# SPRING: Fast Pseudorandom Functions from Rounded Ring Products

Abhishek Banerjee[1]$^{\star}$, Hai Brenner[2]$^{\star\star}$, Gaëtan Leurent[3], Chris Peikert[1]$^{\star\star\star}$, and Alon Rosen[2]$^{\dagger}$

[1] Georgia Institute of Technology
[2] IDC Herzliya
[3] INRIA Team SECRET

**Abstract.** Recently, Banerjee, Peikert and Rosen (EUROCRYPT 2012) proposed new theoretical pseudorandom function candidates based on "rounded products" in certain polynomial rings, which have rigorously provable security based on worst-case lattice problems. The functions also enjoy algebraic properties that make them highly parallelizable and attractive for modern applications, such as evaluation under homomorphic encryption schemes. However, the parameters required by BPR's security proofs are too large for practical use, and many other practical aspects of the design were left unexplored in that work.

In this work we give two concrete and practically efficient instantiations of the BPR design, which we call SPRING, for "subset-product with rounding over a ring." One instantiation uses a generator matrix of a binary BCH error-correcting code to "determinstically extract" nearly random bits from a (biased) rounded subset-product. The second instantiation eliminates bias by working over suitable moduli and decomposing the computation into "Chinese remainder" components.

We analyze the concrete security of these instantiations, and provide initial software implementations whose throughputs are within small factors (as small as 4.5) of those of AES.

**Keywords:** pseudorandom functions, noisy learning problems, learning with rounding, lattices

$^{\star}$ Research supported by the fourth author's grants.

$^{\star\star}$ Research supported by the ERC under the EU's Seventh Framework Programme (FP/2007-2013) ERC Grant Agreement n. 307952.

$^{\star\star\star}$ This material is based upon work supported by the National Science Foundation under Grant CNS-0716786 and CAREER Award CCF-1054495, by the Alfred P. Sloan Foundation, by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under Contract No. FA8750-11-C-0098, and by BSF grant 2010296. The views expressed are those of the authors and do not necessarily reflect the official policy or position of the National Science Foundation, the Sloan Foundation, DARPA or the U.S. Government, or the BSF.

$^{\dagger}$ Research supported by BSF grant 2010296 and by the ERC under the EU's Seventh Framework Programme (FP/2007-2013) ERC Grant Agreement n. 307952.

## 1 Introduction

Pseudorandom functions (PRFs) are fundamental objects in symmetric cryptography, which are frequently used in the construction of higher-level objects like block ciphers, message authentication codes, and encryption. A PRF takes as arguments a "key" and an input string of a particular length, and deterministically produces an output string of a particular (possibly different) length. Informally, over the choice of a random (and secret) key that is used for all inputs, a PRF cannot be efficiently distinguished from a truly random function via adaptive oracle (i.e., "black-box") access.

Constructions of PRFs have historically fallen into two broad classes: theoretically sound designs that admit security proofs under mathematically natural intractability assumptions (e.g., [BM82,BBS86,GGM84,NR95,NR97,NRR00]), and designs whose conjectured security is supported by the lack of any known effective cryptanalytic attack (e.g., DES [DES77], AES [DR00], and countless others). Constructions of the latter type can be very fast and are dominant in practice, but their internal complexity and lack of rigorous security reductions may increase their risk of succumbing to new kinds of attacks, especially for very strong and subtle security notions like PRFs. And while theoretically sound designs tend to be mathematically elegant, they have so far been far too inefficient or otherwise impractical for real-world use.

Mathematical simplicity is sometimes viewed as a weakness in terms of security, because underlying "structure" can sometimes be used as a lever for non-trivial attacks. On the other hand, a number of recently proposed applications of PRFs, requiring properties such as efficient homomorphic evaluation [ACPR13] or "key-homomorphism" [BLMR13], demonstrate that PRFs with algebraic structure can deliver significant performance advantages over those that lack such structure (in some cases offering an improvement of several orders of magnitude; see for instance [ACPR13] vs. [GHS12,CCK$^+$13]).

At the heart of these recent developments is a new class of candidate PRFs constructed by Banerjee, Peikert and Rosen (BPR) [BPR12], which enjoy useful algebraic structure, and may help to bridge the gap between theoretical soundness and practical efficiency. The BPR constructions are based on the pseudorandomness properties of *rounded subset-products* in suitable polynomial rings. BPR give evidence for the asymptotic security of their constructions, proving them secure under well-studied hardness assumptions like "learning with errors" $\mathsf{LWE}$ [Reg05,Pei09] and its ring variant ring-$\mathsf{LWE}$ [LPR10], and by implication, worst-case problems on point lattices.

### 1.1 The SPRING Family of Pseudorandom Functions

One of the main constructions in [BPR12] is a class of PRF candidates that we call SPRING, which is short for "**s**ubset-**p**roduct with **r**ounding over a **ring**." Let $n$ be a power of two, and let $R$ denote the polynomial ring $R := \mathbb{Z}[X]/(X^n + 1)$, which is known as the $2n$th *cyclotomic ring*.[4] For a positive integer $p$, let $R_p$ denote the quotient ring

$$R_p := R/pR = \mathbb{Z}_p[X]/(X^n + 1),$$

i.e., the ring of polynomials in $X$ with coefficients in $\mathbb{Z}_p$, where addition and multiplication are modulo both $X^n + 1$ and $p$. (For ring elements $r(X)$ in $R$ or $R_p$, we usually suppress the indeterminate $X$.) We often identify $r \in R_p$ with the vector $\mathbf{r} \in \mathbb{Z}_p^n$ of its $n$ coefficients in some canonical order. Let $R_p^*$ denote the multiplicative group of units (invertible elements) in $R_p$.

For a positive integer $k$, the SPRING family is the set of functions $F_{a,\boldsymbol{s}} : \{0,1\}^k \to \{0,1\}^m$ indexed by a unit $a \in R_p^*$ and a vector $\boldsymbol{s} = (s_1, \ldots, s_k) \in (R_p^*)^k$ of units. The function is defined as the "rounded subset-product"

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

where $S\colon R_p \to \{0,1\}^m$ for some $m \le n$ is an appropriate "rounding" function. For example, BPR uses the rounding function $\lfloor \cdot \rceil_2 \colon R_p \to R_2 \equiv \mathbb{Z}_2^n$ that maps each of its input's $n$ coefficients to $\mathbb{Z}_2 = \{0,1\}$, depending on whether the coefficient is closer modulo $p$ to 0 or to $p/2$. (Formally, each coefficient $b \in \mathbb{Z}_p$ is mapped to $\lfloor \frac{2}{p} \cdot b \rceil \in \mathbb{Z}_2$.)

It is proved in [BPR12] that when $a$ and the $s_i$ are drawn from appropriate distributions, and $p$ is sufficiently large, the above function family is a secure PRF family assuming that the "ring learning with errors" (ring-LWE) problem [LPR10] is hard in $R_p$. This proof is strong evidence that the family has a sound design and is indeed a secure PRF, at least in an asymptotic sense. The intuition behind the security argument is that the rounding hides all but the most-significant bits of the product $a \cdot \prod_i s_i^{x_i}$, and the rounded-off bits can be seen as a kind of "small" error (though one that is generated deterministically from the subset-product, rather than as an independent random variable as in the LWE problem). And indeed, (ring-)LWE and related "noisy learning" assumptions state that noisy products with secret ring elements are indistinguishable from truly uniform values.

We stress that the *known proof* of security (under ring-LWE) requires the modulus $p$ to be very large, i.e., exponential in the input length $k$. Yet as discussed in [BPR12], the large modulus appears to be an artifact of the proof technique, and the family appears not to *require* such large parameters for concrete security. Indeed, based on the state of the art in attacks on "noisy learning" problems like (ring-)LWE, it is reasonable to conjecture that the SPRING functions can be secure for rather small moduli $p$ and appropriate rounding functions (see Section 4 for further details).

[4] It is the $2n$th cyclotomic ring because the complex roots of $X^n + 1$ are all the $2n$th primitive roots of unity. The BPR functions can be defined over other cyclotomic rings as well, but in this work we restrict to powers of two for simplicity and efficiency.

### 1.2 Our Contributions

In this work we give two new, optimized instantiations of the SPRING PRF family for parameters that offer high levels of concrete security against known classes of attacks, and provide very high-performance software implementations.

Because we aim to design practical functions, we instantiate the SPRING family with relatively small moduli $p$, rather than the large ones required by the theoretical security reductions from [BPR12]. This allows us to follow the same basic construction paradigm as in [BPR12], while taking advantage of the fast integer arithmetic operations supported by modern processors. We instantiate the parameters as various (but not all) combinations of

$$n = 128, \quad p \in \{257, 514\}, \quad k \in \{64, 128\},$$

which (as explained below in Section 1.3) yields attractive performance, and allows for a comfortable margin of security. The choice of modulus $p \in \{257, 514\}$ is akin to the one made in SWIFFT, for a practical instantiation of a theoretically sound lattice-based collision-resistant hash function [LMPR08]. Also as in SWIFFT, our implementations build on Fast Fourier Transform-like algorithms modulo $q = 257$.

Working with small moduli $p$ requires adjusting the rounding function $S(\cdot)$ in the SPRING construction so that its output on a uniformly random element of $R_p^*$ does not have any noticeable bias (which otherwise would clearly render the function insecure as a PRF). We use rounding functions of the form $S(b) = G(\lfloor b \rceil_2)$, where $\lfloor \cdot \rceil_2 \colon R_p \to R_2$ is the usual coefficient-wise rounding function that provides (conjectured) indistinguishability from a potentially *biased* random function, and $G \colon R_2 \to \{0,1\}^m$ for some $m \leq n$ is an appropriate post-processing function that reduces or removes the bias. (In [BPR12], $G$ is effectively the identity function, because a huge modulus $p$ ensures no noticeable bias in the rounded output.)

For each value of the modulus $p \in \{257, 514\}$ we have a different concrete instantiation, which we respectively call SPRING-BCH and SPRING-CRT. These instantiations differ mainly in the computation of the subset-products in $R_p^*$, and in the definition of the bias-reducing function $G$.

*SPRING-BCH.* In this instantiation, we use an odd modulus $p = q = 257$, which admits very fast subset-product computations in $R_q^*$ using Fast Fourier Transform-type techniques (as mentioned above). However, because $p$ is odd, the usual rounding function $\lfloor \cdot \rceil_2 \colon R_p \to R_2$ has bias $1/q$ on each of the output coefficients (bits). To reduce this bias, the function $G$ multiplies the 128-dimensional, $1/q$-biased bit vector by the $64 \times 128$ generator matrix of a binary (extended) BCH error-correcting code with parameters $[n, m, d] = [128, 64, 22]$, yielding a syndrome with respect to the dual code. This simple and very fast "deterministic extraction" procedure (proposed in [AR13]) reduces the bias exponentially in the distance $d = 22$ of the code, and yields a 64-dimensional vector that is $2^{-145}$-far from uniform when applied to a 128-dimensional bit vector of independent $1/q$-biased bits. However, this comes at the cost of outputting $m = 64$ bits instead of $n = 128$, as determined by the rate $m/n$ of the code.

*SPRING-CRT.* In this instantiation, we use an even modulus $p = 2q = 514$, and decompose the subset-product computation over $R_{2q}^*$ into its "Chinese remainder" components $R_2^*$ and $R_q^*$. For the $R_q^*$ component we use the same evaluation strategy as in SPRING-BCH, but for fast subset-products in the $R_2^*$ component we need new techniques. We prove that the multiplicative group $R_2^*$ decomposes into $n/2$ small cyclic groups, having power-of-two orders at most $n$. We also give explicit "sparse" generators for these cyclic components, and devise fast algorithms for converting between the "cyclic" representation (as a vector of exponents with respect to the generators) and the standard polynomial one. These tools allow us to transform a subset-product in $R_2^*$ into a subset-*sum* of vectors of (small) exponents with respect to the generators, followed by one fast conversion from the resulting vector of exponents to the polynomial it represents.

For rounding $R_{2q}^*$, we show that standard rounding of a uniformly random element of $R_{2q}^*$ to $R_2$ directly yields $n-1$ independent and unbiased bits, so our function $G$ simply outputs these bits. The main advantage over SPRING-BCH is the larger output size (almost twice as many bits), and hence larger throughput, and in the simpler and tighter analysis of the bias. On the other hand, we also show that the CRT decomposition of $R_{2q}^*$ can be exploited somewhat in attacks, by effectively canceling out the $R_2^*$ component and recognizing the bias of the rounded $R_q^*$ component. Fortunately, for our parameters the best attacks of this type appear to take almost $2^{128}$ bit operations, and around $2^{119}$ space.

We refer to Sections 2 and 3 for further details on these instantiations, and to Section 4 for a concrete security analysis.

### 1.3 Implementations and Performance

We implement the two variants of SPRING described above, both for standalone evaluations on single inputs, and in a counter-like (CTR) mode that is able to amortize much of the work across consecutive evaluations. For the counter itself we use the Gray code, which is a simple way of ordering the strings in $\{0,1\}^k$ so that successive strings differ in only one position. Then when running SPRING in counter mode, each successive subset-product can be computed from the previous one with just one more multiplication by either a seed element or its inverse. More precisely, we store the currently computed subset-product $b := a \prod_{i=1}^{n} s_i^{x_i}$. (The Gray code starts with $0^k$, so the initial subset-product is simply $a$.) If the next input $x'$ flips the $i$th bit of $x$, then we update the old subset-product to $b' = b \cdot s_i$ if $x_i = 0$, otherwise $b' = b \cdot s_i^{-1}$.

For the SPRING-CRT instantiation, which works in $R_{2q}^* \cong R_2^* \times R_q^*$, we use two methods for computing (subset-)products in the $R_2^*$ component. The first uses the cyclic decomposition of $R_2^*$ as described above, and is the fastest method we have found for computing a standalone subset-product "from scratch." The other method uses the native "carryless polynomial multiplication" (PCLMUL) instruction available in recent Intel processors, and/or precomputed tables, for single multiplications in the Gray code counter mode.

We benchmarked our implementations on a range of CPUs with several different microarchitectures. As a point of reference, we use the highly optimized

AES benchmarks from eBACS [eBA], and the bitsliced implementation for Käsper and Schwabe [KS09]. We report our performances measures in Table 1, using high-end desktop processors (Core i7), and small embedded CPUs found in tablets and smart-phones (Atom and ARM Cortex). Even though the architectures of those machine are quite different, our results are very consistent: in counter mode, SPRING-BCH is between 8 and 10 times slower than AES (as measured by output throughput), while SPRING-CRT is about 4.5 times slower than AES (disregarding AES implementation with AES-NI when they are available). We expect similar results on other CPUs with similar SIMD engines. Finally, we mention that the very latest Intel CPUs (Haswell microarchitecture) include a new 256-bit wide SIMD engine with support for integer operations (AVX2). We expect that an AVX2 implementation of SPRING would run about twice as fast on those processors, yielding very compelling performance.

**Table 1.** Implementation results for SPRING-BCH and SPRING-CRT with $n = 128$, in both standalone and Gray code counter mode (CTR). Speeds are presented in processor cycles per output byte, and are compared with the best known AES implementations.

| | SPRING-BCH | | SPRING-CRT | | AES-CTR | |
|---|---|---|---|---|---|---|
| | Standalone | CTR | Standalone | CTR | w/o AES-NI | w/AES-NI |
| ARM Cortex A15 | 220 | 170 | 250 | 77 | 17.8 | N/A |
| Atom | 247 | 137 | 235 | 76 | 17 | N/A |
| Core i7 Nehalem | 74 | 60 | 76 | 29.5 | 6.9 | N/A |
| Core i7 Ivy Bridge | 60 | 46 | 62 | 23.5 | 5.4 | 1.3 |

### 1.4 Organization

The rest of the paper is organized as follows. We discuss the details of the SPRING-BCH and SPRING-CRT instantiations in Sections 2 and 3 respectively. We follow by analyzing the concrete security of our instantiations against known attacks in Section 4.1, expanding on one class of combinatorial attacks on SPRING-CRT in Section 4.2. Finally, we present certain implementation details and code optimizations in Section 5.

## 2 SPRING-BCH

Here we describe our first instantiation, SPRING-BCH, which works over $R_q^*$ for a suitable prime $q$, and uses a BCH code for reducing the bias of the rounded subset-product.

### 2.1 Fast Subset Product in $R_q$

Efficient operations in the ring $R_q$ were given in prior work by Lyubashevsky *et al.* [LMPR08] (following [Mic02,PR06,LM06]). They give a Chinese Remainder decomposition of this ring as $R_q \cong \mathbb{Z}_q^n$, for prime $q = 1 \pmod{2n}$, and gave fast FFT-like algorithms for converting between (the standard polynomial representation of) $R_q$ and $\mathbb{Z}_q^n$. In particular, the multiplicative group of units $R_q^*$ is isomorphic to $(\mathbb{Z}_q^*)^n$. Since $\mathbb{Z}_q^*$ is cyclic and of order $q-1$, a subset-product in $R_q$ reduces to a subset-sum of $n$-dimensional vectors of exponents modulo $q-1$ (with respect to some generator of $\mathbb{Z}_q^*$). Once the final vector of exponents have been computed, the corresponding element in $\mathbb{Z}_q^n$ can be computed by table lookups, and finally converted to its polynomial representation via the FFT-like algorithm from [LMPR08].

### 2.2 Rounding via BCH Code

Since $q$ is odd, the usual rounding function $\lfloor\cdot\rceil_2 \colon R_q \to R_2$, when applied to a random input in $R_q$, outputs a ring element in $R_2$ whose (bit) coefficients are independent and have bias $1/q$. In this subsection we define a function $G \colon R_2 \to \{0,1\}^m$ that dramatically reduces this bias using a BCH code.

**Definition 1 ([NN90]).** *The* bias *of a distribution $X \in \{0,1\}^m$ with respect to $I \subseteq [m]$ is defined as*

$$bias_I(X) = \left|\Pr\Big[\bigoplus_{i\in I} x_i = 0\Big] - \Pr\Big[\bigoplus_{i\in I} x_i = 1\Big]\right|.$$

*Let max-bias$(X)$ denote the maximal bias of $X$ over all nonempty $I \subseteq [m]$.*

**Theorem 1 ([NN90]).** *Let $X \in \{0,1\}^m$ be a random variable. Then*

$$2 \cdot \Delta(X, U_m) \leq \sqrt{2^m} \cdot max\text{-}bias(X)$$

*where $\Delta(X, U_m)$ denotes the statistical difference of $X$ from the uniform distribution on $m$ bits.*

**Proposition 1 ([AR13]).** *Let $G$ be a generator matrix of a binary linear code with parameters $[n, m, d]$, and let $D \in \{0,1\}^n$ be a distribution of independent bits such that $bias_{\{i\}}(D) \leq \epsilon$ for every $i \in [n]$. Then max-bias$(G \cdot D) \leq \epsilon^d$.*

From the above we get that when applied to a random input $b \in R_q$, the statistical distance of the distribution $S(b)$ from uniform is at most $(1/q)^d\sqrt{2^m}/2$. Note that in SPRING-BCH, we are actually applying $G$ to $\lfloor b\rceil_2$ for a random *unit* $b \in R_q^*$, in which case the coefficients of $\lfloor b\rceil_2$ are not quite independent. Since we are anyway only heuristically modeling the subset-products as uniformly random and independent, we believe that it is safe to heuristically assume that $G$ provides low bias in our instantiation.

In terms of implementation, generator matrices of BCH codes over $GF(2)$ are preferable, since the rows of the matrix are cyclic shifts of a single row, which facilitates fast implementation. We note that $n$ is a power of 2, and any BCH code over $GF(2)$ is of length $2^t - 1$ for some integer $t$. To make the matrix compatible with an $n$ that is a power of two, we use the extended-BCH code, which is obtained in a standard way by appending a parity bit to the codewords, and increases the code distance $d$ by one. We finally note that for our chosen parameters $n = 128, m = 64$, the BCH code with parameters $[127, 64, 21]$ and its extension with parameters $[128, 64, 22]$ have the largest known minimum distance for these specific rates.

## 3 SPRING-CRT

We now describe our second instantiation, called SPRING-CRT, which uses *unbiased* rounding on an *even* modulus of the form $p = 2q$, where $q$ is an odd prime as in the instantiation from the previous section.

By the Chinese Remainder Theorem, the natural ring homomorphism $R_{2q} \to R_2 \times R_q$ is a ring isomorphism, and moreover, there is an explicit map which lets us convert back and forth between the two representations. Specifically, it is easy to verify that the pair $(b_2, b_q) \in R_2 \times R_q$ corresponds to

$$b = q \cdot \bar{b}_2 + (q+1) \cdot \bar{b}_q \pmod{2q} \tag{2}$$

for arbitrary $\bar{b}_2, \bar{b}_q \in R_{2q}$ such that $\bar{b}_2 = b_2 \pmod 2$ and $\bar{b}_q = b_q \pmod q$. The CRT isomorphism also induces a group isomorphism between $R_{2q}^*$ and $R_2^* \times R_q^*$, and thus lets us represent the seed elements and their subset-products as pairs in $R_2^* \times R_q^*$. We compute products in the $R_q^*$ component as detailed in Section 2.1 above. In the following subsections, we define an unbiased rounding function from $R_{2q}^*$ to $R_2$, and give fast algorithms for computing products in the $R_2^*$ component.

### 3.1 Unbiased Rounding of $R_{2q}^*$

We start by describing how the rounding function from $R_{2q}$ to $R_2$ can be computed directly from the Chinese remainder components $(b_2, b_q) \in R_2 \times R_q$ of a given $b \in R_{2q}$. As above, let $\bar{b}_q, \bar{b}_2 \in R_{2q}$ denote arbitrary mod-$2q$ representatives of $b_2, b_q$. By Equation (2) and the definition of the rounding function $\lfloor \cdot \rceil \colon R_{2q} \to R_2$,

$$\lfloor b \rceil_2 = \lfloor q(\bar{b}_q + \bar{b}_2) + \bar{b}_q \rceil_2 = \lfloor (\bar{b}_q + \bar{b}_2) + \bar{b}_q / q \rceil.$$

If we choose the coefficients of $\bar{b}_q$ from $[-q/2, q/2) \cap \mathbb{Z}$, then each coefficient of $\bar{b}_q / q$ is in the interval $[-1/2, 1/2)$, so

$$\lfloor b \rceil_2 = \bar{b}_q + \bar{b}_2 \bmod 2. \tag{3}$$

Equivalently, the coefficient vector of $\lfloor b \rceil_2$ is the exclusive-or of the coefficient vector of $\bar{b}_2$ and the least-significant bits of the coefficients of $\bar{b}_q$.

In SPRING-CRT, we need an unbiased rounding function $S$ from the *unit group* $R_{2q}^* \cong R_2^* \times R_q^*$ to $R_2$. An element of $R_2$, viewed as a polynomial, is a unit if and only if the sum of its coefficients is odd. So for a uniformly random element of $R_2^*$, any fixed choice of $n-1$ coefficients (e.g., all but the constant term) are uniformly random and independent, and the remaining one is determined. Because of Equation (3) above, any fixed choice of $n-1$ coefficients of $\bar{b}_2$ are uniformly random and independent, over the random choice of $b_2 \in R_2^*$ alone. Therefore, we define our generalized rounding function on $b \in R_{2q}^*$ to output a fixed $n-1$ bits of $\lfloor b \rceil_2 \in R_2$, which is perfectly unbiased.

Note that the above argument depends only on the random choice of the $R_2^*$ component, and doesn't use any of the randomness in the $R_q^*$ component. Using such an argument, $n-1$ independent and unbiased bits is the most we can possibly obtain. Since the number of units in $R_{2q}^*$ is exactly $(q-1)^n \cdot 2^{n-1}$, which is divisible by $2^n$, it seems plausible that there could exist a rounding function that outputs $n$ (nearly) unbiased bits given a random unit in $R_{2q}^*$, but so far we have not been able to find such a function. The main difficulty seems to be that the coefficients of the representative $\bar{b}_q$ are noticeably biased modulo 2.

## 3.2 Fast Arithmetic in $R_2^*$

We now give an algebraic decomposition of the group $R_2^*$, and present fast algorithms for performing subset-products and associated arithmetic operations.

The following theorem says that the unit group $R_2^*$ decomposes into the product of several small cyclic components, having power-of-2 orders at most $n$. Due to space constraints, a proof is deferred to the full version.

**Theorem 2.** *Define* $g_{0,0} = 1+(1+x)$ *and* $g_{i,k} = 1+(1+x)^{2^i+k}$ *for* $1 \le i < \lg(n)$ *and* odd $k \in \{1, \dots, 2^i\}$. *Then*

$$R_2^* \cong C_2^{n/4} \times C_4^{n/8} \times \ldots \times C_{n/2}^1 \times C_n^1 = \prod_{i=1}^{\lg(n)-1} C_{2^i}^{2^{j-i-1}} \times C_n, \tag{4}$$

*with each* $g_{i,k}$ *being a generator of one of the* $C_{n/2^i}$ *cyclic components.*

There are several ways of representing elements in $R_2^*$, which each allow for certain arithmetic operations to be performed more or less efficiently. We use the following three representations, the first of which is very good for fast multiplication, and the last of which is used for rounding. (As we shall see, the middle one is a convenient intermediate representation.)

1. Using the cyclic decomposition given in Theorem 2, we can represent an element by its tuple of integer exponents with respect to the generators $g_{i,k}$. We call this the *exponent representation.*
2. We can represent elements in $R_2$ by their vectors of $\mathbb{Z}_2$-coefficients with respect to what we call the *radix basis* $\{(1+x)^i\}_{0 \le i < n}$. (An element is in $R_2^*$ if and only if its coefficient for the basis element $(1+x)^0 = 1$ is 1.) The name of this basis arises from the fact that $(1+x)^n = 1 + x^n = 0 \pmod 2$, and therefore the coefficients can be thought of as digits in the "radix" $1+x$.

3. Finally, elements in $R_2$ can be represented by their vectors of $\mathbb{Z}_2$-coefficients with respect to the *power basis* $\{x^i\}_{0 \leq i < n}$, i.e., in the usual way as polynomials in $x$.

We now give algorithms for efficiently converting from the exponent representation to the power representation, using the radix representation as an intermediary.

*From exponents to radix basis.* We first make a few useful observations about the radix basis, and how powers of the generators $g_{i,k}$ look in this basis.

1. In the radix basis, multiplication by an element of the form $(1+x)^j$ corresponds to shifting the input's coefficient vector $j$ places (and discarding the "top" $j$ coefficients), since $(1+x)^n = 0$ in $R_2$. Therefore, multiplication by $1+(1+x)^j$ corresponds to taking the exclusive-or of the input's coefficient vector with that vector shifted by $j$ positions.
2. For any $j$ and $\ell$, we have that $(1+(1+x)^j)^{2^\ell} = 1+(1+x)^{j \cdot 2^\ell} \in R_2^*$, since the intermediate binomial coefficients $\binom{2^\ell}{i}$ for $0 < i < 2^\ell$ are all even.
3. Raising any generator $g_{i,k}$ to half its order yields $g_{i,k}^{n/2^{i+1}} = 1+(1+x)^j$, where $j = n/2 + (n/2^{i+1})k$. Moreover, the product of any two elements of this type, for $n/2 \leq j_1, j_2 < n$, is

$$(1+(1+x)^{j_1})(1+(1+x)^{j_2}) = 1+(1+x)^{j_1}+(1+x)^{j_2}.$$

   Thus, a subset-product of elements of this type can be computed as $\prod_{j \in \mathcal{I}}(1+(1+x)^j) = 1+\sum_{j \in \mathcal{I}}(1+x)^j$, for any $\mathcal{I} \subseteq \{n/2, \ldots, n-1\}$.

Now let the exponent representation of some $b \in R_2^*$ be $\{e_{i,k}\}$, where each $e_{i,k}$ denotes the exponent of the generator $g_{i,k}$. Write $e_{i,k} = \sum_{\ell=0}^{\lg(n)-i-1} e_{i,k,\ell} \cdot 2^\ell$, i.e., each $e_{i,k,\ell}$ is the $\ell$th bit of $e_{i,k}$, and observe that

$$g_{i,k}^{e_{i,k}} = \prod_{\ell=0}^{\lg(n)-i-1} \left(g_{i,k}^{2^\ell}\right)^{e_{i,k,\ell}}, \tag{5}$$

where we know by Item 2 above that $g_{i,k}^{2^\ell} = 1+(1+x)^{(2^i+k)\cdot 2^\ell}$.

We can now describe the algorithm that converts from exponent to radix representation. We effectively decompose the given powers $e_{i,k}$ of $g_{i,k}$ according to Equation (5), which we can then compute by Items 1 and 2 above. We note that Item 3 lets us handle all the most significant bits of all the exponents very quickly in one shot. (This yields a practical but not asymptotic improvement over handling these bits more naively.) The precise details are given in Algorithm 3.1 below.

If the length of the coefficient vector is considered to be the word-size, then apart from the most significant bits of the exponents (which are handled in one word operation in total), the other bits are handled in one shift-and-XOR

**Algorithm 3.1** Algorithm to convert from exponent to radix representation

```
 1: Input: Exponents $e_{i,k} \in [0, n/2^i)$ for positive odd $k < 2^i$ when $0 < i < \lg(n)$, and
    $k = 0$ when $i = 0$.                          ▷ Let $e_{i,k,\ell}$ denote the $\ell$th bit of $e_{i,k}$.
 2: Output: A bit vector $\mathbf{b} \in \mathbb{Z}_2^n$ representing the coefficients of $b$ in the radix basis.
 3: $\mathbf{b} \leftarrow (1, 0, \ldots, 0)$          ▷ Initialize the vector $\mathbf{b}$ to represent $1 \in R_2^*$
 4: for every valid $(i, k)$ pair do
 5:     if $e_{i,k,\lg(n)-i-1} = 1$ then
 6:         $b[n/2 + k \cdot n/2^{i+1}] \leftarrow 1$
 7: for every valid $(i, k)$ pair do
 8:     for $\ell = (\lg(n) - 2) - i$ down to 0 do
 9:         if $e_{i,k,\ell} = 1$ then
10:             $\mathbf{b} \leftarrow \mathbf{b} \oplus (\mathbf{b} \gg ((2^i + k) \cdot 2^\ell))$     ▷ The shift-and-XOR operation.
```

operation each, which is a constant number of word operations each. Since the exponents take $n - 1$ bits in total, the algorithm performs a total of $O(n)$ word operations. (Since each word is $n$ bits long, the bit complexity of Algorithm 3.1 is $O(n^2)$.)

*From radix basis to power basis.* For the second step, we have a bit vector $\mathbf{b} \in \mathbb{Z}_2^n$ representing some $b \in R_2$ with respect to the radix basis, and wish to convert to the power basis. We express $b$ as follows:

$$\begin{aligned} b = \sum_{i=0}^{n-1} b_i (1+x)^i &= \sum_{i=0}^{n/2-1} b_i (1+x)^i + (1+x)^{n/2} \sum_{i=0}^{n/2-1} b_{i+n/2} (1+x)^i \\ &= \sum_{i=0}^{n/2-1} b_i (1+x)^i + (1+x^{n/2}) \sum_{i=0}^{n/2-1} b_{i+n/2} (1+x)^i \qquad (6) \\ &= \left( \sum_{i=0}^{n/2-1} (b_i + b_{i+n/2})(1+x)^i \right) + x^{n/2} \sum_{i=0}^{n/2-1} b_{i+n/2} (1+x)^i \pmod 2, \qquad (7) \end{aligned}$$

where (6) follows from the fact that $(1+x)^{2^j} = 1 + x^{2^j} \pmod 2$ (since $\binom{2^j}{i}$ is even for every $0 < i < 2^j$), and $n$ is a power of 2. Converting the $n$-bit vector $\mathbf{b}$ therefore reduces to two conversions of $n/2$-bit vectors, namely, the top half of $\mathbf{b}$ and the exclusive-or of the top and bottom halves of $\mathbf{b}$. This directly yields a simple divide-and-conquer algorithm to transform the coefficient vector $\mathbf{b}$ in place, which is detailed in Algorithm 3.2 below. The number of bit operations follows the simple recursive equation $T(n) = 2T(n/2) + n/2$, which solves to $T(n) = O(n \log n)$.

## 4 Security Analysis

In this section we analyze the security of our construction against known classes of attacks, and introduce new attacks specific to the structure of $R_{2q}$.

**Algorithm 3.2** Algorithm to transform from radix basis to power basis of $R_2$

```
1: procedure Radix-to-Power($\mathbf{b}, f, \ell$)
Input: array $\mathbf{b}$, index $f$ and length $\ell$ ▷ The initial call is made with $f = 0$ and $\ell = n$
Output: subvector $\mathbf{b}[f, f + \ell - 1]$ converted to power basis
2:     if $\ell > 1$ then
3:         for $i = 0$ to $\ell/2 - 1$ do
4:             $b[f + i] \leftarrow b[f + i] \oplus b[f + \ell/2 + i]$
5:         Radix-to-Power($\mathbf{b}, f, \ell/2$)
6:         Radix-to-Power($\mathbf{b}, f + \ell/2, \ell/2$)
```

## 4.1 Overview of Known Attacks

The concrete security of the SPRING PRF for practical parameters is not well understood, but to date there are no known attacks that nontrivially exploit the internal subset-product structure. As discussed earlier, the SPRING construction follows the paradigm from [BPR12], which results in a PRF that is secure against all efficient adversaries, assuming the hardness of the (ring-)LWE problem (appropriately parameterized). Informally, the ring-LWE problem asks the adversary to distinguish many pairs $(a_i, b_i) \in R_p \times R_p$, where each $a_i$ is chosen uniformly and $b_i \approx a_i \cdot s$ is its *noisy* product with a secret ring element $s$, from uniformly random pairs. The BPR security reductions make two assumptions that do not hold in our instantiations: (1) the parameter $p$ is exponential in the input length $k$ of the PRF, and (2) the seed elements $s_i$ are all "small" ring elements in $R$; more precisely, they are drawn from the error distribution from the underlying ring-LWE assumption. However, as we shall see in what follows, relaxing these requirements do not appear to introduce any concrete attacks against the function family.

For the sake of modeling certain attacks against SPRING, we can think of it as a LWE-type learning problem. The difference here is that *all* ring elements output by SPRING have rounding errors in them, whereas ring-LWE releases the multiplicand $a$ without any error. In this respect, attacking SPRING seems potentially harder than attacking ring-LWE.

The main classes of attacks against noisy learning problems akin to LWE are: (1) brute-force attacks on the secret, (2) combinatorial-type attacks following [BKW03,Wag02,MR09], (3) lattice reduction attacks, and (4) algebraic attacks following [AG11]. We consider each of these in turn. We note that the lattice and algebraic attack strategies described below apply to (ring-)LWE with our parameters. It is not clear whether these attacks will adapt to SPRING, where multiplicands are not known exactly, but to be conservative we assume that they might. While most of these attacks take a prohibitively large amount of time and/or space (more than $2^{200}$), one kind of birthday-type attack technique performs reasonably well against SPRING-CRT. Even in this case, its running time is nearly $2^{128}$ bit operations and its space requirements are about $2^{119}$, when $n = 128$.

*Brute-force and combinatorial attacks.* A brute-force attack involves searching for a secret $s_i \in R_p^*$, or for the round-off terms in enough samples to uniquely determine an $s_i$. The secret and round-off terms come from sets of size at least $(p/2)^n$, which is prohibitively large for all our parameters. Combinatorial (or "generalized birthday") attacks on noisy learning problems [BKW03,Wag02] work by drawing an huge number of samples and finding (via birthday collisions) small combinations that sum to lie in a small enough subgroup, then testing whether the noise can be detected. This works for small error rates because the small combinations still retain small error terms. In the case of SPRING-CRT, this style of attack looks the most promising, and a concrete attack in this vein is developed further in Section 4.2.

*Lattice attacks.* Lattice attacks on (ring-)LWE typically work by casting it as a bounded-distance decoding (BDD) problem on a certain class of random lattices (see for instance [MR09,LP11,LN13,vdPS13]). At a high level, the attack draws a sufficiently large number $L$ of samples $(a_i, b_i) \in R_p \times R_p$, so that the secret (in the LWE case) is uniquely determined with good probability. With error rate $1/2$, we need $L \geq \lg(p/2)$ by a simple information-theoretic argument. The attack collects the samples into vectors $\boldsymbol{a}, \boldsymbol{b} \in R_p^L$, and considers the "$p$-ary" lattice $\mathcal{L}$ of dimension $N = nL$ (over $\mathbb{Z}$) corresponding to the set of vectors $s \cdot \boldsymbol{a} \in R_p^L$ for all $s \in R_p$. It then attempts to determine whether $\boldsymbol{b}$ is sufficiently close to $\mathcal{L}$, which corresponds to whether $(a_i, b_i)$ are LWE samples or uniform. In our setting, because the error rate $1/2$ is so large, the distance from $\boldsymbol{b}$ to $\mathcal{L}$ (in the LWE case) is nearly the minimum distance of the lattice, up to a constant factor no larger than four (this is a conservative bound). Therefore, for the attack to succeed it needs to solve BDD (or the shortest vector problem SVP) on $\mathcal{L}$ to within an very small constant approximation factor. For the parameters in our instantiations, the lattice dimension is at least $N \geq n \lg(p/2) \geq 896$ (and likely more). For this setting, the state of the art in BDD and SVP algorithms [CN11,LN13,MV10], take time at least $2^{0.48N} \geq 2^{430}$, and likely more. Moreover, the SVP algorithm of [MV10], which appears to provide the best heuristic runtime in this setting, as a most conservative estimate requires space at least $2^{0.18N} \geq 2^{160}$.

*Algebraic attacks.* Finally, the algebraic "linearization" attack of Arora and Ge [AG11] yields a lower bound on $p$ for security. The attack is applicable when every coefficient of every error term is guaranteed to belong to a known set of size $d$; in our setting, $d = p/2$. The attack requires at least $N/n$ ring-LWE samples to set up and solve a dense linear system of dimension $N$, where

$$N = \binom{n+d}{n} \approx 2^{(n+d) \cdot H(n/(n+d))}$$

and $H(\delta) = -\delta \lg(\delta) - (1-\delta) \lg(1-\delta)$ is the binary entropy function for $\delta \in (0,1)$. Therefore, the attack requires time and space at least $N^2$, which is at least $2^{384}$ for even the most aggressive of all our parameters.

## 4.2 Birthday-type Attack on SPRING-CRT

We now describe a specific birthday-type attack on SPRING-CRT, which exploits the structure of the ring $R_{2q}$. The main idea is to cancel out the $R_2$ component and to detect the bias in the remaining $R_q$ component.

To do this, we first split the input $x$ into two parts, as $x = y\|z$ for $y, z$ of certain lengths. Then the SPRING-CRT function (for simplicity, without dropping a bit after rounding) can be written as

$$F(y\|z) = \lfloor a \cdot S_y \cdot S_z \rceil_2,$$

where $S_y$ and $S_z$ respectively represent the subset product of the keys $s_i$ indicated by the bits of $y$ and $z$.

The basic goal in the attack is to try to find values $y$ and $y'$ such that $S_y = S_{y'} \bmod 2$. If we have two such values, then $a \cdot S_y \cdot S_z = a \cdot S_{y'} \cdot S_z \bmod 2$ for any $z$, so the $R_2$ component of the output will be the same for the inputs $y\|z$ and $y'\|z$. By Equation (3) in Section 3.1, this implies that in $F(y\|z) \oplus F(y'\|z)$, the respective $R_2$ components cancel each other out. Since rounding of a uniform element in $R_q$ has a bias of $1/q$ in each coefficient, the bits of $F(y\|z) \oplus F(y'\|z)$ will be the sum of two biased bits, i.e., the bias is $1/q^2$. This can be detected using $q^4/4$ pairs of output bits (with varying $z$).

If we repeat the test for $2^n$ different choices of $(y, y')$, with high probability, one choice satisfies $S_y + S_{y'} = 0 \bmod 2$, and we would be able to detect the bias by the method detailed above. (By contrast, the test would not detect such bias in a truly random function, with high probability.) We can collect the data for the attack with $2^{n/2}$ distinct choices of $y$ and $y'$, each of them using $q^4/(4n)$ values of $z$. This requires $2^{n/2} \cdot q^4/(4n)$ queries *and* space, and a time complexity of $2^n \cdot q^4/2$.

*Generalizing the attack.* We can generalize this analysis using $y$ and $y'$ such that $(S_y + S_{y'})^2 = 0 \bmod 2$. This implies that the $S_{y,2} + S_{y',2}$ is a multiple of $x^{n/2} + 1$, where $S_{y,2} = S_y \bmod 2$ and similarly for $S_{y',2}$. Thus, $c_i$ and $c_{i+n/2}$, the coefficients of $x^i$ and $x^{n+i/2}$ respectively in $S_{y,2} + S_{y',2}$, are the same. This implies that if we XOR the lower and upper halves of $F(y\|z) \oplus F(y'\|z)$, we can effectively remove the $R_2$ component as above, and then can recognize the bias in the $R_q$ component. Since we sum four bits to remove the $R_2$ component, we reduce the bias, but a random pair $y, y'$ satisfies the condition with probability $2^{-n/2}$ instead of $2^{-n}$. This gives an attack with query and space complexity $2^{n/4} \cdot q^8/(4n)$ and time complexity $2^{n/2} \cdot q^8/4$. This can be generalized further: if we use $y$ and $y'$ such that $(S_y + S_{y'})^t = 0 \bmod 2$ (for $t$ a power of 2), we reach a time complexity of $2^{n/t} \cdot q^{4t}/(2t)$.

With $q = 257$ and $n = 128$, our best attack on SPRING-CRT (using $t = 2$) has time complexity roughly $2^{64+64-2} = 2^{126}$, and query and space complexity roughly $2^{n/2} \cdot q^8/(4n) \approx 2^{119}$.

## 5 Implementation Details

The SPRING design is targeted for efficient implementation using SIMD instructions, and a well-optimized implemention allows us to reach throughputs that are not too far from those of classical symmetric-key constructions.

SIMD instructions perform a given operation on multiple data in parallel. Processors with a SIMD engine usually come with a set of dedicated registers, which can contain a vector of integers or floating point data, and the SIMD instruction set computes arithmetic operations in parallel on the vectors elements, e.g., addition, multiplication, bitwise operations, rotations, etc. as well as some permutations of the vector elements. SIMD instructions were introduced in personal computers to improve the efficiency of multimedia computations, and are now very widely available. SIMD engines with 128-bit wide vectors are available on all desktop processors (SSE2 on x86/x86_64, Altivec on PowerPC, NEON on ARM), and even on embedded platforms such as smart-phones and tablets, with ARM Cortex-A or Intel Atom. Very recently, Intel has introduced AVX2, with integer operations on 256-bit SIMD vectors.

We implemented the two instantiations of SPRING defined in Sections 2 and 3. SPRING-BCH involves a subset-product in $R_q^*$, followed by rounding and bias reduction (using the BCH code), while SPRING-CRT involves a subset product in $R_{2q}^*$ followed by rounding. As described in Section 3.1, this can be implemented as separate subset-products in $R_2^*$ and $R_q^*$, followed by an extraction of the least significant bits in the $R_q^*$ component and an exclusive-or with the $R_2^*$ component. For each version, we have an implementation of the PRF with standalone subset-products, and an amortized implementation in Gray code counter mode where we just perform one ring multiplication before each rounding operation. In the following subsections we explain how to efficiently implement the main operations.

### 5.1 Computations in $R_2^*$

*Subset-sum and conversion from exponent to power basis.* We use the cyclic decomposition of $R_2^*$ given in Theorem 2, and store the key in exponent representation, so that the subset-product is a subset-sum of the exponents. A polynomial in $R_2$ (of degree less than 128) is represented by 32 one-bit indices, 16 two-bit indices, 8 three-bit indices, 4 four-bit indices, 2 five-bit indices, 1 six-bit index, and 1 seven-bit index. We store all 64 indices as 8-bit integers, and use SIMD instructions to compute the sum. Since all the cyclic groups have an order that is a power of 2, we can use 8-bit additions, and remove the extra bits at the end. The conversion to the power basis is done using Algorithms 3.1 and 3.2. Algorithm 3.2 is rewritten iteratively using shift, mask and xor instructions, taking advantage of the inherent parallelism of bitwise operations, as shown in Algorithm 5.1.

**Algorithm 5.1** Iterative version of Algorithm 3.2 using the parallelism of bitwise operations

```
1: procedure Radix-to-Power($\mathbf{b}$)
Input: 128-bit vector $\mathbf{b}$
2:     $\mathbf{b} \leftarrow \mathbf{b} \oplus (\mathbf{b} \wedge \texttt{0xffffffffffffffff0000000000000000}) \gg 64$
3:     $\mathbf{b} \leftarrow \mathbf{b} \oplus (\mathbf{b} \wedge \texttt{0xffffffff00000000ffffffff00000000}) \gg 32$
4:     $\mathbf{b} \leftarrow \mathbf{b} \oplus (\mathbf{b} \wedge \texttt{0xffff0000ffff0000ffff0000ffff0000}) \gg 16$
5:     $\mathbf{b} \leftarrow \mathbf{b} \oplus (\mathbf{b} \wedge \texttt{0xff00ff00ff00ff00ff00ff00ff00ff00}) \gg 8$
6:     $\mathbf{b} \leftarrow \mathbf{b} \oplus (\mathbf{b} \wedge \texttt{0xf0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0}) \gg 4$
7:     $\mathbf{b} \leftarrow \mathbf{b} \oplus (\mathbf{b} \wedge \texttt{0xcccccccccccccccccccccccccccccccc}) \gg 2$
8:     $\mathbf{b} \leftarrow \mathbf{b} \oplus (\mathbf{b} \wedge \texttt{0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa}) \gg 1$
```

*Polynomial multiplication.* In counter mode, we found it more efficient to compute a single ring multiplication directly than to use the exponent representation.

On recent Intel CPUs (starting from the Westemere architecture introduced in 2010) and AMD CPUs (starting from the Bulldozer architecture introduced in 2011), there is a carry-less multiplication operation, `pclmulqdq`, that computes a 64-bit polynomial multiplication modulo two. This gives a very efficient implementation of the $R_2$ multiplication.

Alternatively, we take take advantage of the fact that one of the operands is always a polynomial from the key (or its inverse). Therefore, we can see it as a multiplication by a fixed element in $R_2$, which is a linear operation. We can precompute tables corresponding to this linear operation with 8-bit subsets of the input range, and compute the full multiplication using $n/8$ table accesses and xors.

More precisely, we precompute $z \cdot s$ for all polynomials $z$ of degree less than 8, and we write a degree-128 polynomial $z$ as $z_0 + x^8 \cdot z_1 + \cdots + x^{120} \cdot z_{15}$, where all the $z_i$ are of degree at most 7. Then we can compute $z \cdot s$ as $(z_0 \cdot s) + x^8 \cdot (z_1 \cdot s) + \cdots + x^{120} \cdot (z_{15} \cdot s)$, which requires only 16 table accesses, rotations, and xors. This trick takes about 1MB of extra memory to store the tables, but this is negligible on the platforms we target.

### 5.2 Computations in $R^*_{257}$

Following [LMPR08], we use the Chinese Remainder Theorem isomorphism of the ring $R_q \cong \mathbb{Z}_q^n$ when $q = 1 \pmod{2n}$ is prime. A product in $R_q$ therefore corresponds to a component-wise multiplication of vectors in $\mathbb{Z}_q^n$. Moreover, there are fast FFT-like algorithms, often called "number theoretic transforms" (NTT), for converting between the polynomial representation of $R_q$ and the $n$-fold product ring $\mathbb{Z}_q^n$.

*Subset-sum and conversion.* Since the ring elements we multiply are all part of the key, we can generate and store them as vectors in the product ring $\mathbb{Z}_q^n$. Moreover, since these elements are all unit, their entries in $\mathbb{Z}_q^n$ are non-zero, and we can actually store the discrete logarithms of the entries (with respect to some generator of $\mathbb{Z}_q^*$), so that the subset-product becomes a subset-sum.

Exponentiation by the final summed exponents can be implemented with a simple table lookup. However, we found a slightly more efficient version using vector permutation (`pshufb` in SSSE3) instructions as a 4-bit to 8-bit parallel table lookup. We use the fact that $g^{a+16\times b} = g^a \cdot (g^{16})^b$, where $a$ and $b$ are both 4-bit values, and we use 4-bit to 8-bit tables for $g^x$ and $(g^{16})^x$.

*Product and conversion.* In Gray code counter mode, we do not use the exponent representation, because a point-wise multiplication is more efficient than a point-wise addition followed by exponentiations. This is because the point-wise multiplication can be parallelized easily while the exponentiation requires either serial table lookups, or a more complex sequence of SIMD operations.

**NTT** The bottleneck of our function is the NTT computation, therefore we have to optimize this part aggressively. In our implementation, we reuse the code from the SIMD hash function [LBF08] which happens to use the same parameters as the transformation needed in SPRING. The main tricks used in this implementation are:

*Representation of elements.* Element in $\mathbb{Z}_{257}$ are stored as signed 16-bit words. The choice of the modulus 257 allows an efficient implementation of the field operations, because 257 is a prime and $256 = -1 \pmod{257}$. Moreover, $\mathbb{Z}_{257}^*$ is a cyclic group of 256 elements, where the subset sum of the logarithms can be computed with a simple 8-bit addition.

*Reduction.* We use `(x&255) - (x>>8)` to do a partial reduction modulo 257, with the output in $[-127, 383]$. When a full reduction to a smaller range is needed, we subtract 257 to values greater than 128 to reduce the range to $[-128, 128]$. This can be performed completely with SIMD instructions and does not require any division. We note that it is not necessary to perform a reduction after each field operation, because we have some extra bits in a 16-bit word; we have to study the NTT algorithm to find out where reductions are needed.

*Multiplication.* To compute a multiplication in $\mathbb{Z}_{257}$, we reduce both operands to $[-128, 128]$, and the result can be computed with a single 16-bit multiplication without any overflow.

*Using a two-dimensional NTT.* Because SIMD instructions compute the same operation on each element of the vectors, we do not use the classical radix-2 NTT algorithm. Instead, we rewrite the one-dimensional NTT as a two-dimensional one. In our implementation, we rewrite an NTT of size 64 as a two-dimensional NTT of size $8 \times 8$. The input data is seen as a $8 \times 8$ matrix, and the computation of the $\mathsf{NTT}_{64}$ is done in three steps:

– First we compute 8 parallel $\mathsf{NTT}_8$ on the columns of the matrix using a decimation in time algorithm.

– We multiply by the twiddle factors, transpose the matrix, and permute the row and the columns following the bit reversal order.
– Then we compute 8 parallel $\mathsf{NTT}_8$ on the columns of the matrix using a decimation in frequency algorithm.

The first and the last step are easy to parallelize with SIMD instructions because they compute the same transformation on 8 independent inputs. Moreover, the root of unity used in the $\mathsf{NTT}_8$ is 4, so the multiplications needed for the $\mathsf{NTT}_8$ are simply bit shifts. The transposition can be implemented using merge operations available on most SIMD instruction sets (e.g., `punpcklwd`/`punpckhwd` in SSE).

For the 128-dimensional NTT, we reused the code of the $\mathsf{NTT}_{64}$, and we have to perform an extra layer of butterfly operations and multiplications by twiddle factors (we decompose the $\mathsf{NTT}_{128}$ as an $\mathsf{NTT}_{64}$ and a $\mathsf{NTT}_2$).

### 5.3 Reducing Bias with a BCH Code

After rounding the $R_{257}$ computation output, we are left with a $n$-dimensional vector over $\mathbb{Z}_2$, each element with a bias of 1/257. We apply the generator polynomial of a BCH code in order to reduce the output's bias. Specifically, for the case of $n = 128$ we apply the extension using a parity bit on the BCH code with parameters $[127, 64, 21]$ in order to gain a generator for a code with distance 22. Therefore, the whole computation is done using just a few shift and xor instructions, and one final negate instruction. We use the BCH generator polynomial

$$1 + x^2 + x^7 + x^8 + x^{10} + x^{12} + x^{14} + x^{15} + x^{16} + x^{23} + x^{25} + x^{27} + x^{28} + x^{30} + x^{31} \\ + x^{32} + x^{33} + x^{37} + x^{38} + x^{39} + x^{40} + x^{41} + x^{42} + x^{44} + x^{45} + x^{48} + x^{58} + x^{61} + x^{63}$$

since it matches our desired code parameters and has a minimal number of nonzero coefficients. The output is 64 bits which makes consecutive outputs easy to maintain in a packed array of 64-bit words. We note that if the PCLMUL instruction is available, we can apply the generator polynomial on 127 rounded output bits immediately by xoring outputs of such two instructions.

## References

[ACPR13] G. Alberini, E. Crockett, C. Peikert, and A. Rosen. Fast homomorphic evaluation of symmetric-key primitives, 2013. Manuscript.

[AG11] S. Arora and R. Ge. New algorithms for learning in presence of errors. In *ICALP (1)*, pages 403–415. 2011.

[AR13] G. Alberini and A. Rosen. Efficient rounding procedures of biased samples, 2013. Manuscript.

[BBS86] L. Blum, M. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM J. Comput.*, 15(2):364–383, 1986.

[BKW03] A. Blum, A. Kalai, and H. Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, 2003.

[BLMR13] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan. Key homomorphic PRFs and their applications. In *CRYPTO*, pages 410–428. 2013.

[BM82] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13(4):850–864, 1984. Preliminary version in FOCS 1982.

[BPR12] A. Banerjee, C. Peikert, and A. Rosen. Pseudorandom functions and lattices. In *EUROCRYPT*, pages 719–737. 2012.

[CCK$^+$13] J. H. Cheon, J.-S. Coron, J. Kim, M. S. Lee, T. Lepoint, M. Tibouchi, and A. Yun. Batch fully homomorphic encryption over the integers. In *EUROCRYPT*, pages 315–335. 2013.

[CN11] Y. Chen and P. Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT*, pages 1–20. 2011.

[DES77] National Bureau of Standards, U.S. Department of Commerce, Washington D.C. National Bureau of Standards Data Encryption Standard, FIPS-Pub.46. 1977.

[DR00] J. Daemen and V. Rijmen. Rijndael for AES. In *AES Candidate Conference*, pages 343–348. 2000.

[eBA] eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`, accessed 11 November 2013.

[GGM84] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986. Preliminary version in FOCS 1984.

[GHS12] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, pages 850–867. 2012.

[KS09] E. Käsper and P. Schwabe. Faster and Timing-Attack Resistant AES-GCM. In C. Clavier and K. Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009. ISBN 978-3-642-04137-2.

[LBF08] G. Leurent, C. Bouillaguet, and P.-A. Fouque. SIMD Is a Message Digest. Submission to NIST, 2008. `http://www.di.ens.fr/~leurent/files/SIMD.pdf`.

[LM06] V. Lyubashevsky and D. Micciancio. Generalized compact knapsacks are collision resistant. In *ICALP (2)*, pages 144–155. 2006.

[LMPR08] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFT: A modest proposal for FFT hashing. In *FSE*, pages 54–72. 2008.

[LN13] M. Liu and P. Q. Nguyen. Solving BDD by enumeration: An update. In *CT-RSA*, pages 293–309. 2013.

[LP11] R. Lindner and C. Peikert. Better key sizes (and attacks) for LWE-based encryption. In *CT-RSA*, pages 319–339. 2011.

[LPR10] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 2013. To appear. Preliminary version in Eurocrypt 2010.

[Mic02] D. Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. *Computational Complexity*, 16(4):365–411, 2007. Preliminary version in FOCS 2002.

[MR09] D. Micciancio and O. Regev. Lattice-based cryptography. In *Post Quantum Cryptography*, pages 147–191. Springer, February 2009.

[MV10] D. Micciancio and P. Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *SODA*, pages 1468–1480. 2010.

[NN90] J. Naor and M. Naor. Small-bias probability spaces: Efficient constructions and applications. *SIAM J. Comput.*, 22(4):838–856, 1993. Preliminary version in STOC 1990.

[NR95] M. Naor and O. 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.

[NR97] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. *J. ACM*, 51(2):231–262, 2004. Preliminary version in FOCS 1997.

[NRR00] M. Naor, O. Reingold, and A. Rosen. Pseudorandom functions and factoring. *SIAM J. Comput.*, 31(5):1383–1404, 2002. Preliminary version in STOC 2000.

[Pei09] C. Peikert. Public-key cryptosystems from the worst-case shortest vector problem. In *STOC*, pages 333–342. 2009.

[PR06] C. Peikert and A. Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. In *TCC*, pages 145–166. 2006.

[Reg05] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):1–40, 2009. Preliminary version in STOC 2005.

[vdPS13] J. van de Pol and N. P. Smart. Estimating key sizes for high dimensional lattice based systems. Cryptology ePrint Archive, Report 2013/630, 2013. `http://eprint.iacr.org/`.

[Wag02] D. Wagner. A generalized birthday problem. In *CRYPTO*, pages 288–303. 2002.