requires operation between adjacent rows, a bit of data juggling is necessary. We store rows 0 and 8 in the first register, rows 1 and 9 in the second, etc. This is done with the `VPERM2I128` instruction. Because the last pass requires operations between rows i and $i + 8$, which is again not possible if they are in the same register, we perform the same data-juggling operation again. This puts the rows back in the right order.

Performing the rejection sampling is easy. With AVX2 instructions, we use the `VPCMPEQW` to perform a coefficient-wise comparison with $(-1, \ldots, -1)$, and a `VPMOVMSKB` to extract the result of the comparison into a 16-bit integer. It is slightly different on an ARM processor with the NEON instruction set as there is nothing like `VPMOVMSKB`. However, we achieve the same result, using some tricks. We first convert the `int16x8_t` NEON vector into a `int8x16_t` NEON vector, and then we `ZIP` the low and the high part of this vector. This gives two `int8x8_t` NEON vector, `d0` and `d1`. We use the `VSRI` on `d0` and `d1` with constant 4, then we transfer the low and the high part of the obtained vector in ARM registers. Then, we obtain what we need using shift and xor. When only 128-bit vectors are available, the function is easier to program with $d = 8$ (where d is the vector width), whereas $d = 16$ is slightly more programmer-friendly when 256-bit vectors are available. This is not a very hard constraint though, as both values of d can be dealt with efficiently using both instructions sets.

5 Conclusion

In this paper, we propose to use the rejection sampling as a technique to cancel the bias in the SPRING PRF and PRG. We revisit the attack on SPRING-CRT and find new attacks on SPRING. Finally, our experimentation shows that this leads to a very efficient stream cipher, whose security is very high. We think that lattice-based cryptography can be used with high performance.

References

AKPW13. Joël Alwen, Stephan Krenn, Krzysztof Pietrzak, and Daniel Wichs. Learning with rounding, revisited - new reduction, properties and applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 57–74. Springer, Heidelberg, August 2013.

BBL+15. Abhishek Banerjee, Hai Brenner, Gaëtan Leurent, Chris Peikert, and Alon Rosen. SPRING: Fast pseudorandom functions from rounded ring products. In Carlos Cid and Christian Rechberger, editors, *FSE 2014*, volume 8540 of *LNCS*, pages 38–57. Springer, Heidelberg, March 2015.

BFS04. Magali Bardet, Jean-Charles Faugere, and Bruno Salvy. On the complexity of gröbner basis computation of semi-regular overdetermined algebraic equations. In *Proceedings of the International Conference on Polynomial System Solving*, pages 71–74, 2004.

BGL+14. Hai Brenner, Lubos Gaspar, Gaëtan Leurent, Alon Rosen, and François-Xavier Standaert. FPGA implementations of SPRING - and their countermeasures against side-channel attacks. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 414–432. Springer, Heidelberg, September 2014.

BGM+15. Andrej Bogdanov, Siyao Guo, Daniel Masny, Silas Richelson, and Alon Rosen. On the hardness of learning with rounding over small modulus. Cryptology ePrint Archive, Report 2015/769, 2015. `http://eprint.iacr.org/2015/769`.

BPR12. Abhishek Banerjee, Chris Peikert, and Alon Rosen. Pseudorandom functions and lattices. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 719–737. Springer, Heidelberg, April 2012.

CLS06. Scott Contini, Arjen K. Lenstra, and Ron Steinfeld. VSH, an efficient and provable collision-resistant hash function. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 165–182. Springer, Heidelberg, May / June 2006.

Fau99. Jean-Charles Faugere. A new efficient algorithm for computing gröbner bases (f 4). *Journal of pure and applied algebra*, 139(1):61–88, 1999.

Gal12. François Le Gall. Faster algorithms for rectangular matrix multiplication. In *53rd FOCS*, pages 514–523. IEEE Computer Society Press, October 2012.

LBF08. Gaëtan Leurent, Charles Bouillaguet, and Pierre-Alain Fouque. SIMD Is a Message Digest. Submission to NIST, 2008.

LMPR08. Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFT: A modest proposal for FFT hashing. In Kaisa Nyberg, editor, *FSE 2008*, volume 5086 of *LNCS*, pages 54–72. Springer, Heidelberg, February 2008.

LPR13. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)*, 60(6):43, 2013.

Lyu09. Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 598–616. Springer, Heidelberg, December 2009.

MS95. Willi Meier and Othmar Staffelbach. The self-shrinking generator. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 205–214. Springer, Heidelberg, May 1995.

Reg05. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.

A Attack on SPRING-CRT

In SPRING-CRT, the authors of [BBL+15] used a modulus $\tilde{q} = 2 \cdot q$ so that a uniform value over $\mathbb{Z}_{\tilde{q}}$ can be easily transformed into a uniform value over $\mathbb{Z}_2$ using the most significant bits. However, it introduced a weakness : if we control the product over $\mathbb{Z}_2$, then the most significant bit is again biased.

Furthermore, since n is a power of two, we have over $\mathbb{F}_2$ the factorization $x^n + 1 = (x+1)^n$ and we can use this property as follows. Let κ be a power of two. Suppose we have $\mathbf{x}$ and $\mathbf{x}'$ such that for a certain index i_0, there exist i such that $x_i \neq x'_i$ and $\prod_{i \geq i_0} \mathbf{s}_i^{x_i - x'_i} = 1 \bmod (2, x^\kappa + 1)$. Furthermore, if for all $i < i_0$, $x_i = x'_i$, we have $\prod_i \mathbf{s}_i^{x_i - x'_i} = 1 \bmod (2, \mathbf{x}^\kappa + 1)$. Now, observe that the bias of $\lfloor c + 2u \rceil_2$ for a uniform $u \in \mathbb{Z}_q$ and constant c is $(-1)^c/q$ when q is odd. Therefore, assuming $\prod_i \mathbf{s}_i^{x_i} \bmod q$ is uniform when $(x_i)_{i<i_0}$ is taken uniformly and with the same condition over $\mathbf{x}'$, we have for all $0 < t < \kappa$:

$$bias\Big(\sum_{j=0}^{n/\kappa-1} \big(\lfloor a \prod_i \mathbf{s}_i^{x_i} \rceil_2 + \lfloor a \prod_i \mathbf{s}_i^{x'_i} \rceil_2\big)_{t+j\kappa}\Big) = q^{-2n/\kappa},$$

while if $\prod_i \mathbf{s}_i^{x_i - x'_i} \neq 1 \bmod (2, x^\kappa + 1)$ the other bias are null.

Then, we choose $i_0 \approx 4n/\kappa \log_2(q) + 2\ln(\kappa)$ and asks for $2^{\kappa/2} 2^{i_0}/\kappa$ blocks of values. After computing the sums over n/κ bits, it remains to find a correlation between all pairs of $2^{\kappa/2}$ vectors of 2^{i_0} bits. While [BBL+15] computed all of them naively, we can view this as multiplying matrices whose coefficient are either -1 or 1. Using the Hoeffding lemma, we conclude that we have a constant advantage.

In our case, we choose $\kappa = 64$ so that $i_0 = 72$ and the bottleneck is the multiplication of a matrix with 2^{32} rows and 2^{72} columns by its transpose.

Asymptotically, when there is $n^{72/32}$ columns for n rows, this takes time at most $O(n^{3.49})$ [Gal12]. Removing the Landau notation, this indicates (if 2^{72} is sufficiently large [5]) a time of roughly 2^{104} operations. The exact function is still unclear, but the complexity is certainly less than 2^{72-32} square matrix multiplications of size 2^{32}. Using Strassen algorithm for the square matrix multiplication gives a total of around 2^{130} operations [6].

If k is too small, then we can ask only for 2^k blocks, so that there are $\kappa 2^{k-i_0}$ rows in the matrix and the advantage $\kappa^2 2^{2(k-i_0)-\kappa}$. For $k = 64$, we choose $\kappa = 128$, $i_0 = 42$ so that there are 2^{45} rows in the matrix and the advantage is $\approx 2^{-38}$ for a complexity with Strassen's algorithm of $2^5 7^{42} \approx 2^{123}$.

Using $\kappa = \Theta(\sqrt{n \log(q)})$ for $k \geq O(\sqrt{n \log(q)})$, we get a complexity of $2^{O(\sqrt{n \log(q)})}$. This sub-exponential attack makes us wonder about the security of Ring-LWE with an even modulus.

[5] Since 3.49 is strictly above the number given in [Gal12], for any $\epsilon > 0$, the complexity of a multiplication is less than $\epsilon n^{3.49}$ for all sufficiently large n.

[6] [BBL+15] used $i_0 = 4n/\kappa \log_2(q)$ so that their claimed complexity is 2^{126} but this is not enough to get a constant advantage.