SWIFFTX: A Proposal for the SHA-3 Standard

Yuriy Arbitman Gil Dogon[*] Vadim Lyubashevsky[†] Daniele Micciancio[‡]
Chris Peikert[§] Alon Rosen[¶]

October 30, 2008

Abstract

This report describes the SWIFFTX hash function. It is part of our submission package to the SHA-3 hash function competition.

The SWIFFTX compression functions have a simple and mathematically elegant design. This makes them highly amenable to analysis and optimization. In addition, they enjoy two unconventional features:

Asymptotic proof of security: it can be formally proved that finding a collision in a randomly-chosen compression function from the SWIFFTX family is at least as hard as finding short vectors in cyclic/ideal lattices in the *worst case*.

High parallelizability: the compression function admits efficient implementations on modern microprocessors. This can be achieved even without relying on multi core capabilities, and is obtained through a novel cryptographic use of the *Fast Fourier Transform* (FFT).

The main building block of SWIFFTX is the SWIFFT family of compression functions, presented in the 2008 workshop on Fast Software Encryption (Lyubashevsky et al., FSE'08). Great care was taken in making sure that SWIFFTX does not inherit the major shortcoming of SWIFFT – linearity – while preserving its provable collision resistance.

The SWIFFTX compression function maps 2048 input bits to 520 output bits. The mode of operation that we employ is HAIFA (Biham and Dunkelman, 2007), resulting in a hash function that accepts inputs of any length up to $2^{64}-1$ bits, and produces message digests of the SHA-3 required lengths of 224, 256, 384 and 512 bits.

[*]Mobileye Inc., Israel.

[†]Tel-Aviv University, Israel. E-mail `vlyubash@cs.ucsd.edu`.

[‡]University of California at San Diego. E-mail: `daniele@cs.ucsd.edu`. Supported by the National Science Foundation under Grant CCF-0634909. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[§]SRI International. E-mail: `cpeikert@alum.mit.edu`. Supported by the National Science Foundation under Grants CNS-0716786 and CNS-0749931. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

[¶]IDC Herzliya, Israel. E-mail: `alon.rosen@idc.ac.il`.

1 Introduction

In this report we describe the SWIFFTX cryptographic hash function. The main goals in the design of SWIFFTX were

- to define a function with a very regular structure and clean mathematical description, which encourages cryptanalytic efforts;
- to use innovative but principled design choices, that can be justified through rigorous mathematical proofs of security. This is true not only for the high level mode of operation, but remarkably also for the underlying compression function;
- to allow very efficient implementations on modern microprocessors, with a substantial amount of parallelism that is easily exploitable both on single-core (through the use of "single-instruction multiple-data" SIMD operations) and multi-core systems, without the need for specialized hardware. (Though very fast hardware implementation is clearly also possible.) We remark that efficiency on modern microprocessors is not achieved through the use of 64-bit arithmetic. In fact, most computation performed by SWIFFTX consists of arithmetic modulo 257, which can be reasonably implemented on 8-bit processors.

The main component of our function is the SWIFFTX compression function (described in detail in the rest of this document) which maps 2048-bit inputs to 520-bit outputs. Then, we use HAIFA [1] as an iterative framework to obtain a hash function that takes as input an arbitrary number of bits and produces a 520 bit output. The various digest sizes required by NIST are obtained by a final post-processing stage (similar, but not identical, to the basic compression function) that maps 520 bits to the desired output of $512, 384, 256$, or 224 bits.

1.1 Mode of operation

The HAIFA framework was selected to fix many of the flaws of the traditional Merkle-Damgard construction, and to provide much stronger security guarantees against pre-image and second-preimage attacks. However, other iterative frameworks are conceivable, and the main innovation of our proposal is in the design and selection of the SWIFFTX compression function.

For a detailed description of the design and rationale of the HAIFA framework we refer to the original paper [1]. Because of the structure of the SWIFFTX hash function, it supports HMAC, HMAC as pseudo-random function and randomized hashing in the same way as the SHA family. In the rest of this document we focus on the design of the SWIFFTX compression function and the post-processing stage. See Figure 1 for a visual description of how the SWIFFTX compression function fits within the HAIFA framework. As specified by the HAIFA framework, we use a single 520-bit IV to produce individual IV's for each of the digest lengths. The IV that we chose was generated from the digits of the decimal expansion of e (see section C for the actual IV value).

1.2 Compression function

The SWIFFTX compression fuction is based on the recently proposed SWIFFT compression function of [4, 3, 8]. A remarkable property of SWIFFT is that it admits an *asymptotic proof of security* (against collision-finding and preimage attacks) under *worst-case* assumptions about the complexity of certain lattice problems. (See Section 4.1 for details about these provable security

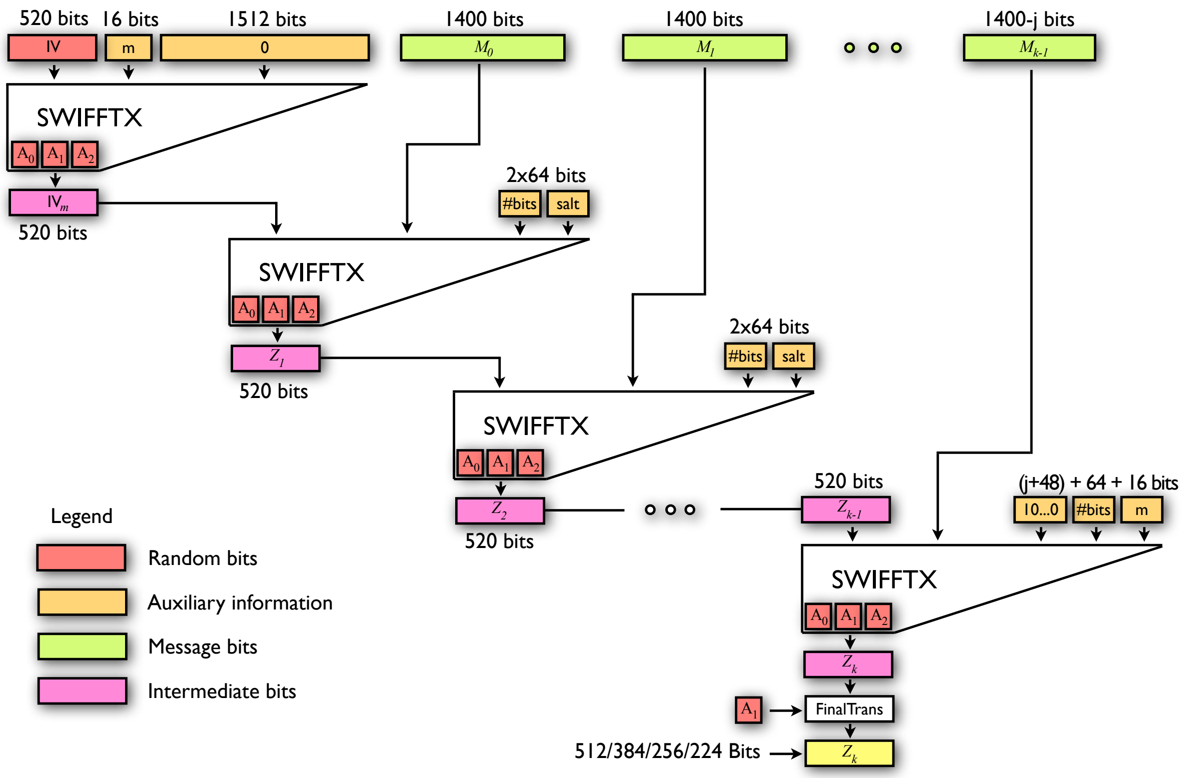

Figure 1: The SWIFFTX hash function.

properties.) The SWIFFT compression function has some features (e.g., linearity) that, while useful in certain applications [6, 5], are undesirable in a general purpose hash function as sought by NIST. These features also make SWIFFT susceptible to k-list/generalized birthday attacks [10, 2], which substantially degrades the quantitative strength of the function. (In particular, the cost of collision-finding attacks on SWIFFT is, by design, much less that $2^{n/2}$, where n is the output size.)

Our SWIFFTX compression function uses SWIFFT as a building block, in such a way that

- the linearity properties are completely disrupted, substantially strengthening SWIFFT against k-list attacks, and achieving other desirable pseudo-randomness properties;
- the asymptotic provable security guarantees of the SWIFFT compression function against collision-finding and preimage attacks (described in theoretical works [4, 3, 8]) are maintained.

The SWIFFTX function is comprised of three layers, whose functionality and design rationale are summarized here and will be elaborated on in Section 2:

1. An inner layer of 3 parallel invocations of SWIFFT on the same input but independent and distinct "randomizers."

   The purpose of this layer is to serve as a *one-way function* that is sufficiently hard to invert. We use 3 parallel invocations of SWIFFT so that the output is large enough to resist k-list/generalized birthday attacks for inversion and collision-finding. This layer also provides a small amount of compression, but that is ancillary; from a design perspective, any (almost) injective one-way function with strong diffusion/mixing properties would suffice for our goals in this layer.

2. The result produced by the inner layer can be interpreted as a sequence of numbers modulo 257. An intermediate layer converts the output of the inner layer from base 257 to binary, and applies S-boxes that are simple permutations on 8 bits.

   The purpose of this layer is to destroy the linear homomorphism of the inner layer of basic SWIFFT functions, which is a necessary condition for pseudorandomness and defeating k-list attacks. The primary goal of this layer is to significantly increase the degree of the entire SWIFFTX function, viewed as a polynomial over either $GF(2)$ or $GF(257)$. The requirements on the S-boxes are therefore quite modest; in particular, they need not be designed to resist linear or differential cryptanalysis, but only to have reasonably large degree over $GF(2)$. Random permutations suffice for this purpose.

3. An outer layer consisting of a single invocation of SWIFFT on the binary output of the intermediate layer. Here we reuse some of the randomizer matrices from the first step; this does not have any impact on the security proof.

   This final layer serves as the main compression step of SWIFFTX.

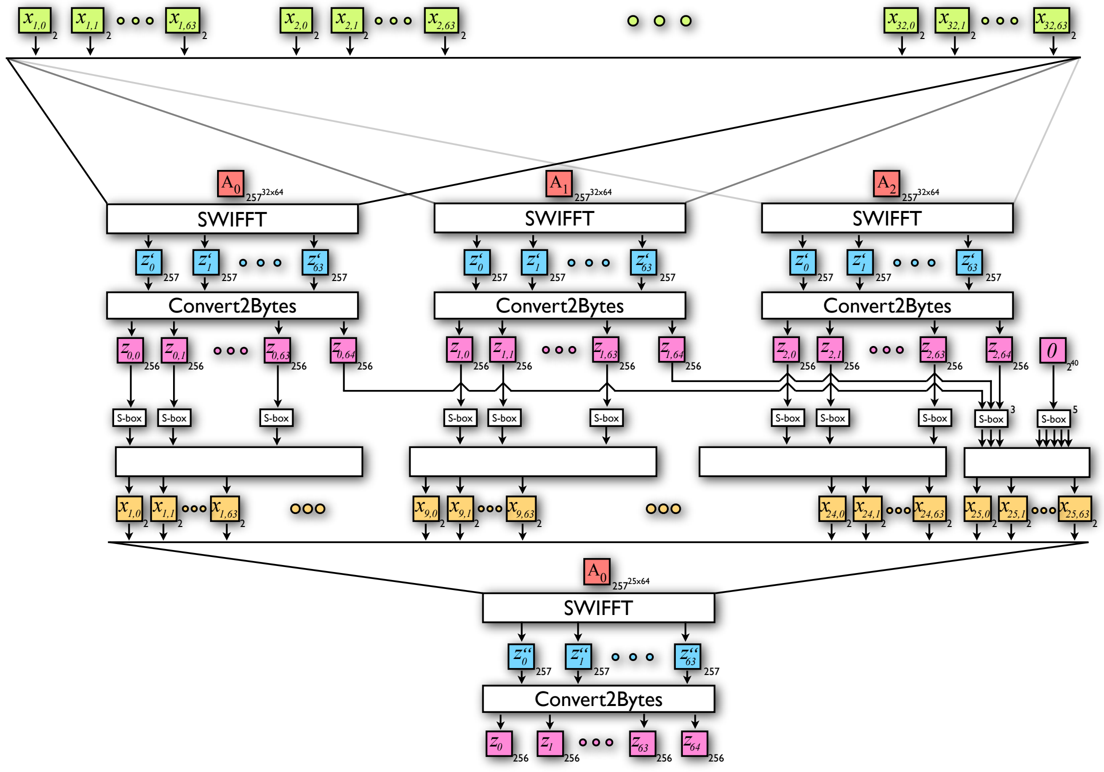

Figure 2: The SWIFFTX compression function.

Note that the outer layer by itself is not strongly collision-resistant, due to k-list attacks of moderate complexity, as described in [4]. However, the one-wayness of the inner layer prevents any collisions discovered in the outer layer from being converted into collisions in the entire function.

From an asymptotic security perspective, notice that because the intermediate layer is a permutation, finding collisions (respectively, preimages) in SWIFFTX implies finding collisions (respectively, preimages) in at least one of the four SWIFFT components. For the reason, the asymptotic proofs of security for SWIFFT also apply to the SWIFFTX design.

2 SWIFFTX Design

In this section we describe the SWIFFTX compression function, giving a modular description of its main components, which are: the SWIFFT function (Section 2.1), the ConvertToBytes procedure (Section 2.2), an S-box (Section 2.3). In addition, there is a FinalTransform procedure (Section 2.5) that is called once to produce the final digest.

Pseudocode for each of the components is included in Appendix A. We stress that this pseudocode (and reference implementation) is not intended to correspond to an *efficient* implementation. SWIFFTX has a mathematically elegant structure, and as such, it can be described and implemented in several different but functionally equivalent ways. Our reference implementation was chosen as the *simplest* possible description of the function, in order to provide a good reference for cryptanalysis and verification of more complex (and efficient) implementations.

2.1 SWIFFT

The main component of the SWIFFTX compression function is the SWIFFT function (Figure 3 and Section A.2). This function takes as input either $m = 32$ (or in one special case, $m = 25$) 64-bit words $x_1, \ldots, x_m$ for a total of 2048 (or 1600) bits, and outputs 64 elements $z'_0, \ldots, z'_{63} \in \mathbb{Z}_{257} = \{0, \ldots, 256\}$. The function is indexed by either 2048 or 1600 fixed "randomizer" elements $a_{1,0}, \ldots, a_{m,63} \in \mathbb{Z}_{257}$, which are taken to be uniformly random integers modulo 257. (For concreteness and to ensure the lack of trapdoors, we generated these randomizers using the decimal expansion of π (see Section A.3), but any other random way to choose them is also acceptable.)

SWIFFT can be described mathematically as follows. Let

$$\mathrm{rev}\colon \{0, \ldots, 63\} \to \{0, \ldots, 63\}$$

be the "bit-reversal" function that on input a 6-bit binary number $b_5b_4b_3b_2b_1b_0$ outputs the number with binary representation

$$\mathrm{rev}(b_5b_4b_3b_2b_1b_0) = b_0b_1b_2b_3b_4b_5.$$

On input $x_1, \ldots, x_m$, where each $x_i = x_{i,0} \cdots x_{i,63}$ consists of 64 bits, SWIFFT outputs the sequence of values $z'_0, \ldots, z'_{63} \in \mathbb{Z}_{257}$, where

$$z'_i = \sum_{j=1}^{m} a_{j,i} \sum_{k=0}^{63} x_{j,\mathrm{rev}(k)} \cdot \omega^{(2i+1)k}$$

where $\omega = 42$ and all the arithmetic is performed modulo 257.

This computation can equivalently be described as follows:

- Permute the bits $x_{j,0}, \ldots, x_{j,63}$ in each word x_j according to the bit-reversal function to obtain $x_{j,\mathrm{rev}(0)}, \ldots, x_{j,\mathrm{rev}(63)}$.
- Interpret each (permuted) word $x_{j,\mathrm{rev}(0)}, \ldots, x_{j,\mathrm{rev}(63)}$ as the coefficients of a polynomial of degree (at most) 63:

$$p_j(\alpha) = x_{j,\mathrm{rev}(0)} + x_{j,\mathrm{rev}(1)} \cdot \alpha + \cdots + x_{j,\mathrm{rev}(63)} \cdot \alpha^{63}.$$

- Evaluate each polynomial $p_j(\cdot)$ on all the odd powers $\omega, \omega^3, \omega^5, \ldots, \omega^{127}$ of $\omega = 42$, where all arithmetic is modulo 257.
- Multiply each of the resulting values $p_j(\omega^{2i+1})$ by $a_{j,i}$ and sum over all j, to obtain

$$z'_i = a_{1,i} \cdot p_1(\omega^{2i+1}) + \cdots + a_{m,i} \cdot p_m(\omega^{2i+1}).$$

Evaluating $p_j(\alpha)$ on all odd powers of ω can be performed in a number of ways. One way that admits an efficient implementation is by

1. pre-multiplying each degree-i coefficient of p_j by ω^i, then
2. evaluating the resulting polynomials on all powers of ω^2.

The second operation above can be performed efficiently using the Fast Fourier Transform algorithm.

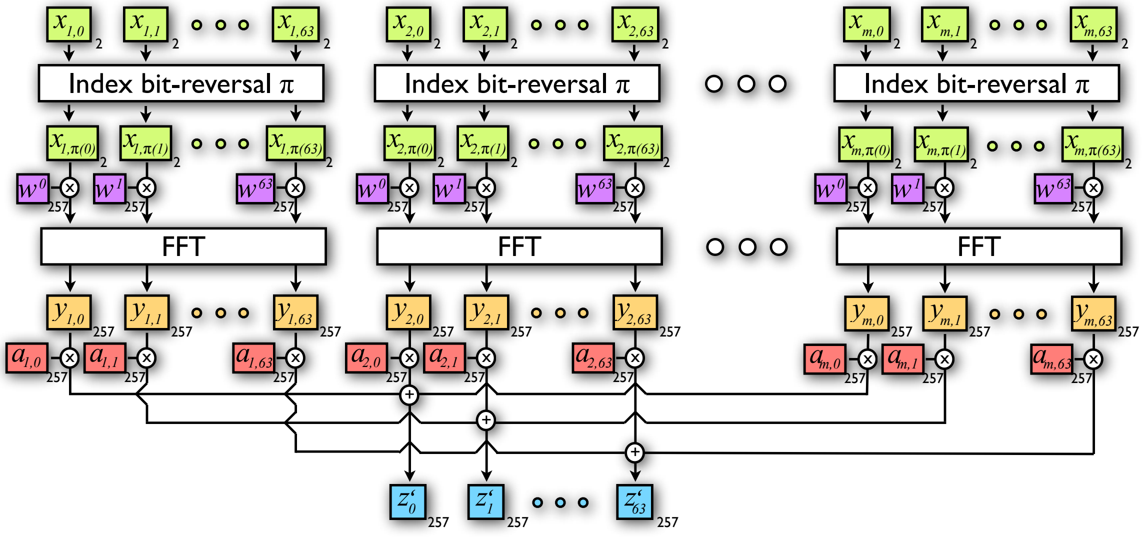

Figure 3: The SWIFFT function.

Our choice of parameters $\omega = 42$ and modulus 257 were dictated entirely by efficiency concerns. In fact, the cryptographic strength of the function is largely independent of the particular values (indeed, in the post-processing stage of SWIFFTX we use an instantiation of SWIFFT with a different modulus). Because 257 is prime, the ring of integers modulo 257 forms a finite field, whose multiplicative group is cyclic and has exactly 256 elements. In particular, any generator of the multiplicative group has order 256. All we need here is an element of order 128, which can be obtained by squaring any generator of the multiplicative group. The value $\omega = 42$ is one such element of order 128, and has additional properties that are convenient for highly-optimized implementations.

Full pseudocode for SWIFFT is given in Section A.2.

2.2 ConvertToBytes

Because the output of the SWIFFT function is comprised of elements in $\mathbb{Z}_{257}$, we need a function that converts them into binary quantities for further use. We perform a simple change of base from 257 to 256 by taking groups of 8 elements $z'_0, \ldots, z'_7 \in \mathbb{Z}_{257}$ and producing 8 elements $z_0, \ldots, z_7$ in $\mathbb{Z}_{256}$ and a bit $b \in \{0,1\}$ such that

$$\sum_{i=0}^{7} z'_i \cdot 257^i = \sum_{i=0}^{7} z_i \cdot 256^i + b \cdot 256^8.$$

We then take the bit b from 8 such groups and combine them together into one byte (see figure 4). Therefore, ConvertToBytes is an injective function mapping 64 elements of $\mathbb{Z}_{257}$ into 65 bytes.

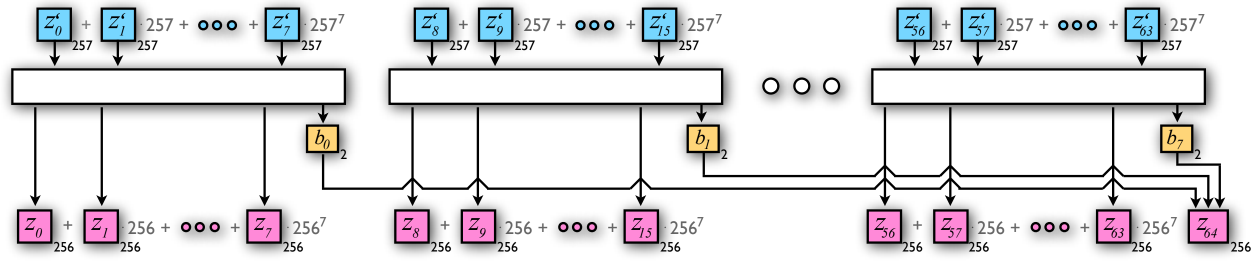
$z'_0 + z'_1 \cdot 257 + \cdots + z'_7 \cdot 257^7$
$z'_8 + z'_9 \cdot 257 + \cdots + z'_{15} \cdot 257^7$
$z'_{56} + z'_{57} \cdot 257 + \cdots + z'_{63} \cdot 257^7$
b_0
b_1
b_7
$z_0 + z_1 \cdot 256 + \cdots + z_7 \cdot 256^7$
$z_8 + z_9 \cdot 256 + \cdots + z_{15} \cdot 256^7$
$z_{56} + z_{57} \cdot 256 + \cdots + z_{63} \cdot 256^7$
z_{64}

Figure 4: The ConvertToBytes procedure.

2.3 S-Box

The linearity of the SWIFFT functions in the inner and outer layers of SWIFFTX is broken by the change of base performed by ConvertToBytes, as well as by an S-box (Figure 7). The S-box is a simple permutation over $\{0,1\}^8$, i.e., mapping one byte to one byte. To ensure the lack of a trapdoor, the S-box was constructed from the digits of e in a manner that ensures it is a permutation.

2.4 SWIFFTX Compression Function (putting everything together)

The SWIFFTX compression function takes as input 2048 bits and applies three SWIFFT functions with distinct randomizers. Notice that the FFT operation need only be done once for each of the input blocks, and can be reused across the three applications of SWIFFT. The output of the three SWIFFT functions is then fed to the ConvertToBytes function to obtain $3 \times 65 = 195$ bytes, then each of those bytes is fed into the S-box. We arrange these 195 bytes as in Figure 2 and append 5 bytes corresponding to S-box(0). The result is 200 bytes, or 1600 bits that are used as the input to the next SWIFFT. The output of this SWIFFT is fed to the ConvertToBytes function, and we end up with 520 bits, which are then either fed to the next compression function, or to the FinalTransform function, described below.

2.5 FinalTransform

While the output of SWIFFTX is almost regularly distributed over the domain $\mathbb{Z}_{257}^{64}$, the output of the entire hash function should be regularly distributed over $\mathbb{Z}_2^{512}$. When converted to 65 bytes using the ConvertToBytes function, the 520 resulting bits are statistically biased. Therefore, after the final block of the input has been processed, it is necessary to convert these 520 skewed bits into 512 uniformly-distributed bits. Because our main objective is to preserve the security proof, we perform an operation that "smooths" the output and is theoretically equivalent to evaluating one more SWIFFT function having a 520-bit input.

We extend the 520 bits to 576 bits by padding with zero bits, then break these into 9 groups of 64 bits. We treat each of the groups as a polynomial $\mathbf{x}_i$ of degree at most 63. We then use 576 randomizer elements that were already created and create 9 polynomials $\mathbf{p}_i$. We then compute $\mathbf{x}_0\mathbf{p}_0 + \ldots + \mathbf{x}_8\mathbf{p}_8$ over the ring $\mathbb{Z}_{256}[\alpha]/(\alpha^{64}+1)$. The result is a polynomial of degree 63 whose coefficients are elements modulo 256 (i.e. bytes), and therefore we have the required 512 bits.

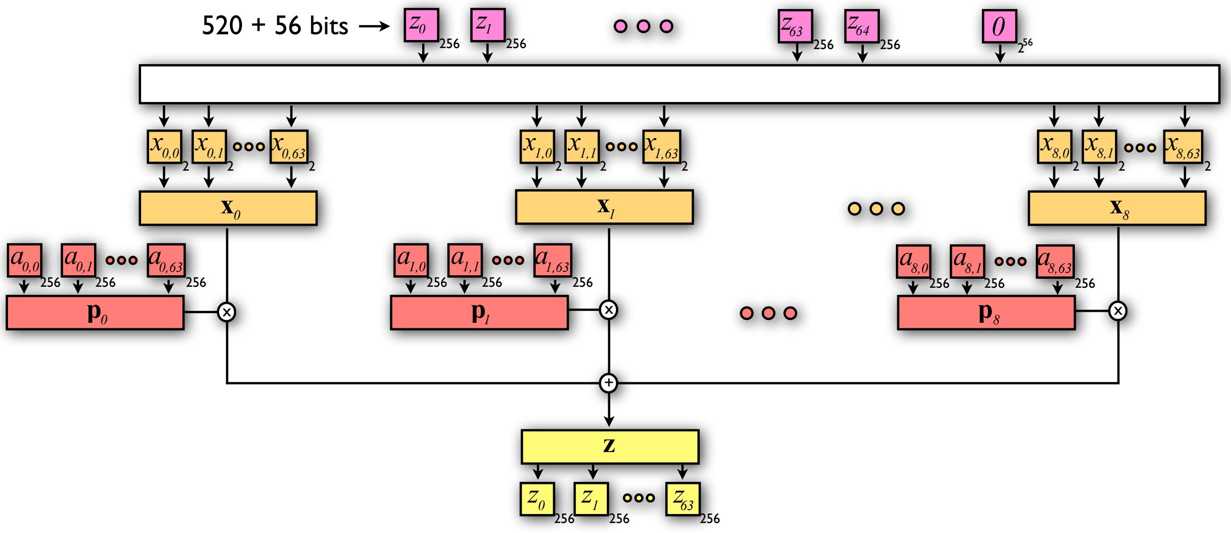

Figure 5: FinalTransform procedure.

3 SIMD implementation

The SWIFFT compression functions are highly parallelizable and admit very efficient implementation on modern microprocessors. In this section, we describe how to exploit this fact in order to achieve substantial speed up on processors that are equipped with SIMD architecture. These ideas naturally extend to fast hardware implementations.

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, and that modern microprocessors have explicit instructions for exactly these kinds of operations. We now proceed in more detail.

For concreteness we set parameters $n = 64$, $m = 32$, and modulus $p = 257$. This corresponds to the inner layer evaluations of SWIFFT within SWIFFTX. The outer layer can be described by taking $m = 25$. 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 $mn = 2048$-bit input, viewed as $m = 32$ binary vectors $\mathbf{x}_0, \ldots, \mathbf{x}_{31} \in \{0,1\}^{64}$. (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

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

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 32 $\mathbf{y}_j$ vectors:

$$z_i = \sum_{j=0}^{31} a_{i,j} \cdot y_{i,j} \pmod{257},$$

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

3.1 Computing F

The most expensive part of the computation is clearly the computation of the transformation F on the 32 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 (1) 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.

3.2 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.

3.3 Summary

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.

3.4 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. 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 Security Analysis

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

4.1 Interpretation of Security Proofs

An asymptotic proof of one-wayness for the basic SWIFFT function was given in [7], and an asymptotic proof of collision-resistance (a stronger property) was given independently in [8] and [3]. 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 [7, 8, 3] for relevant definitions and 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 is able to find a collision in SWIFFT for *uniformly random* choice of randomizers in some feasible amount of time T with some noticeable probability δ. The algorithm might only succeed on a small (but noticeable) fraction of randomizers, 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.

4.2 The underlying assumption

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.

At a minimum, the asymptotic proofs of security indicate that there are no unexpected "structural weaknesses" in the design of SWIFFT, at least in terms of collision-resistance. Specifically, the ability to find collisions efficiently (in an asymptotic sense) would necessarily require new algorithmic insights about finding short vectors in *arbitrary* ideal lattices (over the ring $\mathbb{Z}[\alpha]/(\alpha^n + 1)$).

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 2nth 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.[1] Further connections between the complexity of lattice problems and algebraic number theory were given in [9].

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 way to exploit this algebraic structure in any significant way. 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.

4.3 Cryptanalysis

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

[1]In [7, 8, 3], 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 equivalent in terms of lengths of vectors, and the complexity of finding short vectors is the same under both mappings.

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 the basic SWIFFT function has an output length of about $n \lg p$ bits, it does *not* enjoy a full $n \lg p$ “bits of security” for one-wayness, nor $(n \lg p)/2$ bits of security for collision resistance. Nor does the basic SWIFFT function satisfy additional desirable properties, such as pseudorandomness. The enhanced SWIFFTX function was designed to address these issues.

The SWIFFT function by itself is linear and is therefore susceptible to k-list/generalized birthday attacks [10] as described in [4]. It was shown that one could find collisions and preimages in the SWIFFT function using approximately 2^{106} and 2^{128} operations, respectively. We now describe how similar attacks can be applied to the SWIFFTX function and the reasons why we believe that they are not any more effective than naive brute-force attacks.

The SWIFFTX compression function is linear in the inner and outer layers, but the intermediate layer was designed to break all linearity between the two layers. It is our belief that when the three layers are combined together, the entire function is highly non-linear. Nevertheless, it is possible to mount attacks separately on each layer. For example, collisions in the inner layer of SWIFFTX (which consists of three distinct SWIFFT functions) clearly correspond to collisions in the entire function. The inner layer is a linear function (so generalized birthday attacks can be applied) that maps 2048 bits to 1536 bits. To find a collision in such a function using techniques described in [4] would require approximately 2^{384} operations, which is greater than the approximately 2^{256} operations needed to break the entire function using the standard birthday attack.

A way to find a preimage of any output y of the function involves first finding the preimage of y in the outer layer, then inverting the intermediate layer, and finding the preimage in the inner layer. One can use the birthday attack to find a preimage of *y in the ouside layer* using the generalized birthday techniques. Since the input to the outer layer is 1600 bits, and the output is 520 bits, and the function is linear, methods identical to those described in [4] can be used to obtain preimages in time approximately 2^{100}. The intermediate layer of the function is trivially invertible, so we can obtain a string whose preimage under the inner layer alone would be a true preimage of y. Finding preimages in the inner layer using the techniques described in [4] would require approximately 2^{512} operations, which is equivalent to the time necessary to mount the trivial attack on the SWIFFTX function with 512-bit digests.

5 Performance

In this section we report on the performance of SWIFFTX (both SIMD and non-SIMD versions) on various input sizes. The performance of the SIMD version was about 8 times faster than its non-SIMD counterpart. Also, because of the expensive final transformation stage, the throughput rate for hashing short messages was significantly less than for long ones.

We did not optimize for 8-bit or 64-bit machines, and did not test the speed of our submitted implementation on such machines. We conjecture that by using the best compiler optimizations (as we did on 32bit version) one can potentially gain a two-fold improvement on 64-bit machines. For the case of 8-bit machines, a conservative estimate would be to multiply by 4 the results from 32bit architectures, since in the worst case an operation that requires multiplication on 32bit machine would require 4 such operations on 8bit machine.

5.1 Speed Measurement

To measure the speed of SWIFFTX, we applied two standard methods: for measuring the rate in MB/Secs we measured the running time is seconds. On Windows machines, for example, this was accomplished using the 'clock()' function (from ⟨time.h⟩). For runs that take about 10 seconds, which was the case for our tests, the time spent on non-SWIFFTX code is amortized and can be neglected. We divided the length of the hashed message by the time (multiplying by the number of trials, of course) to get the rate in MB/Secs. To count the number of cycles per byte we used the ubiquitous RDTSC instruction, removing the overhead of the measurement itself and averaging over a number of trials. For both measurements the variance turned out to be low, so the results we present here are indeed representative.

5.2 The test scenarios

In the course of the development process we have carried out a large number of runs (different versions, different machines and OSes, etc). For clarity we present here the results of three basic tests:

Test 1: Short message test. This test hashed an empty message for 200000 times.

Test 2: Long message test. This test hashed a message of 64×10^6 bytes. The message was produced from NIST's KAT code - replicating a short pattern of 64 bytes 10^6 times.

We also ran a SWIFFTX internal tables initialization test. This test just called the SWIFFTX initialization function 2×10^9 number of times. It took 4.22×10^{-9} seconds (which corresponds to 9 cycles) for a single initialization function call.

The numbers above were chosen to produce a running time of about 10 seconds on a typical machine we have, for each test. As said above, we have carried out many tests, some took many hours of runs, but we found this short 10-seconds test to be a representative as well. All the tests were carried out for 512bit digest size, since we have no special treatment of shorter digest size (in terms of speed/memory, and also algorithmically).

5.3 Platforms

We conducted tests on the following four standard platforms:

1. By T60 we denote the Lenovo/ThinkPad T60 with Intel Centrino Duo T7400, each core running @ 2.16GHz with 1GB of RAM running Windows XP Professional build 5.1.2600, 32bit.

2. By MSI we denote MSI laptop with Intel Core Duo T7250, each core running @ 2.00GHz with 2GB of RAM running Windows Vista Home Basic build 6.0.6000, 32bit.

3. By DESKTOP1 we denote a desktop computer with Intel Core Duo E4500, each core running @ 2.20GHz with 2GB of RAM running Windows XP Professional build 5.1.2600, 32bit.

4. By DESKTOP2, we denote a desktop computer with Intel Core Duo E4500, each core running @ 2.20GHz with 2GB of RAM running Linux SUSE tirana 2.6.22.

Note that all of the machine above represent a typical configuration as NIST define the reference platform in section 6.B of the submission requirements document.

5.4 SIMD architecture

Since the main CPU-cycles-consuming part of SWIFFTX is the four applications of SWIFFT, the potential of high speed on modern architectures is strong. And indeed, compiling the code for SSE-2 instruction set produced a much faster result than on the NIST's reference platform using ANSI C. Here we provide only one example, but in the future we plan to explore this potential further, by, for example, implementing SWIFFTX on GPUs. Since almost every modern machine today is equipped with a powerful graphics card, we believe that SIMD timings may in practice be even more important than NIST's constrained ANSI C reference implementation. We ran the SIMD tests on DESKTOP2.

5.5 Table of results

	Test 1	Test 2	Test 2 (SIMD)
MB/sec	2.92	6.22	37
Cycles/Byte	657	320	57

Figure 6: Performance table

5.6 Memory footprint size

The compiled SWIFFTX binary occupies 20480 bytes on the disk. While running KAT-MCT code, for example, the memory consumption is a total of 1416 KBytes, which includes the KAT and MCT code.

Acknowledgments

Thanks to Eli Biham, Ron Rivest, and Eran Tromer for advice and encouragement.

References

[1] E. Biham and O. Dunkelman. A framework for iterative hash functions - HAIFA. Technical report, Technion Computer Science Department Technical Report CS-2007-15, 2007.

[2] 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.

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

[4] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFT: a modest proposal for FFT hashing. In *FSE*, 2008.

[5] Vadim Lyubashevsky. Lattice-based identification schemes secure under active attacks. In *International workshop on practice and theory in public key cryptography - PKC 2008*, volume 4939 of *Lecture Notes in Computer Science*, pages 162–179, Barcelona, Spain, March 2008. Springer.

[6] Vadim Lyubashevsky and Daniele Micciancio. Asymptotically efficient lattice-based digital signatures. In Ran Canetti, editor, *Theory of cryptography conference - Proceedings of TCC 2008*, volume 4948 of *Lecture Notes in Computer Science*, pages 37–54, New York, NY, USA, March 2008. Springer.

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

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

[9] C. Peikert and A. Rosen. Lattices that admit logarithmic worst-case to average-case connection factors. In *STOC*, 2007.

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

A Pseudocode

A.1 SWIFFTX Compression Function

Input: Binary string $\mathbf{x}$ of length 2048.
Output: Binary string $\mathbf{z}$ of length 520.

```
1: Read the input $\mathbf{x}$ as $\mathbf{x}_0\mathbf{x}_1 \dots \mathbf{x}_{31}$, where $\mathbf{x}_i$ are bit-strings of length 64.
2: for $j = 0$ to 2 do
3:    $[z_{0,j}, z_{1,j}, \dots, z_{64,j}] \leftarrow \text{ConvertToBytes}(\text{SWIFFT}(32, \mathbf{x}_0, \dots, \mathbf{x}_{31}, \mathbf{A}_j))$
4:    for $i = 0$ to 64 do
5:       $z_{i,j} \leftarrow \text{SBox}[z_{i,j}]$
6:    end for
7: end for
8: $\mathbf{r} \leftarrow z_{0,0}||z_{0,1}|| \dots ||z_{0,63}||z_{1,0}||z_{1,1}|| \dots ||z_{1,63}||z_{2,0}||z_{2,1}|| \dots ||z_{2,63}||z_{0,64}||z_{1,64}||z_{2,64}||(\text{SBox}[0])^5$
9: Treat $\mathbf{r}$ as a bit-string (i.e. convert every byte $z_{i,j}$ and the five bytes $(\text{SBox}[0])^5$ to their binary
   representation)
10: Read the 1600 bit-string $\mathbf{r}$ as $\mathbf{x}_0\mathbf{x}_1 \dots \mathbf{x}_{24}$ where, $\mathbf{x}_i$ are bit-strings of length 64.
11: $[z_0, z_1, \dots, z_{64}] \leftarrow \text{ConvertToBytes}(\text{SWIFFT}(25, \mathbf{x}_0, \dots, \mathbf{x}_{24}, \mathbf{A}_0))$
12: $\mathbf{z} \leftarrow z_0||z_1|| \dots ||z_{64}$
13: Treat $\mathbf{z}$ as a bit-string (i.e. convert every byte $z_i$ to its binary representation)
14: ouptut $\mathbf{z}$
```

A.2 SWIFFT

Input: Integer $k > 0$, k strings $\mathbf{x}_0, \dots, \mathbf{x}_{k-1} \in \{0,1\}^6 4$, and $k' \times 64$ matrix $\mathbf{A}$ where $k' \geq k$.
Output: Numbers $z_0, \dots, z_{63}$, where each $z_i \in \mathbb{Z}_{257}$.

```
1: for $i = 0$ to $k - 1$ do
2:    $\mathbf{x}_i \leftarrow \text{IndexBitReversal}(\mathbf{x}_i)$
3: end for
4: Interpret each $\mathbf{x}_i$ as a polynomial of degree at most 63. For example, the string 1100...001
   corresponds to $1 + \alpha + \alpha^{63}$.
5: $w \leftarrow 42$
6: Initialize $z_0, z_1, \dots z_{63}$ to 0.
7: for $i = 0$ to $k - 1$ do
8:    for $j = 0$ to 63 do
9:       $z_j \leftarrow z_j + \mathbf{A}[i][j] \cdot \mathbf{x}_i(w^{2j+1}) \bmod 257$
10:   end for
11: end for
12: output $z_0, \dots, z_{63}$
```

A.3 Generating the randomizer matrices (GenerateA's)

Input: Vector $\mathbf{p}$ containing the decimal part of π (i.e. $\mathbf{p}[0] = 1, \mathbf{p}[1] = 4, \mathbf{p}[2] = 1, \mathbf{p}[3] = 5, \dots$)
Output: Matrices $\mathbf{A}_0, \mathbf{A}_1, \mathbf{A}_2 \in \mathbb{Z}_{257}^{32\times 64}$.

```
1: Initialize array $\mathbf{a}$ to have 6144 entries ($6144 = 32 * 64 * 3$)
2: $ca \leftarrow 0$
```

```
3: $cp \leftarrow 0$
4: **while** $ca < 6144$ **do**
5:    $n \leftarrow \mathbf{p}[cp] * 100 + \mathbf{p}[cp+1] * 10 + \mathbf{p}[cp+2]$
6:    **if** $n < 257 * 3$ **then**
7:       $\mathbf{a}[ca] \leftarrow n \bmod 257$
8:       $ca \leftarrow ca + 1$
9:    **end if**
10:   $cp \leftarrow cp + 3$
11: **end while**
12: **for** $k = 0$ to $2$ **do**
13:    **for** $i = 0$ to $31$ **do**
14:       **for** $j = 0$ to $63$ **do**
15:          $\mathbf{A}_k[i][j] \leftarrow \mathbf{a}[2048k + 64i + j]$
16:       **end for**
17:    **end for**
18: **end for**
19: output matrices $\mathbf{A}_0, \mathbf{A}_1, \mathbf{A}_2$
```

A.4 The Index-Bit Reversal function (IndexBitReversal)

Input: Binary string $\mathbf{x}$ of length 64.
Output: Binary string $\mathbf{x}'$ of length 64.

```
1: Write $\mathbf{x}$ as $x_0x_1 \dots x_{63}$ where $x_i$ are bits.
2: **for** $i = 0$ to $63$ **do**
3:    Write $i$ as a 6-bit binary number $i_0i_1i_2i_3i_4i_5$
4:    $j \leftarrow i_5i_4i_3i_2i_1i_0$
5:    $x'_i \leftarrow x_j$
6: **end for**
7: $\mathbf{x}' \leftarrow x'_0x'_1 \dots x'_{63}$
8: output $\mathbf{x}'$
```

A.5 The Final Transformation (FinalTransform)

Input: Binary string $\mathbf{x}$ of length 520
Output: Binary string $\mathbf{z}$ of length 512

```
1: $\mathbf{x} \leftarrow \mathbf{x}||0^{56}$
2: **for** $i = 0$ to $8$ **do**
3:    Set $\mathbf{p}_i$ to the polynomial $\mathbf{A}_1[i][0] + \mathbf{A}_1[i][1]\alpha + \mathbf{A}_1[i][2]\alpha^2 + \dots + \mathbf{A}_1[i][63]\alpha^{63}$.
4: **end for**
5: Break $\mathbf{x}$ into 9 bit-strings $\mathbf{x}_0, \dots, \mathbf{x}_8$ of length 64 and treat each $\mathbf{x}_i$ and $\mathbf{p}_i$ as polynomials of
   degree at most 63
6: Compute $\mathbf{z} \leftarrow \sum_{i=0}^{8} \mathbf{x}_i \cdot \mathbf{p}_i$ where polynomial multiplication is over the ring $\mathbb{Z}_{256}[\alpha]/(\alpha^{64} + 1)$
7: Treat $\mathbf{z} = z_0 + z_1\alpha + z_2\alpha^2 + \dots + z_{63}\alpha^{63}$ as a string of bytes $z_0||z_1||z_2||\dots||z_{63}$
8: Output $\mathbf{z}$
```

A.6 Convert To Bytes function (ConvertToBytes)

Input: $z_0, \ldots, z_{63}$, where $z_i \in \mathbb{Z}_{257}$
Output: $z'_0, \ldots, z'_{64}$ where $z'_i \in \mathbb{Z}_{256}$

```
1: Find the unique elements $z'_0, \ldots, z'_{63}, b_0, \ldots, b_7$ where $z'_i \in \mathbb{Z}_{256}$ and $b_i \in \{0,1\}$ such that
   $$\text{for all } 0 \le k \le 7, \sum_{i=0}^{7} z_{8k+i} \cdot 257^i = \sum_{i=0}^{7} z'_{8k+i} \cdot 256^i + b_k \cdot 256^8$$
2: $z'_{64} \leftarrow \sum_{i=0}^{7} b_0 \cdot 2^i$
3: output $z'_0, \ldots, z'_{64}$
```

B The S-Box

0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
7d	d1	70	0b	fa	39	18	c3	f3	bb	a7	d4	84	25	3b	3c	0
2c	15	69	9a	f9	27	fb	02	52	ba	a8	4b	20	b5	8b	3a	1
88	8e	26	cb	71	5e	af	ad	0c	ac	a1	93	c6	78	ce	fc	2
2a	76	17	1f	62	c2	2e	99	11	37	65	40	fd	a0	03	c1	3
ca	48	e2	9b	81	e4	1c	01	ec	68	7a	5a	50	f8	0e	a3	4
e8	61	2b	a2	eb	cf	8c	3d	b4	95	13	08	46	ab	91	7b	5
ea	55	67	9d	dd	29	6a	8f	9f	22	4e	f2	57	d2	a9	bd	6
38	16	5f	4c	f7	9e	1b	2f	30	c7	41	24	5c	bf	05	f6	7
0a	31	a5	45	21	33	6b	6d	6c	86	e1	a4	e6	92	9c	df	8
e7	be	28	e3	fe	06	4d	98	80	04	96	36	3e	14	4a	34	9
d3	d5	db	44	cd	f5	54	dc	89	09	90	42	87	ff	7e	56	a
5d	59	d7	23	75	19	97	73	83	64	53	a6	1e	d8	b0	49	b
3f	ef	bc	7f	43	f0	c9	72	0f	63	79	2d	c0	da	66	c8	c
32	de	47	07	b8	e9	1d	c4	85	74	82	cc	60	51	77	0d	d
aa	35	ed	58	7c	5b	b9	94	6e	8d	b1	c5	b7	ee	b6	ae	e
10	e0	d6	d9	e5	4f	f1	12	00	d0	f4	1a	6f	8a	b3	b2	f

Figure 7: The S-Box

C The IV

1f, d7, 60, 96, f1, f5, f7, 5d, bb, 3e, 73, d4, 4c, 76, 61, 23, 52, 3b, 7e, b2, 0d, a6, ab, ab, d2, 87, 03, 3b, 9d, 54, 75, 2b, 3c, 4e, 27, 04, 53, 76, e2, 84, 48, 73, ea, fb, e9, f1, c3, fb, 13, 0b, 3d, bb, be, 9a, 59, 95, a7, fd, f4, f9, cc, 9c, 52, 08, a8