SWIFFT: A Modest Proposal for FFT Hashing [*][†]

Vadim Lyubashevsky, UCSD
Daniele Micciancio, UCSD
Chris Peikert, SRI International
Alon Rosen, IDC Herzliya

Abstract

We propose SWIFFT, a collection of compression functions that are highly parallelizable and admit very efficient implementations on modern microprocessors. The main technique underlying our functions is a novel use of the *Fast Fourier Transform* (FFT) to achieve "diffusion," together with a linear combination to achieve compression and "confusion." We provide a detailed security analysis of concrete instantiations, and give a high-performance software implementation that exploits the inherent parallelism of the FFT algorithm. The throughput of our implementation is competitive with that of SHA-256, with additional parallelism yet to be exploited.

Our functions are set apart from prior proposals (having comparable efficiency) by a supporting asymptotic *security proof*: it can be formally proved that finding a collision in a randomly-chosen function from the family (with noticeable probability) is at least as hard as finding short vectors in cyclic/ideal lattices in the *worst case*.

1 Introduction

In cryptography, there has traditionally been a tension between efficiency and rigorous security guarantees. The vast majority of proposed cryptographic hash functions have been designed to be highly efficient, but their resilience to attacks is only based on intuitive arguments and validated by intensive cryptanalytic efforts. Recently, new cryptanalytic techniques [24, 25, 4] have started casting serious doubts both on the security of these specific functions and on the effectiveness of the underlying design paradigm.

On the other side of the spectrum, there are hash functions having rigorous asymptotic proofs of security (i.e., security reductions), assuming that various computational problems (like the discrete logarithm problem or factoring large integers) are hard to solve on the average. Unfortunately, all such proposed hash functions have had computation cost comparable to typical public key cryptographic operations, making them unattractive from a practical point of view.

1.1 Our Proposal: SWIFFT

We propose the *SWIFFT* collection of compression functions, and give a high-performance software implementation. SWIFFT is very appealing and intuitive from a traditional design perspective,

[*]**mod·est**, adj.: 1. Marked by simplicity. 2. Is modular (Latin).

[†]To appear in Fast Software Encryption, Proceedings of FSE 2008, LNCS, Springer. This material is based upon work supported by the National Science Foundation under Grants CCF-0634909 (Lyubashevsky and Micciancio), CNS-0716786 and CNS-0749931 (Peikert). Any opinions, findings, and conclusions or recommedations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

and, at the same time, achieves the robustness and reliability benefits of *provable* asymptotic security under a mild assumption. The functions correspond to a simple algebraic expression over a certain polynomial ring, as described in Section 2.1. Here we describe a high-level algorithm for the fast evaluation of a SWIFFT compression function.

The algorithm takes as input a binary string of length mn (for suitable parameters m, n), which is viewed as an $n \times m$ binary matrix $(x_{i,j}) \in \{0,1\}^{n \times m}$. It then performs the following two steps, where all operations are performed in $\mathbb{Z}_p$ for an appropriate modulus p:

1. The input matrix $(x_{i,j})$ is first processed by multiplying the ith row by ω^{i-1} for $i = 1, \dots, n$ (where $\omega \in \mathbb{Z}_p$ is an appropriate fixed element).

   Then the *Fast Fourier Transform* (FFT) is computed (over $\mathbb{Z}_p$) on each column $j = 1, \dots, m$:

   $$(y_{1,j} \ , \ \dots \ , \ y_{n,j}) = \mathrm{FFT}(\omega^0 \cdot x_{1,j} \ , \ \dots \ , \ \omega^{n-1} \cdot x_{n,j}).$$

   We remark that this operation is easy to invert, and is performed to achieve "diffusion," i.e., to mix the input bits of every column.

2. A linear combination is then computed across each row $i = 1, \dots, n$:

   $$z_i = a_{i,1} \cdot y_{i,1} + \cdots + a_{i,m} \cdot y_{i,m} = \sum_{j=1}^{m} a_{i,j} \cdot y_{i,j},$$

   where the coefficients $a_{i,j} \in \mathbb{Z}_p$ are fixed as part of the function description.

   This operation compresses the input, achieving "confusion."

The output is the vector $(z_1, \dots, z_n) \in \mathbb{Z}_p^n$.

Consider an attempt to invert the function, i.e., to find some input $(x_{i,j})$ that evaluates to a given output $(z_1, \dots, z_n)$. Viewed independently, each linear equation $z_i = \sum_{j=1}^{m} a_{i,j} \cdot y_{i,j}$ on the rows admits a large number of easily-computed solutions. However, there are strong dependencies among the equations. In particular, every column $(y_{1,j}, \dots, y_{n,j})$ is constrained to be the result of applying Step 1 to an n-dimensional *binary* vector $(x_{1,j}, \dots, x_{n,j}) \in \{0,1\}^n$.

Perhaps surprisingly, these constraints turn out to be sufficient to guarantee asymptotically that the SWIFFT functions are *provably* one-way and collision-resistant. More precisely, the family admits a very strong security reduction: finding collisions on the average (when the coefficients $a_{i,j}$ are chosen at random in $\mathbb{Z}_p$) with any noticeable probability is at least as hard as solving an underlying mathematical problem on certain kinds of *point lattices* in the *worst case*. This claim follows from the fact that the SWIFFT functions are a special case of the *cyclic/ideal lattice*-based functions of [13, 17, 12].

SWIFFT's simple design has a number of other advantages. First, it also enables unconditional proofs of a variety of *statistical* properties that are desirable in many applications of hash functions, both in cryptography and in other domains. Second, its underlying mathematical structure is closely related to well-studied cryptographic problems, which permits easy understanding and analysis of concrete instantiations. Third, it is extremely parallelizable, and admits software implementations with throughput comparable to (or even exceeding) the SHA-2 family on modern microprocessors.

1.2 Related Work

Using the Fast Fourier Transform (FFT) as a building block in hash functions is not new. For example, Schnorr proposed a variety of FFT-based hash functions [19, 20, 21], which unfortunately were subsequently cryptanalyzed and shown to be insecure [7, 2, 22]. Our compression functions are set apart from previous work by the way that the FFT is used, and the resulting proof of security. Namely, while in previous work [19, 20, 21] the FFT was applied to *unrestricted* input vectors $(x_1, \ldots, x_n) \in \mathbb{Z}_p^n$, here we require the input values x_i to be bits. This introduces non-linear constraints on the output values of the FFT operation, a fact that plays a fundamental role both in our theoretical proof of security, as well as on the analysis of our concrete functions. Our novel use of FFT may be of independent interest, and might find other applications in cryptographic design.

Another important ingredient in the conceptual design of our functions (and associated proof of security) is the use of lattices with special structure as an underlying mathematical problem. Special classes of lattices (with closely related, but somewhat different structure than ours) also have been used before in practical constructions (most notably, the NTRU encryption scheme [10] and LASH hash function [3]), but without any security reductions.

More closely related to our work is the theoretical study initiated by Ajtai [1] of cryptographic functions that are provably secure under worst-case assumptions for lattice problems. Ajtai's work and subsequent improvements [8, 6, 14, 15] do not lead to very efficient implementations, mostly because of the huge size of the function description and slow evaluation time (which grow quadratically in the security parameter). A first step toward bridging the gap between theoretical constructions and practical functions was taken by Micciancio [13], who proposed using lattices with special structure (namely, cyclic lattices) and showed how they lead to cryptographic functions that have provable worst-case hardness and still admit fast implementations using FFT. The main limitation of the functions proposed in [13] was the notion of security achieved: they are provably one-way (under a worst-case assumption on cyclic lattices), but not collision resistant, as subsequently shown by Peikert and Rosen [17] and Lyubashevsky and Micciancio [12]. Those works then modified and generalized the function originally proposed in [13] to achieve collision resistance.

¿From a theoretical point of view, the SWIFFT compression functions proposed in this paper are equivalent to and inherit all provable security features from the cyclic/ideal hash functions of [17, 12]. But differently from [13, 17, 12], the emphasis in this paper is on practical implementation issues, and the construction of concrete instances and variants of those functions that enjoy very efficient implementation from a practical point of view. For a deeper understanding of the theoretical ideas underlying the proofs of security of our compression functions, we refer the reader to [13, 17, 12].

2 SWIFFT Compression Functions

In this section, we describe an algebraic expression that is the underlying foundation of the SWIFFT functions, and how it is related to the FFT-based algorithm described in Section 1.1. We then propose a set of concrete parameters on which our implementation and security analysis are based.

2.1 Algebraic Description

The SWIFFT functions correspond to a simple algebraic expression over certain a polynomial ring. A *family* of SWIFFT functions is described by three main parameters: let n be a power of 2, let

$m > 0$ be a small integer, and let $p > 0$ be a modulus (not *necessarily* prime, though we will soon see that certain prime p will be convenient). Define R to be the ring $R = \mathbb{Z}_p[\alpha]/(\alpha^n + 1)$, i.e., the ring of polynomials (in α) having integer coefficients, modulo p and $\alpha^n + 1$. Any element of R may therefore be written as a polynomial of degree $< n$ having coefficients in $\mathbb{Z}_p = \{0, \ldots, p-1\}$.

A particular *function* in the family is specified by m fixed elements $\mathbf{a}_1, \ldots, \mathbf{a}_m \in R$ of the ring R, called "multipliers." The function corresponds to the following expression over the ring R:

$$\sum\nolimits_{i=1}^{m} (\mathbf{a}_i \cdot \mathbf{x}_i) \quad \in \quad R, \tag{1}$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_m \in R$ are polynomials having *binary* coefficients, and corresponding to the binary input of length mn.

To compute the above expression, the main bottleneck is in computing the polynomial products $\mathbf{a}_i \cdot \mathbf{x}_i$ over R. It is well-known that the *Fast Fourier Transform* (FFT) provides an $O(n \log n)$-time algorithm that can be used for multiplying polynomials of degree $< n$. The multiplication algorithm starts by using the FFT to compute (all at once) the *Fourier coefficients* of each polynomial, i.e., the values on all the $2n$th roots of unity over the complex field $\mathbb{C}$. It then multiplies the respective Fourier coefficients of the two polynomials, and finally interpolates back to a degree $< 2n$ polynomial via an inverse FFT.

Because we are working modulo p and $\alpha^n + 1$, there is an even more convenient and efficient method for computing the polynomial products in the ring R. Suppose that p is prime and $p - 1$ is a multiple of $2n$. Then $\mathbb{Z}_p$ is a field, and it contains a multiplicative subgroup of order $2n$ whose elements are all the $2n$th roots of unity in $\mathbb{Z}_p$ (i.e., the roots of the polynomial $\alpha^{2n} - 1 \bmod p$). Let $\omega \in \mathbb{Z}_p$ be some generator of this subgroup, i.e., an element of order $2n$. The n *odd* powers $\omega^1, \omega^3, \ldots, \omega^{2n-1}$ are exactly the *primitive* $2n$th roots of unity, i.e., the roots of $\alpha^n + 1$.

In order to compute a polynomial product $\mathbf{a}_i \cdot \mathbf{x}_i$ modulo p and $\alpha^n + 1$, it suffices to compute only the n *primitive* Fourier coefficients of $\mathbf{a}_i$ and $\mathbf{x}_i$, i.e., the values $\mathbf{a}_i(\omega^1), \mathbf{a}_i(\omega^3), \ldots, \mathbf{a}_i(\omega^{2n-1})$, and likewise for $\mathbf{x}_i$. The primitive coefficients can be computed all at once by preprocessing the input and then applying an n-dimensional FFT (which uses half the space), as described in the algorithm from Section 1.1. Furthermore, because the field $\mathbb{Z}_p$ has roots of unity, the FFT can be performed over $\mathbb{Z}_p$ using the nth primitive root of unity ω^2, instead of over $\mathbb{C}$.[1]

In addition to using an FFT, other significant optimizations are possible when computing Expression (1). First, because the multipliers $\mathbf{a}_i$ are fixed in advance and determined uniquely by their primitive Fourier coefficients, we can simply store and work with their Fourier representation. Additionally, because the FFT is linear and a bijection, there is no need to even apply an *inverse* FFT. In other words, the value of Expression (1) is correctly and uniquely determined by summing the Fourier representations of $\mathbf{a}_i \cdot \mathbf{x}_i$. Combining all these observations, we are left with the high-level algorithm as described in Section 1.1, which we implement (using additional optimizations) in Section 3.

2.2 Concrete Parameters

In this paper we primarily study one family of SWIFFT compression functions, obtained by choosing concrete values for the parameters n, m, and p as follows:

$$n = 64, \quad m = 16, \quad p = 257.$$

[1]Performing an FFT over $\mathbb{Z}_p$ rather than $\mathbb{C}$ is often called a *number theoretic transform* (NTT) in the literature; however, we will retain the FFT terminology due to broad familiarity.

For these parameters, any fixed compression function in the family takes a binary input of length $mn = 1024$ bits (128 bytes), to an output in the range $\mathbb{Z}_p^n$, which has size $p^n = 257^{64} \approx 2^{512}$. An output in $\mathbb{Z}_p^n$ can easily be represented using 528 bits (66 bytes). Other unambiguous representations (using > 512 bits) are also possible; the representation does not affect security.

We now briefly explain our choice of parameters. The first consideration is the security of the compression function. As we will explain in the security analysis of Section 5, the function corresponds to a subset-sum from mn bits to roughly $n \lg p$ bits. We first set the constraints $mn = 1024$ and $n \lg p \approx 512$, because solving such subset-sum problems appears to be intractable. In order for our proofs of security to go through, we also need the polynomial $\alpha^n + 1$ to be irreducible over $\mathbb{Z}[\alpha]$, which is true if and only if n is a power of 2. (If a *reducible* polynomial is used, actual attacks can become possible, as we show in Section 5.3 for similar functions in the literature.)

Next, we optimize the running time and space of the function by choosing n to be large, and p and m to be small (subject to the above constraints). As discussed above, the Fast Fourier Transform is most efficiently and conveniently computed when p is prime and $p - 1$ is a multiple of $2n$. Our choice of n is the largest power of 2 that admits such a p.

Finally, to fix one concrete function from the family, the multipliers $\mathbf{a}_i$ should be chosen uniformly and independently at random from the ring R; this is equivalent to choosing their primitive Fourier coefficients uniformly and independently at random from $\mathbb{Z}_p$. We note that the multipliers (or their Fourier coefficients) should be chosen using "trusted randomness," otherwise it may be possible to embed a "backdoor" in the resulting function. For example, one might derive the multipliers using some deterministic transformation on the digits of π.

3 Implementation

Our implementation uses two main techniques for achieving high performance, both relating to the structure of the Fast Fourier Transform (FFT) algorithm. The first observation is that the input to the FFT is a *binary* vector, which limits the number of possible input values (when restricting our view to a small portion of the input). This allows us to precompute and store the results of several initial iterations of the FFT in a lookup table. The second observation is that the FFT algorithm consists of operations repeated in parallel over many pieces of data, for which modern microprocessors have special-purpose instruction sets.

Recall the parameters $n = 64$, $m = 16$, and modulus $p = 257$. Let ω be a 128th root of unity in $\mathbb{Z}_p = \mathbb{Z}_{257}$, i.e., an element of order $128 = 2n$. (We will see later that it is convenient to choose $\omega = 42$, but most of the discussion is independent from the choice of ω.)

The compression function takes an $mn = 1024$-bit input, viewed as $m = 16$ binary vectors $\mathbf{x}_0, \ldots, \mathbf{x}_{15} \in \{0,1\}^{64}$. (For convenience, entries of a vector or sequence are numbered starting from 0 throughout this section.) The function first processes each vector $\mathbf{x}_j$, multiplying its ith entry by ω^i (for $i = 0, \ldots, 63$), and then computing the Fourier transform of the resulting vector using ω^2 as a 64th root of unity. More precisely, each input vector $\mathbf{x}_j \in \{0,1\}^{64}$ is mapped to $\mathbf{y}_j = F(\mathbf{x}_j)$, where $F \colon \{0,1\}^{64} \to \mathbb{Z}_{257}^{64}$ is the function

$$F(\mathbf{x})_i = \sum_{k=0}^{63} (x_k \cdot \omega^k) \cdot (\omega^2)^{i \cdot k} = \sum_{k=0}^{63} x_k \cdot \omega^{(2i+1)k}. \tag{2}$$

The final output $\mathbf{z}$ of the compression function is then obtained by computing 64 distinct linear combinations (modulo 257) across the ith entries of the 16 $\mathbf{y}_j$ vectors:

$$z_i = \sum_{j=0}^{15} a_{i,j} \cdot y_{i,j} \pmod{257},$$

where the $a_{i,j} \in \mathbb{Z}_{257}$ are the primitive Fourier coefficients of the fixed multipliers.

Computing F. The most expensive part of the computation is clearly the computation of the transformation F on the 16 input vectors $\mathbf{x}_j$, so we first focus on the efficient computation of F. Let $\mathbf{y} = F(\mathbf{x}) \in \mathbb{Z}_{257}^{64}$ for some $\mathbf{x} \in \{0,1\}^{64}$. Expressing the indices i, k from Equation (2) in octal as $i = i_0 + 8i_1$ and $k = k_0 + 8k_1$ (where $j_0, j_1, k_0, k_1 \in \{0, \ldots, 7\}$), and using $\omega^{128} = 1 \pmod{257}$, the ith component of $\mathbf{y} = F(\mathbf{x})$ is seen to equal

$$\begin{aligned} y_{i_0+8i_1} &= \sum_{k_0=0}^{7} (\omega^{16})^{i_1 \cdot k_0} \left(\omega^{(2i_0+1)k_0} \cdot \sum_{k_1=0}^{7} \omega^{8k_1(2i_0+1)} \cdot x_{k_0+8k_1} \right) \\ &= \sum_{k_0=0}^{7} (\omega^{16})^{i_1 \cdot k_0} \left(m_{k_0,i_0} \cdot t_{k_0,i_0} \right), \end{aligned}$$

where $m_{k_0,i_0} = \omega^{(2i_0+1)k_0}$ and $t_{k_0,i_0} = \sum_{k_1=0}^{7} \omega^{8k_1(2i_0+1)} x_{k_0+8k_1}$. Our first observation is that each 8-dimensional vector $\mathbf{t}_{k_0} = (t_{k_0,0}, t_{k_0,1}, \ldots, t_{k_0,7})$ can take only 256 possible values, depending on the corresponding input bits $x_{k_0}, x_{k_0+8}, \ldots, x_{k_0+8\cdot 7}$. Our implementation parses each 64-bit block of the input as a sequence of 8 bytes $X_0, \ldots, X_7$, where $X_{k_0} = (x_{k_0}, x_{k_0+8}, \ldots, x_{k_0+8\cdot 7}) \in \{0,1\}^8$, so that each vector $\mathbf{t}_{k_0}$ can be found with just a single table look-up operation $\mathbf{t}_{k_0} = T(X_{k_0})$, using a table T with 256 entries. The multipliers $\mathbf{m}_{k_0} = (m_{k_0,0}, \ldots, m_{k_0,7})$ can also be precomputed.

The value $\mathbf{y} = F(\mathbf{x})$ can be broken down as 8 (8-dimensional) vectors

$$\mathbf{y}_{i_1} = (y_{8i_1}, y_{8i_1+1}, \ldots, y_{8i_1+7}) \in \mathbb{Z}_{257}^{8}.$$

Our second observation is that, for any $i_0 = 0, \ldots, 7$, the i_0th component of $\mathbf{y}_{i_1}$ depends only on the i_0th components of $\mathbf{m}_{k_0}$ and $\mathbf{t}_{k_0}$. Moreover, the operations performed for every coordinate are exactly the same. This permits parallelizing the computation of the output vectors $\mathbf{y}_0, \ldots, \mathbf{y}_7$ using SIMD (single-instruction multiple-data) instructions commonly found on modern microprocessors. For example, Intel's microprocessors (starting from the Pentium 4) include a set of so-called SSE2 instructions that allow operations on a set of special registers each holding an 8-dimensional vector with 16-bit (signed) integer components. We only use the most common SIMD instructions (e.g., component-wise addition and multiplication of vectors), which are also found on most other modern microprocessors, e.g., as part of the AltiVec SIMD instruction set of the Motorola G4 and IBM G5 and POWER6. In the rest of this section, operations on 8-dimensional vectors like $\mathbf{m}_{k_0}$ and $\mathbf{t}_{k_0}$ are interpreted as scalar operations applied component-wise to the vectors, possibly in parallel using a single SIMD instruction.

Going back to the computation of $F(\mathbf{x})$, the output vectors $\mathbf{y}_{i_1}$ can be expressed as

$$\mathbf{y}_{i_1} = \sum_{k_0=0}^{7} (\omega^{16})^{i_1 \cdot k_0} (\mathbf{m}_{k_0} \cdot \mathbf{t}_{k_0}).$$

Our third observation is that the latter computation is just a sequence of 8 component-wise multiplications $\mathbf{m}_{k_0} \cdot \mathbf{t}_{k_0}$, followed by a single 8-dimensional Fourier transform using ω^{16} as an 8th root of unity in $\mathbb{Z}_{257}$. The latter can be efficiently implemented using a standard FFT network consisting of just 12 additions, 12 subtractions and 5 multiplications.

Optimizations relating to $\mathbb{Z}_{257}$. One last source of optimization comes from two more observations that are specific to the use of 257 as a modulus, and the choice of $\omega = 42$ as a 128th root of unity. One observation is that the root used in the 8-dimensional FFT computation equals $\omega^{16} = 2^2 \pmod{257}$. So, multiplication by $(\omega^{16}), (\omega^{16})^2$ and $(\omega^{16})^3$, as required by the FFT, can be simply implemented as left bit-shift operations (by 2, 4, and 6 positions, respectively). Moreover, analysis of the FFT network shows that modular reduction can be avoided (without the risk of overflow using 16-bit arithmetic) for most of the intermediate values. Specifically, in our implementation, modular reduction is performed for only 3 of the intermediate values. The last observation is that, even when necessary to avoid overflow, reduction modulo 257 can be implemented rather cheaply and using common SIMD instructions, e.g., a 16-bit (signed) integer can be reduced to the range $\{-127, \ldots, 383\}$ using $x \equiv (x \wedge 255) - (x \gg 8) \bmod 257$, where $\wedge$ is the bit-wise "and" operation, and $\gg 8$ is a right-shift by 8 bits.

Summary and performance. In summary, function F can be computed with just a handful of table look-ups and simple SIMD instructions on 8 dimensional vectors. The implementation of the remaining part of the computation of the compression function (i.e., the scalar products between $y_{i,j}$ and $a_{i,j}$) is straightforward, keeping in mind that this part of the computation can also be parallelized using SIMD instructions, and that reduction modulo 257 is rarely necessary during the intermediate steps of the computation due to the use of 16-bit (or larger) registers.

We implemented and tested our function on a 3.2GHz Intel Pentium 4. The implementation was written in C (using the Intel intrinsics to instruct the compiler to use SSE2 instructions), and compiled using gcc version 4.1.2 (compiler flags -O3) on a PC running under Linux kernel 2.6.18. Our tests show that our basic compression function can be evaluated in 1.5 μs on the above system, yielding a throughput close to 40 MB/s in a standard chaining mode of operation. For comparison, we tested SHA256 on the same system using the highly optimized implementation in `openssl` version 0.9.8 (using the `openssl speed` benchmark), yielding a throughput of 47 MB/s when run on 8KB blocks.

Further optimizations. We remark that our implementation does not yet take advantage of all the potential for parallelism. In particular, we only exploited SIMD-level parallelism in individual evaluations of the transformation function F. Each evaluation of the compression function involves 16 applications of F, and subsequent multiplication of the result by the coefficients $a_{i,j}$. These 16 computations are completely independent, and can be easily executed in parallel on a multicore microprocessor. Our profiling data shows that the FFT computations and multiplication by $a_{i,j}$ currently account for about 90% of the running time. So, as multicore processors become more common, and the number of cores available on a processor increases, one can expect the speed of our function to grow almost proportionally to the number of cores, at least up to 16 cores. Finally, we point out that FFT networks are essentially "optimally parallelizable," and that our compression function has extremely small circuit depth, allowing it to be computed extremely fast in customized hardware.

4 Properties of SWIFFT

Here we review a number of statistical and cryptographic properties that are often desirable in hash functions, and discuss which properties our functions do and do not satisfy.

4.1 Statistical Properties

Here we review a number of many well-known and useful *statistical* properties that are often desirable in a family of hash functions, in both cryptographic and non-cryptographic applications (e.g., hash tables, randomness generation). All of these statistical properties can be proved *unconditionally*, i.e., they do not rely on any unproven assumptions about any computational problems.

Universal hashing. A family of functions is called *universal* if, for any fixed distinct x, x', the probability (over the random choice of f from the family) that $f(x) = f(x')$ is the inverse of the size of the range. It is relatively straightforward to show that our family of compression functions is universal (this property is used implicitly in the proofs for the statistical properties below).

Regularity. A function f is said to be *regular* if, for an input x chosen uniformly at random from the domain, the output $f(x)$ is distributed uniformly over the range. More generally, the function is ϵ-regular if its output distribution is within statistical distance (also known as variation distance) ϵ from uniform over the range. The only randomness is in the choice of the input.

As first proved in [13], our family of compression functions is regular in the following sense: all but an ϵ fraction of functions f from the family are ϵ-regular, for some negligibly small ϵ. The precise concrete value of ϵ is determined by the particular parameters (n, m, p) of the family.

Randomness extraction. Inputs to a hash function are often not chosen *uniformly* from the domain, but instead come from some non-uniform "real-world" distribution. This distribution is usually unknown, but may reasonably be assumed to have some amount of uncertainty, or *min-entropy.* For hash tables and related applications, it is usually desirable for the outputs of the hash function to be distributed uniformly (or as close to uniformly as possible), even when the inputs are not uniform. Hash functions that give such guarantees are known as *randomness extractors*, because they "distill" the non-uniform randomness of the input down to an (almost) uniformly-distributed output. Formally, randomness extraction is actually a property of a *family* of functions, from which one function is chosen at random (and obliviously to the input).

The proof of regularity for our functions can be generalized to show that they are also good randomness extractors, for input distributions having enough min-entropy.

4.2 Cryptographic Properties

Here we discuss some well-known properties that are often desirable in cryptographic applications of hash functions, e.g., digital signatures. Under relatively mild assumptions, our functions satisfy several (but not all) of these cryptographic properties.

Informally, a function f is said to *one-way* if, given the value $y = f(x)$ for an x chosen uniformly at random from the domain, it is infeasible for an adversary to find *any* x' in the domain such that $f(x') = y$. It is *second preimage resistant* if, given both x and $y = f(x)$, it is infeasible to find a *different* $x' \neq x$ such that $f(x') = y$. These notions also apply to families of functions, where f is chosen at random from the family.

A family of functions is *target collision resistant* (also called *universal one-way*) if it is infeasible to find a second preimage of x under f, where x is first chosen by the adversary (instead of at random) and then the function f is chosen at random from the family. Finally, the family is fully *collision resistant* if it is infeasible for an adversary, given a function f chosen at random from the family, to find distinct x, x' such that $f(x) = f(x')$.

For hash functions, the notions above are presented in increasing order of cryptographic strength. That is, collision resistance implies target collision resistance, which in turn implies second preimage resistance, which in turn implies one-wayness. All of the above notions are *computational*, in that they refer to the infeasibility (i.e., computational difficulty) of solving some cryptographic problem. However, the concrete effort required to violate these security properties (i.e., the meaning of "infeasible") will vary depending on the specific security notion under consideration, and is discussed in more detail below in Section 5.

As shown in [17, 12], our family of compression functions is *provably* collision resistant (in an asymptotic sense), under a relatively mild assumption about the *worst-case* difficulty of finding short vectors in cyclic/ideal lattices. This in turn implies that the family is also one-way and second preimage resistant. In Section 5.1, we give a detailed discussion and interpretation of the security proofs. In Section 5.2, we discuss the best known attacks on the cryptographic properties of our concrete functions, and give estimates of their complexity.

4.3 Properties *Not Satisfied* by SWIFFT

For general-purpose cryptographic hash functions and in certain other applications, additional properties are often desirable. We discuss some of these properties below, but stress that our functions *do not satisfy these properties*, nor were they intended or designed to.

Pseudorandomness. Informally, a family of functions is *pseudorandom* if a randomly-chosen function from the family "acts like" a *truly random* function in its input-output behavior. More precisely, given (adaptive) *oracle access* to a function f, no adversary can efficiently distinguish between the case where (1) f is chosen at random from the given family, and (2) every output of f is uniformly random and independent of all other outputs. (The formal definition is due to [9].) We stress that the adversary's view of the function is limited to oracle access, and that the particular choice of the function from the family is kept secret.

Our family of functions is *not* pseudorandom (at least as currently defined), due to linearity. Specifically, for any function f from our family and any two inputs x_1, x_2 such that $x_1 + x_2$ is also a valid input, we have $f(x_1) + f(x_2) = f(x_1 + x_2)$. This relation is very unlikely to hold for a random function, so an adversary cannot easily distinguish our functions from random functions by querying the inputs x_1, x_2, and $x_1 + x_2$. However, this homomorphism might actually be considered as a useful *feature* of the function in certain applications (much like homomorphic encryption).

With additional techniques, it may be possible to construct a family of pseudorandom functions (under suitable lattice assumptions) using similar design ideas.

Random oracle behavior. Intuitively, a function is said to behave like a *random oracle* if it "acts like" a truly random function. This notion differs from pseudorandomness in that the function is *fixed* and *public*, i.e., its entire description is known to the adversary. Though commonly used, the notion of "behaving like a random oracle" cannot be defined precisely in any meaningful or achievable way. Needless to say, we do not claim that our functions behave like a random oracle.

5 Security Analysis

In this section, we interpret our asymptotic proofs of security for collision-resistance and the other claimed cryptographic properties. We then consider cryptanalysis of the functions for our specific choice of parameters, and review the best known attacks to determine concrete levels of security.

5.1 Interpretation of Our Security Proofs

As mentioned above, an asymptotic proof of one-wayness for SWIFFT was given in [13], and an asymptotic proof of collision-resistance (a stronger property) was given independently in [17] and [12]. As in most cryptography, security proofs must rely on some precisely-stated (but as-yet unproven) assumption. Our assumption, stated informally, is that finding relatively short nonzero vectors in n-dimensional *ideal lattices* over the ring $\mathbb{Z}[\alpha]/(\alpha^n + 1)$ is infeasible *in the worst case*, as n increases. (See [13, 17, 12] for precise statements of the assumption.)

Phrased another way, the proofs of security say the following. Suppose that our family of functions is not collision resistant; this means that there is an algorithm that, given a *randomly-chosen* function f from our family, is able to find a collision in f in some feasible amount of time T. The algorithm might only succeed on a small (but noticeable) fraction of f from the family, and may only find a collision with some small (but noticeable) probability. Given such an algorithm, there is also an algorithm that can *always* find a short nonzero vector in *any* ideal lattice over the ring $\mathbb{Z}[\alpha]/(\alpha^n + 1)$, in some feasible amount of time related to T and the success probability of the collision-finder. We stress that the best known algorithms for finding short nonzero vectors in ideal lattices require *exponential* time in the dimension n, in the worst case.

The importance of *worst-case* assumptions in lattice-based cryptography cannot be overstated. Robust cryptography requires hardness *on the average*, i.e., almost every instance of the primitive must be hard for an adversary to break. However, many lattice problems are heuristically *easy* to solve on "many" or "most" instances, but still appear hard in the worst case on certain "rare" instances. Therefore, worst-case security provides a very strong and meaningful guarantee, whereas ad-hoc assumptions on the average-case difficulty of lattice problems may be unjustified. Indeed, we are able to find collisions in the related compression function LASH [3] by falsifying its underlying (ad-hoc) average-case lattice assumption (see Section 5.3).

At a minimum, our asymptotic proofs of security indicate that there are no unexpected "structural weaknesses" in the design of SWIFFT. Specifically, violating the claimed security properties (in an asymptotic sense) would necessarily require new algorithmic insights about finding short vectors in *arbitrary* ideal lattices (over the ring $\mathbb{Z}[\alpha]/(\alpha^n + 1)$). In Section 5.3, we demonstrate the significance of our proofs by giving examples of two compression functions from the literature that look remarkably similar to ours, but which admit a variety of very easily-found collisions.

Connection to algebraic number theory. Ideal lattices are well-studied objects from a branch of mathematics called *algebraic number theory*, the study of number fields. Let n be a power of 2, and let $\zeta_{2n} \in \mathbb{C}$ be a primitive $2n$th root of unity over the complex numbers (i.e., a root of the polynomial $\alpha^n + 1$). Then the ring $\mathbb{Z}[\alpha]/(\alpha^n + 1)$ is isomorphic to $\mathbb{Z}[\zeta_{2n}]$, which is the *ring of integers* of the so-called *cyclotomic* number field $\mathbb{Q}(\zeta_{2n})$. Ideals in this ring of integers (more generally, in the ring of integers of any number field) map to n-dimensional lattices under what is known as the *canonical embedding* of the number field. These are exactly the ideal lattices for which we assume finding short vectors is difficult in the worst case.[2] Further connections between the complexity of lattice problems and algebraic number theory were given by Peikert and Rosen [18].

For the cryptographic security of our hash functions, it is important that the extra ring structure does not make it easier to find short vectors in ideal lattices. As far as we know, and despite being a known open question in algebraic number theory, there is no apparent way to exploit this algebraic structure. The best known algorithms for finding short vectors in ideal lattices are the same as those for general lattices, and have similar performance. It therefore seems reasonable to conjecture that finding short vectors in ideal lattices is infeasible (in the worst case) as the dimension n increases.

5.2 Known Attacks

We caution that our asymptotic proofs do not necessarily rule out cryptanalysis of specific parameter choices, or ad-hoc analysis of one *fixed* function from the family. To quantify the exact security of our functions, it is still crucially important to cryptanalyze our specific parameter choices and particular instances of the function.

A central question in measuring the security of our functions is the meaning of "infeasible" in various attacks (e.g., collision-finding attacks). Even though our functions have an output length of about $n \lg p$ bits, we do *not* claim that they enjoy a full $2^{n \lg p}$ "level of security" for one-wayness, nor a $2^{(n \lg p)/2}$ level of security for collision resistance. Instead, we will estimate concrete levels of security for our specific parameter settings. This is akin to security estimates for public-key primitives like RSA, where due to subexponential-time factoring algorithms, a 1024-bit modulus may offer only (say) a 2^{100} concrete level of security.

In Section 5.2.2, we describe how the currently best-known algorithm to find collisions in our functions takes time at least 2^{106} and requires almost as much space. In Section 5.2.3, we describe the best-known inversion attacks, which require about 2^{448} time (but a small amount space).

Throughout this section, it will be most convenient to cryptanalyze our functions using their algebraic characterization as described in Section 2.1, and in particular, Equation (1).

5.2.1 Connection to Subset Sum

A very useful view of our compression function is as a subset sum function in which the weights come from the additive group $\mathbb{Z}_p^n$, and are related algebraically.

An element $\mathbf{a}$ in the ring $R = \mathbb{Z}_p[\alpha]/(\alpha^n + 1)$ can be written as $a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1}$, which we can represent as a vector $(a_0, \ldots, a_{n-1}) \in \mathbb{Z}_p^n$. Because $\alpha^n \equiv -1$ in the ring R, the product of two polynomials $\mathbf{a}, \mathbf{x} \in R$ is represented by the matrix product of the square *skew-circulant* matrix

[2] In [13, 17, 12], the mapping from ideals to lattices is slightly different, involving the coefficient vectors of elements in $\mathbb{Z}[\zeta_{2n}]$ rather than the canonical embedding. However, both mappings are essentially the same in terms of lengths of vectors, and the complexity of finding short vectors is the same under both mappings.

of $\mathbf{a}$ with the vector representation of $\mathbf{x}$:

$$\mathbf{a} \cdot \mathbf{x} \in R \quad \leftrightarrow \quad \begin{bmatrix} a_0 & -a_{n-1} & \cdots & -a_1 \\ a_1 & a_0 & \cdots & -a_2 \\ \vdots & & \ddots & \\ a_{n-1} & a_{n-2} & \cdots & a_0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} \bmod p \tag{3}$$

Thus we can interpret Equation (1) (with fixed multipliers $\mathbf{a}_1, \ldots, \mathbf{a}_m$) as multiplying a fixed matrix $\mathbf{A} \in \mathbb{Z}_p^{n \times mn}$ by an input vector $\mathbf{x} \in \{0,1\}^{mn}$. The matrix $\mathbf{A}$ has the form

$$\mathbf{A} = [\mathbf{A}_1 | \cdots | \mathbf{A}_m] \tag{4}$$

where each $\mathbf{A}_i$ is the $n \times n$ skew-circulant matrix of $\mathbf{a}_i$. Ignoring for a moment the algebraic dependencies within each $\mathbf{A}_i$, this formulation is equivalent to a subset sum function over the group $\mathbb{Z}_p^n$. Indeed, the output of our function is just the sum of a subset of the mn column vectors of $\mathbf{A}$. And in fact, the fastest known algorithm for inverting (or finding collisions in) our function f is the same one that is used for solving the high density subset sum problem [23, 11]. We describe this algorithm next.

5.2.2 Generalized Birthday Attack

Finding a collision in our function is equivalent to finding a nonzero $\mathbf{x} \in \{-1,0,1\}^{mn}$ such that

$$\mathbf{A}\mathbf{x} = \mathbf{0} \bmod p \tag{5}$$

where $\mathbf{A}$ is as in Equation (4). This is because if we find a $\{-1,0,1\}$-combination of the columns of $\mathbf{A}$ that sums to $\mathbf{0} \bmod p$, the subset of the columns corresponding to the -1s collides with the subset corresponding to the 1s. We will now describe an algorithm for finding such an $\mathbf{x}$ for the specific parameters $n = 64$, $m = 16$, $p = 257$. Our goal is to provide a *lower bound* on the running time of the most efficient known algorithm for breaking our function. Therefore the analysis of the function will be fairly conservative.

Given a 64×1024 matrix $\mathbf{A}$ whose coefficients are in $\mathbb{Z}_{257}$, we proceed as follows:

1. Randomly break up the 1024 column vectors of $\mathbf{A}$ into 16 groups of 64 vectors each.
2. From each group, create a list of 3^{64} vectors where each vector in the list is a different $\{-1,0,1\}$-combination of the vectors in the group.

We now have 16 lists each containing $3^{64} \approx 2^{102}$ vectors in $\mathbb{Z}_{257}^{64}$. Notice that if we are able to find one vector from each list such that their sum is the zero vector, then we can solve Equation 5.

Finding one vector from each list such that the sum is $\mathbf{0}$ is essentially the k-list problem that was studied by Wagner [23], and is also related to the technique used by Blum *et al* [5] for solving the parity problem in the presence of noise. The idea is to use the lists to obtain new lists of vectors that are $\{-1,0,1\}$-combinations of $\mathbf{A}$'s columns, but which have many coordinates that are 0. We then continue forming lists in which the vectors have more and more coordinates equal to 0. More precisely, we continue with the algorithm in the following way:

3. Pair up the 16 lists in an arbitrary way.

4. For each pair of lists (L_i, L_j), create a new list $L_{i,j}$ such that every vector in $L_{i,j}$ is the sum of one vector from L_i and one vector from L_j, and the first 13 positions of the vector are all 0 modulo 257.

There are a total of $257^{13} \approx 2^{104}$ different values that a vector in $\mathbb{Z}_{257}^{64}$ can take in its first 13 entries. Since the lists L_i and L_j each contain $3^{64} \approx 2^{102}$ vectors, there are a total of 2^{204} possible vectors that could be in $L_{i,j}$. If we heuristically assume that each of the $257^{13} \approx 2^{104}$ possible values of the first 13 coordinates are equally likely to occur[3], then we expect the list $L_{i,j}$ to consist of $2^{204} \cdot 2^{-104} = 2^{100}$ vectors whose first 13 coordinates are all 0. For convenience, we will assume that the lists have 2^{102} vectors (this again is a conservative assumption that is in the algorithm's favor).

At the end of Step 4, we have 8 lists, each with 2^{102} vectors in $\mathbb{Z}_{257}^{64}$ whose first 13 coordinates are zero. We can now pair up these 8 lists and create 4 lists of 2^{102} vectors whose first 26 coordinates are zero. We continue until we end up with one list of 2^{102} elements whose first 52 coordinates are zero. This means that only the last 12 coordinates of these vectors may be nonzero. If the vectors are randomly distributed in the last 12 coordinates, then there should be a vector which consists of all zeros (because there are only $257^{12} \approx 2^{96}$ possibilities for the last 12 coordinates).

Since we started out with 16 lists of 2^{102} elements, the running time of the algorithm is at least $16 \cdot 2^{102} = 2^{106}$. Notice that it also requires at least 2^{102} space.

5.2.3 Inversion Attacks

Consider the interpretation of our function where we multiply the matrix $\mathbf{A}$ from Equation (4) by an input vector $\mathbf{x} \in \{0,1\}^{1024}$. With extremely high probability, the matrix $\mathbf{A}$ has a set of 64 columns that are linearly independent (modulo p); without loss of generality, assume that they are the last 64 columns. We can then write $\mathbf{A}$ as $[\mathbf{B}|\mathbf{C}]$ where $\mathbf{B}$ consists of the first 960 columns and $\mathbf{C}$ is a 64×64 non-singular matrix. In an inversion attack, we have some specified $\mathbf{y} \in \mathbb{Z}_{257}^{64}$ and want to find an $\mathbf{x} \in \{0,1\}^{1024}$ such that $\mathbf{A}\mathbf{x} = \mathbf{y} \bmod 257$.

The best attack of which we are aware proceeds as follows:

1. Pick a random vector $\mathbf{x}_B \in \{0,1\}^{960}$.

2. Compute vector $\mathbf{x}_C = \mathbf{C}^{-1}(\mathbf{y} - \mathbf{B}\mathbf{x}_B) \bmod p$

3. If $\mathbf{x}_C \in \{0,1\}^{64}$, then output $\mathbf{x} = [\mathbf{x}_B|\mathbf{x}_C]$. Otherwise, start over.

The product $\mathbf{B}\mathbf{x}_B \bmod p$ is distributed almost-uniformly in $\mathbb{Z}_{257}^{64}$, due to the regularity property of our function. Then because the matrix $\mathbf{C}$ is non-singular, the product $\mathbf{C}^{-1}(\mathbf{y} - \mathbf{B}\mathbf{x}_B) \bmod p$ is almost-uniformly distributed in $\mathbb{Z}_{257}^{64}$. Therefore, the probability that $\mathbf{C}^{-1}(\mathbf{y} - \mathbf{B}\mathbf{x}_B) \bmod p$ is a binary vector is approximately $2^{64}/|\mathbb{Z}_{257}^{64}| \approx 2^{-448}$. Thus, we can expect to find a preimage for $\mathbf{y}$ in about 2^{448} time. (Everything above applies equally well to second preimage attacks.)

5.2.4 Lattice Attacks

Lattice reduction is a possible alternative way to find a nonzero vector $\mathbf{x} \in \{-1,0,1\}^{mn}$ that will satisfy Equation (5). If we think of the matrix $\mathbf{A}$ as defining a linear homomorphism from $\mathbb{Z}^{mn}$ to

[3]This is true if all the vectors in the lists are random and independent in $\mathbb{Z}_{257}^{64}$, but this is not quite the case. Nevertheless, since we are being conservative, we will assume that the algorithm will still work.

$\mathbb{Z}_p^n$, then the kernel of $\mathbf{A}$ is $\ker(\mathbf{A}) = \{\mathbf{y} \in \mathbb{Z}^{mn} : \mathbf{A}\mathbf{y} \equiv \mathbf{0} \bmod p\}$. Notice that $\ker(\mathbf{A})$ is a lattice of dimension mn, and a vector $\mathbf{x} \in \{-1, 0, 1\}^{mn}$ such that $\mathbf{A}\mathbf{x} \equiv \mathbf{0} \bmod p$ is a shortest nonzero vector in the ℓ_∞ or "max" norm of this lattice.

Because a basis for $\ker(\mathbf{A})$ can be computed efficiently given $\mathbf{A}$, finding a shortest nonzero vector (in the ℓ_∞ norm) of the lattice would yield a collision in our compression function. The lattice $\ker(\mathbf{A})$ shares many properties with the commonly occurring knapsack-type lattice (see, e.g., [16]). Our lattice is essentially a knapsack-type lattice with some additional algebraic structure. It is worthwhile to note that none of the well-known lattice reduction algorithms take advantage of the algebraic structure that arises here. Because the dimension 1024 of our lattice is too large for the current state-of-the-art reduction algorithms, breaking our function via lattice reduction would require some very novel idea to exploit the additional algebraic structure. As things stand right now, we believe that the generalized birthday technique described in the previous section provides a more efficient algorithm for finding collisions in our function.

Viewing the kernel as a lattice also leads naturally to a relaxed notion of "pseudo-collisions" in our function, which are defined in terms other norms, e.g., the Euclidean ℓ_2 norm or "Manhattan" ℓ_1 norm. Note that for any *actual* collision corresponding to an $\mathbf{x} \in \ker(\mathbf{A})$, we have $\mathbf{x} \in \{-1, 0, 1\}^{mn}$ and therefore the ℓ_2 norm of $\mathbf{x}$ is $\|\mathbf{x}\|_2 \leq \sqrt{mn}$. However, not every nonzero $\mathbf{x} \in \ker(\mathbf{A})$ with $\|\mathbf{x}\|_2 \leq \sqrt{mn}$ determines a collision in our function, because the entries of $\mathbf{x}$ may lie outside $\{-1, 0, 1\}$. We say that such an $\mathbf{x}$ is a *pseudo-collision* for the ℓ_2 norm. More generally, a pseudo-collision for any ℓ_p norm ($1 \leq p < \infty$) is defined to be a nonzero $\mathbf{x} \in \ker(\mathbf{A})$ such that $\|\mathbf{x}\|_p \leq (mn)^{1/p}$. The set of pseudo-collisions only grows as p decreases from ∞ to 1, so finding pseudo-collisions using lattice reduction might be easier in norms such as ℓ_2 or ℓ_1. Finding pseudo-collisions could be a useful starting point for finding true collisions.

5.3 Cryptanalysis of Similar Functions

In this section, we briefly discuss two other compression functions appearing in the literature that bear a strong resemblance to ours, but which *do not* have asymptotic proofs of collision resistance. In fact, these compression functions are *not* collision resistant, and admit quite simple collision-finding algorithms. The attacks are made possible by a structural weakness that stems from the use of *circulant* matrices, which correspond algebraically to rings that are *not integral domains*. Interestingly, integral domains are the crucial ingredient in the asymptotic proofs of collision resistance for our function [17, 12]. We believe that this distinction underscores the usefulness and importance of security proofs, especially *worst-case* hardness proofs for lattice-based schemes.

5.3.1 Micciancio's Cyclic One-Way Function

The provably *one-way* function described by Micciancio [13] is very similar to SWIFFT, and was the foundation for the subsequent collision-resistant functions on which this paper is based [17, 12]. Essentially, the main difference between Micciancio's function and the ones presented in [17, 12] is that the operations are performed over the ring $\mathbb{Z}_p[\alpha]/(\alpha^n - 1)$ rather than $\mathbb{Z}_p[\alpha]/(\alpha^n + 1)$. This difference, while seemingly minor, makes it almost trivial to find collisions, as shown in [17, 12].

Just like ours, Micciancio's function has an interpretation as the product of a matrix $\mathbf{A}$ (as in Equation (4)) and a vector $\mathbf{x} \in \{0, 1\}^{mn}$. The only difference is the matrices $\mathbf{A}_i$ from Equation (4) are *circulant*, rather than skew-circulant (i.e., just like Equation (3), but without negations).

Notice that in the product of a circulant matrix $\mathbf{A}_i$ with the all-1s vector $\mathbf{1}$, all of the entries are the same. Thus, for any circulant $\mathbf{A}_i$, the product $\mathbf{A}_i\mathbf{x}_i \bmod p$ can only take on p distinct values. There are 2^m ways to set each vector $\mathbf{x}_1, \ldots, \mathbf{x}_m$ to be either $\mathbf{0}$ or $\mathbf{1}$, but there are only p distinct values of the compression function $\mathbf{A}\mathbf{x} = \mathbf{A}_1\mathbf{x}_1 + \ldots + \mathbf{A}_m\mathbf{x}_m \bmod p$. Because $2^m > p$ (otherwise the function does not compress), some pair of distinct binary vectors are mapped to the same output. Such a collision can be found in linear time.

5.3.2 LASH Compression Function

LASH is a hash function that was presented at the second NIST hash function workshop [3]. Its compression function $f_\mathbf{H}$ takes an n-bit input $\mathbf{x} = \mathbf{x}_1|\mathbf{x}_2$ where $\mathbf{x}_1, \mathbf{x}_2 \in \{0,1\}^{n/2}$, and is defined as $f_\mathbf{H}(\mathbf{x}) = (\mathbf{x}_1 \oplus \mathbf{x}_2) + \mathbf{H}\mathbf{x} \bmod q$, where $\mathbf{H}$ is a "semi-circulant" $m \times n$ matrix whose entries are from the group $\mathbb{Z}_{256}$:

$$\mathbf{H} = \begin{bmatrix} a_0 & a_{n-1} & a_{n-2} & \cdots & a_1 \\ a_1 & a_0 & a_{n-1} & & a_2 \\ \vdots & & & \ddots & \\ a_{m-1} & a_{m-2} & a_{m-3} & \ldots & a_m \end{bmatrix}.$$

The values $a_0, \ldots, a_{n-1} \in \mathbb{Z}_{256}$ are essentially chosen at random (actually, according to a weak kind of pseudorandom generator).

Contrary to a claim in [3], finding collisions in the compression function $f_\mathbf{H}$ is actually trivial (for random values $a_0, \ldots, a_{n-1}$). Notice that all the rows of the matrix $\mathbf{H}$ have the same sum, and if this sum happens to be 0 mod 256 (which happens with probability 1/256 over the choice of the a_i), then we have

$$f(\mathbf{0}) = 0 + \mathbf{H} \cdot \mathbf{0} = \mathbf{0} = 0 + \mathbf{H} \cdot \mathbf{1} = f(\mathbf{1}),$$

where $\mathbf{0}$ and $\mathbf{1}$ are all-0s and all-1s vectors, respectively.

When n is divisible by some power of 2 (e.g., $n = 640, 1024$ are proposed in [3]), other collisions may be easy to find as well (with some noticeable probability over the choice of the a_i). For example, the inputs $\mathbf{x} = 01 \cdots 01$ and $\mathbf{x}' = 10 \cdots 10$ will collide with probability 1/256 over the choice of the a_i, because $\mathbf{H}\mathbf{x}$ and $\mathbf{H}\mathbf{x}'$ consist of two repeated values (the sum of the even-indexed a_is and the sum of the odd-indexed a_is). Many other kinds of collision are also possible, essentially corresponding to the factorization of the polynomial $\alpha^n - 1$ over $\mathbb{Z}[\alpha]$.

References

[1] M. Ajtai. Generating hard instances of lattice problems. In *STOC*, pages 99–108, 1996.

[2] T. Baritaud, H. Gilbert, and M. Girault. FFT hashing is not collision-free. In *EUROCRYPT*, pages 35–44, 1992.

[3] K. Bentahar, D. Page, J. Silverman, M. Saarinen, and N. Smart. Lash. Technical report, 2nd NIST Cryptographic Hash Function Workshop, 2006.

[4] E. Biham, R. Chen, A. Joux, P. Carribault, W. Jalby, and C. Lemuet. Collisions of SHA-0 and reduced SHA-1. In *EUROCRYPT*, 2005.

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

[6] J. Cai and A. Nerurkar. An improved worst-case to average-case connection for lattice problems. In *FOCS*, pages 468–477, 1997.

[7] J. Daemen, A. Bosselaers, R. Govaerts, and J. Vandewalle. Collisions for Schnorr's hash function FFT-hash presented at crypto '91. In *ASIACRYPT*, 1991.

[8] O. Goldreich, S. Goldwasser, and S. Halevi. Collision-free hashing from lattice problems. Technical Report TR-42, ECCC, 1996.

[9] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[10] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU: A ring-based public key cryptosystem. In *ANTS*, pages 267–288, 1998.

[11] V. Lyubashevsky. The parity problem in the presence of noise, decoding random linear codes, and the subset sum problem. In *RANDOM*, pages 378–389, 2005.

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

[13] D. Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions from worst-case complexity assumptions. *Computational Complexity.* (To appear. Preliminary version in FOCS 2002).

[14] D. Micciancio. Almost perfect lattices, the covering radius problem, and applications to Ajtai's connection factor. *SIAM J. on Computing*, 34(1):118–169, 2004.

[15] D. Micciancio and O. Regev. Worst-case to average-case reductions based on Gaussian measures. *SIAM J. on Computing*, 37(1):267–302, 2007.

[16] P. Nguyen and D. Stehlé. Lll on the average. In *ANTS*, pages 238–256, 2006.

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

[18] C. Peikert and A. Rosen. Lattices that admit logarithmic worst-case to average-case connection factors. In *STOC*, pages 478–487, 2007. Full version in ECCC Report TR06-147.

[19] C. P. Schnorr. FFT-hash, an efficient cryptographic hash function. In *Crypto Rump Session*, 1991.

[20] C. P. Schnorr. FFT–Hash II, efficient cryptographic hashing. In *EUROCRYPT*, pages 45–54, 1992.

[21] C.P. Schnorr and Serge Vaudenay. Parallel FFT-hashing. In *Fast Software Encryption*, pages 149–156, 1993.

[22] S. Vaudenay. FFT-Hash-II is not yet collision-free. In *CRYPTO*, pages 587–593, 1992.

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

[24] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis for hash functions MD4 and RIPEMD. In *EUROCRYPT*, 2005.

[25] X. Wang and H. Yu. How to break MD5 and other hash functions. In *EUROCRYPT*, 2005.