Algorithm Name:

Spectral Hash

Principal Submitter:

Çetin Kaya Koç

koc@cs.ucsb.edu
Tel: (805)-893-8565
Fax: (805)-893-8553

College of Creative Studies
Building 494
University of California, Santa Barbara
Santa Barbara, CA 93106

Algorithm Inventors/Developers:

G. Saldamlı, C. Demirkıran, M. Maguire, C. Minden, J. Topper, A. Troesch, C. Walker and Ç. K. Koç

Algorithm Owner:

Çetin Kaya Koç

SPECTRAL HASH: SHA-3 CANDIDATE

G. SALDAMLI, C. DEMIRKIRAN, M. MAGUIRE, C. MINDEN, J. TOPPER,
A. TROESCH, C. WALKER AND Ç. K. KOÇ

COLLEGE OF CREATIVE STUDIES,
UNIVERSITY OF CALIFORNIA SANTA BARBARA
SANTA BARBARA, CA 93106

Abstract. We describe a new family of hash functions using the discrete Fourier transform and a nonlinear transformation constructed via data dependent permutations. DFT is a well-known cryptographic primitive perfect for generating diffusion and confusion. Due to the usage of the DFT with a nonlinear transformation, the proposed hash generation method is immune to known attacks. Since spectral methods yield efficient and highly parallel architectures, s-hash is highly suitable for hardware realizations.

Contents

1. Introduction

In this document we describe the spectral hashing algorithm. We start with the mathematical background and corresponding arithmetic followed by a design rationale and a detailed description. In Section 5 we turn our attention to the security considerations and discuss spectral hash's immunity to known attacks. We conclude with a discussion of the efficiency of s-hash.

2. Mathematical Background

We now present the following material as mathematical background necessary to describe and implement the spectral hashing algorithm, referred to hereafter as the s-hash algorithm. The necessary operations consist mainly of finite field arithmetic in the fields $GF(2^4)$ and $GF(17)$. Also it is necessary to compute discrete Fourier transforms.

2.1. The Field $GF(2^4)$.

The field $GF(2^4)$ is a binary extension field. Its elements are polynomials of the form $a_3t^3 + a_2t^2 + a_1t^1 + a_0t^0$, where the a_i are binary coefficients (i.e. 0 or 1). This means that any element of $GF(2^4)$ is expressible in binary as a string a, of the form $a = a_3 \,||\, a_2 \,||\, a_1 \,||\, a_0$, where $x \,||\, y$ notates the concatenation of the binary strings x and y. There are three operations that s-hash employs in $GF(2^4)$: addition, multiplication, and inversion.

2.1.1. *Addition.*

The addition in field $GF(2^4)$ is performed modulo 2. Since the addition of elements in $GF(2^4)$ is not the same as conventional addition, we will notate it as $\oplus$ to distinguish it from traditional addition.

Example 1. *Let $x = t^3 + t + 1$ and $y = t^2 + 1$ be elements of $GF(2^4)$. Then,*

$$\begin{aligned} x \oplus y &= (t^3 + t + 1) + (t^2 + 1) \\ &= t^3 + t^2 + t. \end{aligned}$$

In binary notation, one gets $(1011)_2 + (0101)_2 = (1110)_2$. It is apparent that $x \oplus y$ may be computed by simply computing x XOR y at the bit level.

2.1.2. *Multiplication.*

Multiplication in $GF(2^4)$ involves performing polynomial multiplication modulo an irreducible binary polynomial of degree four. There are several such acceptable irreducible polynomials, but s-hash employs

$$f(t) = t^4 + t^3 + t^2 + t + 1.$$

Since multiplication in $GF(2^4)$ is not traditional multiplication, we denote it by $*$.

Example 2. *As before, let $x = t^3 + t + 1$ and $y = t^2 + 1$. Then,*

$$\begin{aligned} x * y &= (t^3 + t + 1) \cdot (t^2 + 1) \\ &= t^5 + t^3 + t^2 + t^3 + t + 1 \\ &= t^5 + t^2 + t + 1 \\ &= t^2 + t \bmod f(t). \end{aligned}$$

Unfortunately, there is no simple operation at the bit level as there is for addition. However, there is more than one valid way to implement multiplication. It is clear that when two elements of $GF(2^4)$ are multiplied together, the resulting unreduced polynomial will never have a degree greater than six. Therefore one could simply use the fact that

$$\begin{aligned} t^4 &\equiv t^3 + t^2 + t + 1 \bmod f(t) \\ t^5 &\equiv 1 \bmod f(t) \\ t^6 &\equiv t \bmod f(t). \end{aligned}$$

One can then substitute these values into any unreduced polynomial to obtain the equivalent reduced polynomial. An example of this technique is as follows.

Example 3. *Let* $x = t^3 + t^2 + 1$ *and* $y = t^2 + 1$. *Then,*

$$\begin{aligned} x * y &= (t^3 + t^2 + 1) \cdot (t^2 + 1) \\ &= t^5 + t^4 + t^2 + t^3 + t^2 + 1 \\ &= t^5 + t^4 + t^3 + 1 \\ &= 1 + t^3 + t^2 + t + 1 + t^3 + 1 \\ &= t^2 + t + 1. \end{aligned}$$

Another option is to simply use look-up tables, since the sizes of the tables are quite small.

2.1.3. *Inversion.* $GF(2^4)$ has an identity element, namely 1. All the elements of $GF(2^4)$ have an inverse, except for 0, which we define to be 0. If x is an element of $GF(2^4)$, its inverse is the element x^{-1} such that $x * x^{-1} = 1$. One can calculate inverses directly since $x^{-1} = x^{15}$ in $GF(2^4)$, but we can also employ look-up tables, which is much more efficient.

2.2. **The Field $GF(17)$.** The field $GF(17)$ is the structure where all the discrete Fourier transform computations takes place. $GF(17)$ employs standard arithmetic modulo 17. We remark that the field $GF(17)$ is also small enough to employ look-up tables for its arithmetic. In fact, two or more operations can be performed in parallel by merging the look-up tables.

2.3. **Discrete Fourier Transform.** As part of the s-hash algorithm, we employ discrete Fourier transforms, hereafter referred to as DFTs. All our DFTs are performed in $GF(17)$. We apply two types of DFTs, 4-point DFTs and 8-point DFTs. Below is the equation for the DFT.

$$\mathbf{X}_i = DFT_d(\mathbf{x}) := \sum_{j=0}^{d-1} \mathbf{x}_j \omega_d^{ij} \bmod 17,$$

where $i = 0, 1, 2, \ldots d - 1$, and d is either 4 or 8. Here ω is the d-th root of unity in $GF(17)$, and $\mathbf{x}$ is an input string of d numbers. For the 4-point DFTs ($d = 4$), $\omega_4 = 4$, and for the 8-point DFTs ($d = 8$), $\omega_8 = 2$.

A fast Fourier transform (FFT) is an efficient algorithm to compute the DFT and its inverse. In practice, s-hash employs FFTs for their greater computational efficiency. Specifically small-radix FFTs are suitable for use in s-hash.

Table 1. Mathematics Reference

$GF(2^4)$	A field of 16 binary polynomial elements.
$GF(17)$	A field of 17 elements, with standard arithmetic modulo 17.
$f(t)$	$t^4 + t^3 + t^2 + t + 1$. An irreducible polynomial in $GF(2^4)$.
$\oplus$	Addition in $GF(2^4)$ equivalent to a bitwise XOR.
$*$	Polynomial multiplication modulo $f(t)$ in $GF(2^4)$. Computable with the following relations, $t^6 \equiv t \bmod f(t)$, $t^5 \equiv 1 \bmod f(t)$, $t^4 \equiv t^3 + t^2 + t + 1 \bmod f(t)$.
ω_d	The d-th root of unity in $GF(17)$. Recall $\omega_4 = 4$ and $\omega_8 = 2$.
$DFT_d(\mathbf{x})$	discrete Fourier transform of the list of points $\mathbf{x}$ of length d.

3. Design Rationale

When designing the spectral hashing algorithm, our main concern was to create an algorithm that was resistant to all known cryptographic attacks. Therefore the main components of the s-hash algorithm were chosen to support this goal. These components are as follows:

- The discrete Fourier transform, which has been shown to be a perfect cryptographic primitive for generating diffusion and confusion by Schnorr [1]. Here we employ a multi-dimensional DFT in order to enlarge the DFT length while enjoying the smaller field arithmetic that can be carried out with tiny look-up tables.
- The inverse map, which has the best known nonlinearity [2] in the field $GF(2^4)$. The inverse map is used in several places to build up a strong infrastructure against differential and linear attacks.
- A nonlinear system of equations, which is constructed by selecting system variables using a permutation table generated by data dependent permutations. The nonlinear transformation is the main protection against pre-image attacks; without knowing the state of the data dependent permutation table one has to trace all possible permutations to find the pre-images.
- In order to avoid problems with the Merkle-Damgard construction, we employ the large-pipe strategy described in [3]. The final hash value, if less than 512-bits, is generated from an array punctured by the data dependent permutations.
- Every single operation in s-hash, either linear or nonlinear, is a one-to-one operator, with the exception of the puncturing performed during the final hash generation. We state that this makes s-hash strong against collision attacks.

Aside from these security oriented principles, efficiency, mainly due to parallelism, was our major concern. For this reason, the multi-dimensional DFT structure was chosen as the foundation of s-hash. If FFT methods are used for multi-dimensional DFT calculations, one only needs a logarithmic number of steps to calculate the DFTs. To be more specific, the 3-dimensional DFTs of s-hash may be calculated in 7 steps if a fully parallel hardware FFT is employed.

In order to complement the FFT calculations, the manipulations (swaps) on the data dependent permutation table were also designed to be completed in 7 steps. Therefore, a fully parallel hardware implementation of s-hash's compression function would only require 7 steps of calculations. In fact, with this novel design, s-hash becomes extremely suitable for hardware realizations.

4. Spectral Hashing Algorithm

Before going into the details of the spectral hashing algorithm, we note that s-hash adapts the classical Merkle-Damgard scheme for hash generation as originally illustrated by the authors in Figure 1. We have of course merely used this as a template for the s-hash algorithm; a more complete idea of the process is depicted in Figure 2. Each component will be discussed in turn, beginning with a summary of the data structures used throughout.

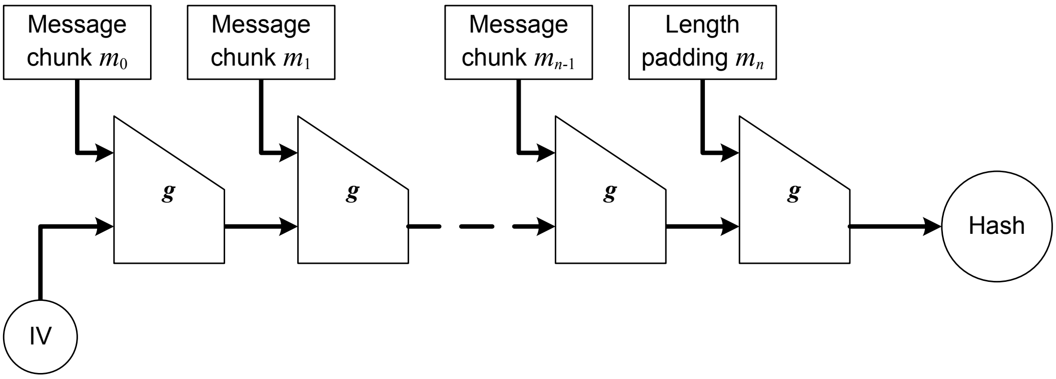


Figure 1. Classical Merkle-Damgard hash scheme.

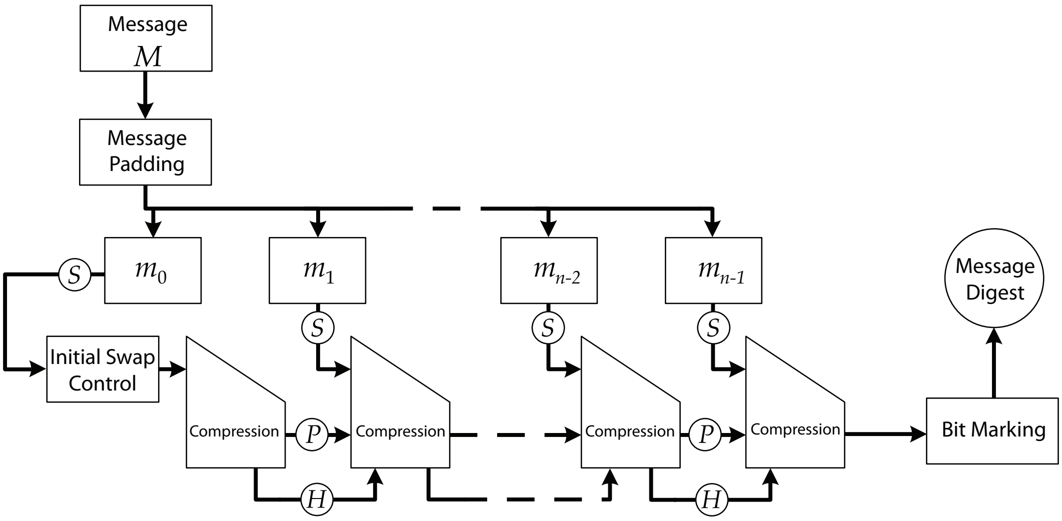


Figure 2. The augmented Merkle-Damgard scheme used in s-hash.

4.1. **Data Structures and Organization.** We now present the data structures and overall organization necessary to understand the s-hash algorithm. We begin with the initial message M, which consists of an arbitrary binary string of length l, $l \leqslant 2^{64}$. Since s-hash operates on 512-bit blocks, the initialization includes a padding scheme which results in an extended message whose length is a multiple of 512. The padded message string $M' = m_0 \,||\, m_1 \,||\, \cdots \,||\, m_{n-1}$ consists of n chunks where each m_i $(i = 0, 1, 2, \ldots, n-1)$ has a 512-bit length.

4.1.1. *State Prism.* For each m_i we construct a $4 \times 4 \times 8$ array, or state prism, in the following fashion. This process need not be done until the chunk m_i is ready to be compressed. We break the chunk into 128 4-bit words. Formally, let $m_i = s_0 \,||\, s_1 \,||\, \cdots \,||\, s_{127}$, where each s_N is a 4-bit binary number for all N in the index set $\{0, 1, \ldots, 127\}$. We then re-index the binary string m_i as follows:

$$S_{(i,j,k)} = s_N,$$

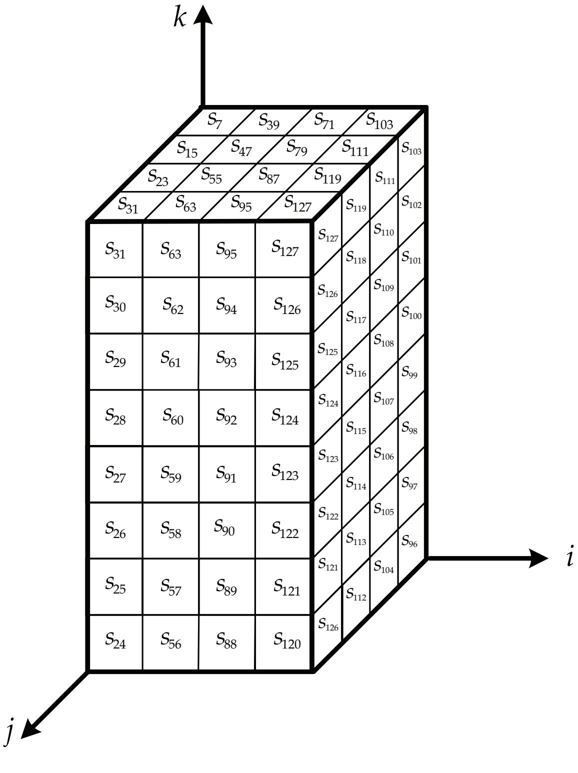
where

$$N = 32i + 8j + k,$$

for $i, j = 0, 1, 2, 3$ and $k = 0, 1, \ldots, 7$. For ease of reversal we also note that

$$\begin{aligned} i &= \left\lfloor \frac{N}{32} \right\rfloor \bmod 4, \\ j &= \left\lfloor \frac{N}{8} \right\rfloor \bmod 4, \\ k &= N \bmod 8. \end{aligned}$$

In fact, this re-indexing corresponds to a filling of a $4 \times 4 \times 8$ prism as shown in Figure 3. We refer to this array as the state prism, or s-prism. In every iteration of the Merkle-Damgard scheme this state prism is filled with a message chunk and then processed. We note that even though the data held in the s-prism consists of only 4-bit words initially, throughout the compression function a fifth bit of data may be needed for any particular word.



k
s_7 s_{39} s_{71} s_{103}
s_{15} s_{47} s_{79} s_{111}
s_{23} s_{55} s_{87} s_{119}
s_{31} s_{63} s_{95} s_{127}
s_{31} s_{63} s_{95} s_{127}
s_{30} s_{62} s_{94} s_{126}
s_{29} s_{61} s_{93} s_{125}
s_{28} s_{60} s_{92} s_{124}
s_{27} s_{59} s_{91} s_{123}
s_{26} s_{58} s_{90} s_{122}
s_{25} s_{57} s_{89} s_{121}
s_{24} s_{56} s_{88} s_{120}
s_{127} s_{126} s_{125} s_{124} s_{123} s_{122} s_{121} s_{126}
s_{119} s_{118} s_{117} s_{116} s_{115} s_{114} s_{113} s_{112}
s_{111} s_{110} s_{109} s_{108} s_{107} s_{106} s_{105} s_{104}
s_{103} s_{102} s_{101} s_{100} s_{99} s_{98} s_{97} s_{96}
i
j

Figure 3. State Prism in the initial state.

4.1.2. *Permutation Prism.* Similar to s-prism, we employ another $4 \times 4 \times 8$ array holding a permutation table that is used for setting up the nonlinear system of equations used in the compression function. This permutation array is referred to as the permutation prism, or p-prism, which is initially configured as

$$P_{(i,j,k)} = N,$$

for $N = 0, 1, \ldots, 127$, $i, j = 0, 1, 2, 3$ and $k = 0, 1, \ldots, 7$ (see Figure 4). Clearly the p-prism holds the permutations of the index set, where each index is a 7-bit string.

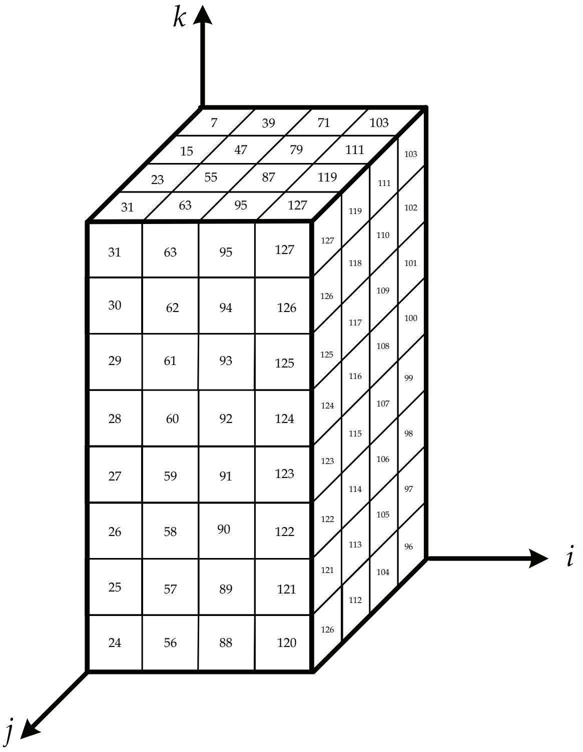


Figure 4. Permutation Prism in the initial state.

4.1.3. *"H" Prism.* Finally, in addition to the above prisms, there is another $4 \times 4 \times 8$ array called the h-prism. This prism holds the data from the s-prism of the previous iteration of the compression function. It is initially configured as

$$H_{(i,j,k)} = 0,$$

for all $i, j = 0, 1, 2$ and all $k = 0, 1, \ldots, 7$. The h-prism holds data from only the final s-prism of a round, so each cell of an h-prism consists solely of 4 bits.

4.2. **Initialization.** The following steps summarize the initialization process, beginning with the message padding:

4.2.1. *Message Padding.* If the length of the message M is not a multiple of 512 bits then it is necessary to append additional bits to the end of the message. This occurs in the following manner:

i. Append the bit '1' to the message.
ii. Append k bits of '0', where k is a non-negative integer such that $l + k + 1 \equiv 448 \bmod 512$.
iii. Append the 64-bit big-endian binary representation of l, returning the padded message string m.

Note that if the length of a message is 64 bits fewer than a multiple of 512, the padding process creates an additional message chunk consisting of '0's along with the 64-bit big-endian binary representation of l.

4.2.2. *Prism Initialization and Initial Swap.* We start by processing chunk m_0. First we break the chunk into 128 4-bit "words" and then map the words into the s-prism as described above. The p-prism and h-prism are also filled with the appropriate values in the aforementioned manner. Then during the initialization, after filing the s-prism with the words of m_0, we modify the p-prism by applying some simple entry swaps. Since the s-prism initially holds 4-bit entries, we apply the following swaps to the p-prism:

$$\text{swap}(P_{(i,j,k)}, P_{(Sh_{(i,j,k)}, Sl_{(i,j,k)}, k)}),$$

where

$$Sl_{(i,j,k)} = S_{(i,j,k)} \bmod 4$$

and

$$Sh_{(i,j,k)} = S_{(i,j,k)} \operatorname{div} 4$$

for all $i, j = 0, 1, 2, 3$ and $k = 0, 1, \ldots, 7$. In other words, $Sl_{(i,j,k)}$ and $Sh_{(i,j,k)}$ represent the two least and most significant bits of the s-prism entry $S_{(i,j,k)}$ respectively. It should be noted that the p-prism is filled, and these initial swaps occur, only during the processing of the chunk m_0 and never again in the s-hash algorithm.

After this setup, the s-prism and p-prism are passed into the compression function for the first time, as we now describe.

4.3. **Compression Function.** The compression function of s-hash takes in a message chunk after it has been placed into the s-prism, and the p-prism from the previous iteration. The compression function also uses h-prism, which contains the s-prism data from the previous iteration. The compression function consists of the following stages:

- Affine Transformation
- Discrete Fourier Transformations
- Nonlinear Transformation

We note that the naming convention of these stages mainly describes the actions on the s-prism; the transformations on the p-prism are computed using the state of the s-prism.

Before going into detail, we sketch how the compression function works. In Figure 5, the left-hand column illustrates the consecutive transformations applied to s-prism. The right-hand column illustrates the modifications on p-prism, which are controlled by the intermediate entries of the s-prism derived from the outputs of each stage. In total p-prism goes through 5 different types of swaps and rotations as shown in the figure.

We now describe in detail each main stage of the compression function.

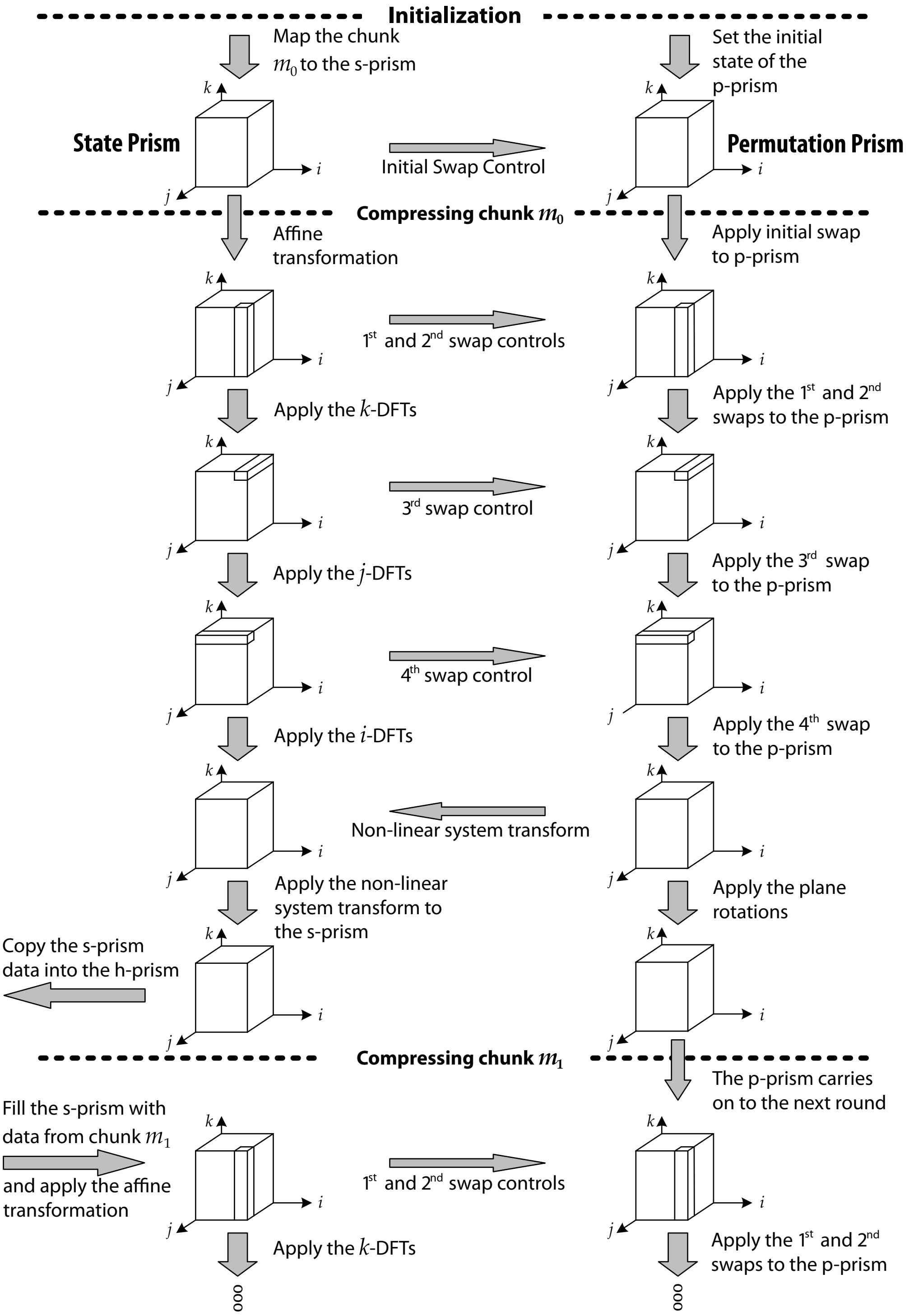

Initialization
Map the chunk m_0 to the s-prism
Set the initial state of the p-prism
State Prism
Permutation Prism
Initial Swap Control
k
i
j
Compressing chunk m_0
Affine transformation
Apply initial swap to p-prism
1st and 2nd swap controls
Apply the k-DFTs
Apply the 1st and 2nd swaps to the p-prism
3rd swap control
Apply the j-DFTs
Apply the 3rd swap to the p-prism
4th swap control
Apply the i-DFTs
Apply the 4th swap to the p-prism
Non-linear system transform
Apply the non-linear system transform to the s-prism
Apply the plane rotations
Copy the s-prism data into the h-prism
Compressing chunk m_1
The p-prism carries on to the next round
Fill the s-prism with data from chunk m_1 and apply the affine transformation
1st and 2nd swap controls
Apply the k-DFTs
Apply the 1st and 2nd swaps to the p-prism
ooo
ooo

Figure 5. Overview of the compression function used in the spectral hashing algorithm.

4.3.1. *Afffine Transform.* The following affine transform is applied to each entry of the s-prism:

$$S_{(i,j,k)} := \alpha(S_{(i,j,k)})^{-1} \oplus \gamma,$$

where

$$\alpha = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} \text{ and } \gamma = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

One can clearly see that the foremost component of the affine transform (i.e $\alpha(S_{(i,j,k)})^{-1}$) is the inverse map, which has the best known nonlinearity [2] in the field $GF(2^4)$. In support of this, AES substitution boxes use the inverse map in $GF(2^8)$ as its main nonlinear component to protect against differential and linear attacks. Therefore, the inverse map in $GF(2^4)$ is taken as a strong function to build immunity from known successful attacks. Additionally, in order to eliminate the weaknesses related to fixed points, the inverse map is coupled with a linear transformation.

4.3.2. *Discrete Fourier Transforms.* After the affine transforms, we simply apply the 3-dimensional DFT to the s-prism. Since some intermediate DFT calculations are needed for p-prism modifications, s-hash may not allow for the implementation of all fast DFT methods. In particular, p-prism swaps use the intermediate values of the standard row-column method of DFTs successively applied through k, j and i axes as seen in Figure 6.

As described in the previous sections, the DFT used is defined over the prime field $GF(17)$, permitting transforms of length 8 and 4 for the principle roots of unity $\omega_8 = 2$ and $\omega_4 = 4$ respectively. Moreover, observe that in the first iteration of the row-column method (i.e. DFT through the k-axis) one has to compute 16 different 1-dimensional 8-point DFTs. However, through the i and j axes, we need to calculate 32 different 4-point DFTs for each axis.

Unlike the DFT revisions of s-prism, p-prism modifications are a little bit more complicated. We perform data (s-prism) dependent swaps on the p-prism similar to those done during the initialization stage.

The first modifications after the initial swaps on the p-prism are performed using swap control planes (sc-planes) of the s-prism. The 1st sc-plane is a 4×4 array created by XORing the two planes (matrices) with k co-ordinates such that $k \equiv 0 \bmod 4$ (see Figure 7). This sc-plane is generated before the 1-dimensional DFT through the k-axis. If explicitly written, the following gives the swapping for all $i, j = 0, 1, 2, 3$:

$$\begin{array}{lll} \text{if } ((S_{(i,j,0)}[0] \oplus S_{(i,j,4)}[0]) = 0) & \text{then,} & \text{swap}(P_{(i,j,0)}, P_{(i,j,7)}), \\ \text{if } ((S_{(i,j,0)}[1] \oplus S_{(i,j,4)}[1]) = 0) & \text{then,} & \text{swap}(P_{(i,j,1)}, P_{(i,j,6)}), \\ \text{if } ((S_{(i,j,0)}[2] \oplus S_{(i,j,4)}[2]) = 0) & \text{then,} & \text{swap}(P_{(i,j,2)}, P_{(i,j,5)}), \\ \text{if } ((S_{(i,j,0)}[3] \oplus S_{(i,j,4)}[3]) = 0) & \text{then,} & \text{swap}(P_{(i,j,3)}, P_{(i,j,4)}). \end{array}$$

Note that $S_{(i,j,k)}[n]$ is a bit from the word located at $S_{(i,j,k)}$, and 3 and 0 are the most and least significant bits respectively.

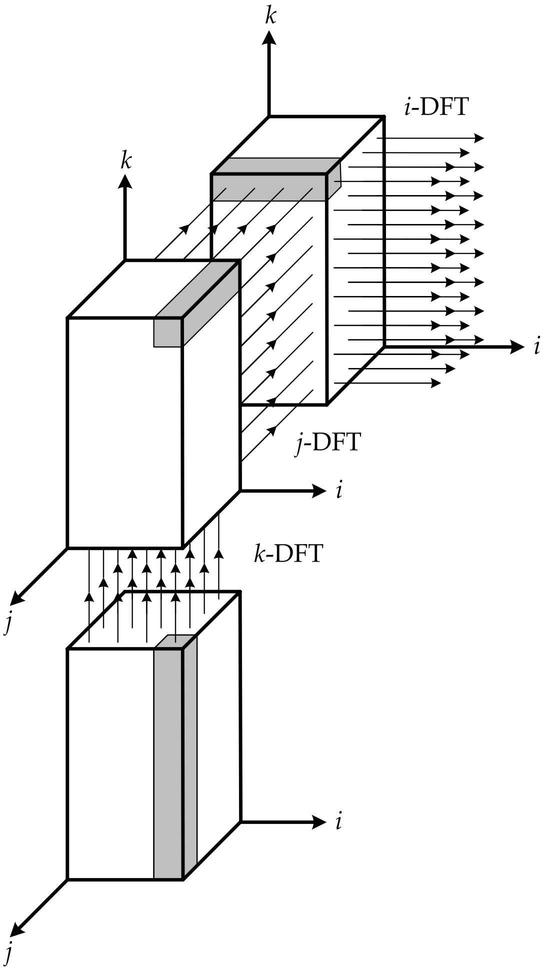


Figure 6. Three dimensional DFTs can be realized by the row-column method: applying 1-dimensional DFTs to the s-prism through k, j and i axes successively.

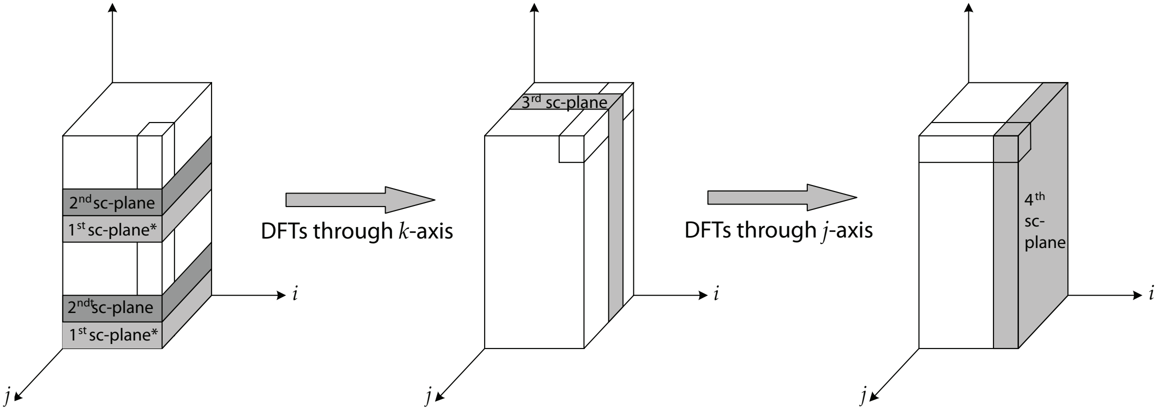


Figure 7. Swap control planes (sc-planes) on the s-prism. (*) The 1st sc-plane is computed by XORing these two.

On the other hand, the 2nd sc-plane consists of two 4×4 arrays controlling the upper and lower halves of the p-prism. These are the planes in the s-prism with k co-ordinates such that $k \equiv 1 \bmod 4$, as seen in Figure 7. The lower sc-plane ($k = 1$) controls the permutations of the lower half of the p-prism as follows:

$$\begin{aligned}
&\text{if } (S_{(i,j,1)}[0] = 0) && \text{then,} && \text{swap}(P_{(i,j,0)}, P_{(i,j,1)}),\\
&\text{if } (S_{(i,j,1)}[1] = 0) && \text{then,} && \text{swap}(P_{(i,j,2)}, P_{(i,j,3)}),\\
&\text{if } (S_{(i,j,1)}[2] = 0) && \text{then,} && \text{swap}(P_{(i,j,1)}, P_{(i,j,2)}),\\
&\text{if } (S_{(i,j,1)}[3] = 0) && \text{then,} && \text{swap}(P_{(i,j,0)}, P_{(i,j,3)}).
\end{aligned}$$

The upper sc-plane ($k = 4$) controls the permutations of the upper half of the p-prism as follows:

$$\begin{aligned}
&\text{if } (S_{(i,j,5)}[0] = 0) && \text{then,} && \text{swap}(P_{(i,j,4)}, P_{(i,j,5)}),\\
&\text{if } (S_{(i,j,5)}[1] = 0) && \text{then,} && \text{swap}(P_{(i,j,6)}, P_{(i,j,7)}),\\
&\text{if } (S_{(i,j,5)}[2] = 0) && \text{then,} && \text{swap}(P_{(i,j,5)}, P_{(i,j,6)}),\\
&\text{if } (S_{(i,j,5)}[3] = 0) && \text{then,} && \text{swap}(P_{(i,j,4)}, P_{(i,j,7)}).
\end{aligned}$$

Notice that the 1st and 2nd sc-planes control the swapping of 16 vectors on the p-prism. Each of these vectors has 8 entries and is parallel to the k-axis.

The 3rd and 4th sc-planes are obtained from the s-prism after the 1-dimensional DFT calculations through the k and j axes respectively. In fact, the 3rd sc-plane is chosen as $S_{(i,2,k)}$ where the 4th sc-plane is $S_{(i,j,3)}$.

Although the input values for the DFT belong to the binary field $GF(2^4)$ (i.e. values represented by at least 4-bits), the DFT operates over the prime field $GF(17)$ and may return 5 bit entries. We select the least significant 4-bits of the sc-plane entries as the swap control bits. Moreover, in order to balance the distribution of the swaps, we involve the index in the calculations. To be more concrete, the following swaps are applied to the p-prism for all $i = 0, 1, 2, 3$ and $k = 0, 1, \ldots, 7$:

$$\begin{aligned}
&\text{if } ((S_{(i,2,k)}[0] \oplus k[0]) = 0) && \text{then,} && \text{swap } (P_{(i,0,k)}, P_{(i,1,k)}),\\
&\text{if } ((S_{(i,2,k)}[1] \oplus k[1]) = 0) && \text{then,} && \text{swap } (P_{(i,2,k)}, P_{(i,3,k)}),\\
&\text{if } ((S_{(i,2,k)}[2] \oplus k[2]) = 0) && \text{then,} && \text{swap } (P_{(i,1,k)}, P_{(i,2,k)}),\\
&\text{if } ((S_{(i,2,k)}[3] \oplus i[0]) = 0) && \text{then,} && \text{swap } (P_{(i,0,k)}, P_{(i,3,k)}).
\end{aligned}$$

Similarly, the following swaps are applied to the p-prism using the 4th sc-plane for all $i = 0, 1, 2, 3$ and $k = 0, 1, \ldots, 7$:

$$\begin{aligned}
&\text{if } ((S_{(3,j,k)}[0] \oplus k[0]) = 0) && \text{then,} && \text{swap}(P_{(0,j,k)}, P_{(1,j,k)}),\\
&\text{if } ((S_{(3,j,k)}[1] \oplus k[1]) = 0) && \text{then,} && \text{swap}(P_{(2,j,k)}, P_{(3,j,k)}),\\
&\text{if } ((S_{(3,j,k)}[2] \oplus k[2]) = 0) && \text{then,} && \text{swap}(P_{(1,j,k)}, P_{(2,j,k)}),\\
&\text{if } ((S_{(3,j,k)}[3] \oplus j[0]) = 0) && \text{then,} && \text{swap}(P_{(0,j,k)}, P_{(3,j,k)}).
\end{aligned}$$

4.3.3. *Nonlinear Transformation.* At this step of the compression function we collect and combine the data from the s-prism and p-prism to set up a nonlinear transformation that acts on the s-prism. The nonlinear transformation is specifically designed to resist pre-image attacks and related weaknesses.

The nonlinear transformation applies the following map to each entry of the s-prism.

$$S_{(i,j,k)} := (S'_{(i,j,k)} \oplus Pl_{(i,j,k)})^{-1} \oplus (S'_{P_{(i,j,k)}} \oplus Ph_{(i,j,k)})^{-1} \oplus H_{(i,j,k)},$$

for all $i, j = 0, 1, 2, 3$ and $k = 0, 1, \ldots, 7$. We now define the elements $S'_{(i,j,k)}$, $Pl_{(i,j,k)}$, and $Ph_{(i,j,k)}$.

As we discussed earlier, DFTs operate over the prime field $GF(17)$ and outcomes of the DFTs may be 5-bit entries. We assign the least significant 4-bits of the s-prism entries to S'. In other words:

$$S'_{(i,j,k)} = S_{(i,j,k)} \bmod 2^4, \quad \text{for } i, j = 0, 1, 2, 3 \text{ and } k = 0, 1, \ldots, 7.$$

Similarly, we pick the least significant 4-bits of the p-prism entries and assign them to Pl;

$$Pl_{(i,j,k)} = P_{(i,j,k)} \bmod 2^4.$$

Also, Ph is the concatenation of the 5th bit of $S_{(i,j,k)}$ and the remaining 3-bits of $P_{(i,j,k)}$ (recall that the entries of p-prism are 7-bit numbers).

$$Ph_{(i,j,k)} = (S_{(i,j,k)} \operatorname{div} 16)\,||(P_{(i,j,k)} \operatorname{div} 2^4).$$

If the message consists of a single chunk, the hash value is deduced from the s-prism at this point. Otherwise, s-hash algorithm behaves according to Merkle-Damgard scheme. The entries of s-prism are stored in h-prism as defined below, and the s-prism is refilled in the successive round.

$$H_{(i,j,k)} := S'_{(i,j,k)}.$$

However, the p-prism undergoes some rotations, called rubics rotations (see Figure 8), which are described as follows:

$$\begin{array}{lll}
\text{if } (k \equiv 0 \bmod 4) & \text{then} & P_{(i,j,k)} := P_{(i,j,k)}, \\
\text{if } (k \equiv 1 \bmod 4) & \text{then} & P_{(i,j,k)} := P_{(3-j,i,k)}, \\
\text{if } (k \equiv 2 \bmod 4) & \text{then} & P_{(i,j,k)} := P_{(j,i,k)}, \\
\text{if } (k \equiv 3 \bmod 4) & \text{then} & P_{(i,j,k)} := P_{(j,3-i,k)}.
\end{array}$$

4.4. **Hash Generation.** The spectral hashing algorithm can be configured to return hash values of 32-bit multiples in between 128-bits and 512-bits. These lengths clearly include the bit sizes 224, 256, 384 and 512.

The procedure is quite simple, and is applied whenever the final states of the s-prism and p-prism are reached. In other words, the desired hash string is generated after s-prism goes through the nonlinear transformation and p-prism is modified via the rubics rotations at the end of the final chunk's processing.

The bits of the hash value are selected from the s-prism entries determined by the s-hash generation table (sg-table). The sg-table is a co-ordinate matrix with rows corresponding to the bit positions of $S_{(i,j,k)}$ and columns corresponding to the least significant two bits of $P_{(i,j,k)}$. To determine which bits of the s-prism entries are used in the hash value, one iterates through the p-prism as follows.

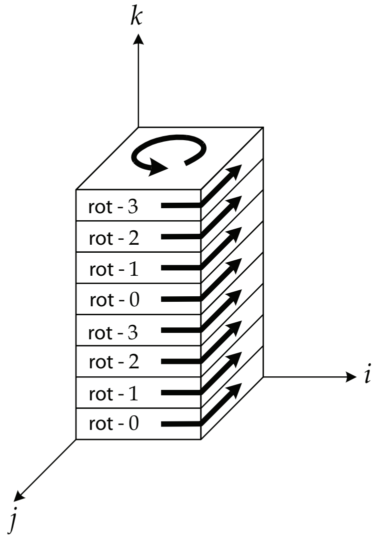


Figure 8. Rubics rotations on p-prism.

For each $P_{(i,j,k)}$, one determines the least two significant bits. Then one looks at the corresponding column of the sg-table, and for each cell in this column that has a star, one marks the corresponding bit of $S_{(i,j,k)}$ and adds it to the hash value. One continues in this manner for each $P_{(i,j,k)}$ in the p-prism. For example, Table 3 states that if the two least significant bits of $P_{(i,j,k)}$ are "00" (observe that 32 such entries exist) then the 0th bit of $S_{(i,j,k)}$ is marked and assigned to the hash value.

Notice that Table 3 presents a generation of $32 * 4 = 128$-bit hash value. The number of stars in the sg-table determines the length of the hash value. In fact, each star adds 32-bits to the final hash value. Therefore by adding more stars to the sg-table, one can generate longer hash values, up to 512-bits long (i.e. the s-prism itself). Specifically, a 224-bit hash generation requires an sg-table with 7 stars.

To create an sg-table, one places the desired number of stars in the table according to Table 2.

$P_{(i,j,k)}[1:0]$		00	01	10	11
bit	3	15	12	7	4
position	2	11	6	3	10
on	1	5	2	9	14
$S_{(i,j,k)}$	0	1	8	13	16

Table 2. Method of generating sg-tables.

$P_{(i,j,k)}[1:0]$		00	01	10	11
bit	3				*
position	2			*	
on	1		*		
$S_{(i,j,k)}$	0	*			

Table 3. 128-bit hash generation.

$P_{(i,j,k)}[1:0]$		00	01	10	11
bit	3			*	*
position	2		*	*	
on	1	*	*		
$S_{(i,j,k)}$	0	*			

Table 4. 224-bit hash generation.

$P_{(i,j,k)}[1:0]$		00	01	10	11
bit	3			*	*
position	2		*	*	
on	1	*	*		
$S_{(i,j,k)}$	0	*	*		

Table 5. 256-bit hash generation.

$P_{(i,j,k)}[1:0]$		00	01	10	11
bit	3		*	*	*
position	2	*	*	*	*
on	1	*	*	*	
$S_{(i,j,k)}$	0	*	*		

Table 6. 384-bit hash generation.

$P_{(i,j,k)}[1:0]$		00	01	10	11
bit	3	*	*	*	*
position	2	*	*	*	*
on	1	*	*	*	*
$S_{(i,j,k)}$	0	*	*	*	*

Table 7. 512-bit hash generation.

After the selection process, the resulting s-prism is called a punctured s-prism and resembles a swiss cheese. The final hash value is simply deduced from the final state of the punctured s-prism by reversing the message map indexing. Formally, the hash string

$$h := H_0 \,||\, H_1 \,||\, \ldots \,||\, H_{127},$$

consists of the 4-bit words $H_I = S_{(i,j,k)}$, where $S_{(i,j,k)}$ is now the punctured s-prism, and $I = 32i + 8j + k$ for $i, j = 0, 1, 2, 3,\ k = 0, 1, \ldots, 7$.

5. Security Considerations

In a nutshell, the security of the s-hash algorithm is derived from the three main mechanisms employed, namely the 3-dimensional DFTs, the inverse map in the field $GF(2^4)$, and the nonlinear transformation. As we mentioned earlier, the 3-dimensional DFT is the foundation of the s-hash compression function, which is shown to be a perfect cryptographic primitive for generating diffusion and confusion [1]. In order to build up security against differential and linear cryptanalysis, we augmented the compression function with the inverse map which has the best known nonlinearity [2] in the field $GF(2^4)$. Moreover, with the addition of the nonlinear transform, which is constructed by selecting system variables using a permutation table generated by data dependent permutations, s-hash becomes immune to pre-image attacks. We further discuss the resistance of these three structures to known cryptanalysis methods.

5.1. **General Birthday Attacks.** There are various cryptanalytic methods used to break hash functions, some of which are also related to block cipher cryptanalysis. These cryptanalytic methods generally focus on the collisions of a hash function, namely finding two messages m and m' such that $h(m) = h(m')$ and $m \neq m'$. The general birthday attack is a generic method of breaking hash algorithms that is reliant on the so called birthday paradox. A hash function is considered to be broken if the probability of collision is below the bound of $2^{(n/2)}$ messages. In s-hash, if 512-bit hash values are considered, one has to employ a search in a message space having cardinality 2^{256}. One way of avoiding such a search may be by guessing the form of the p-prism and then reversing the nonlinear transformation. However, such an attempt would require a search in the symmetric group S_{128} which has a cardinality of $128! \approx 2^{716}$.

5.2. **Differential Attacks.** When one goes through the steps of the compression function, it can be easily seen that the first and third steps consist of bijective functions. It is known that the inverse map in the field $GF(2^4)$ has the best known nonlinearity which does not permit one to find weak differentials such as those exploited in MD5, SHA-1, and other hash functions.

Moreover, being a perfect confusion and diffusion generator, the multi-dimensional DFT makes finding a differential path on s-hash infeasible. With the use of 3-dimensional DFTs, it is immensely hard to track the differential trails in order to find a collision, as even a one word (4 bit entry) change in the data prism affects all the words of the output data prism of the DFT step.

5.3. **Linear Cryptanalysis.** When it comes to linear cryptanalysis, one has to bear in mind that this attack has never been applied to hash functions. To use linear cryptanalysis, one has to approximate a function by linear equations. In the s-hash compression function, we make use of highly nonlinear functions like the inverse function. This function has the best known nonlinearity in the field $GF(2^4)$, which means that it is hard to correlate and has a high degree of nonlinearity. Therefore, linear cryptanalysis seems infeasible to use on s-hash.

5.4. **Further Comments.** The internal states of spectral hash are bijective transformations. Going back through step 3 to produce an "internal hash value" from a pre-specified hash value is hard because of the existence of the data dependent permutations in step 3. Thus, finding a matching internal state in order to construct a collision is not possible due to the fact that constructing different internal states requires finding inverses of different permutations, each of which is specified by a different message initially unknown to the adversary.

Moreover, the specification of data dependent permutations from a given message shows uniform distribution, which means that each different 512-bit message block generates a different permutation. As a result, the probability of finding a pre-specified message from the hash is $1/2^{512}$.

In conclusion, we conjecture that the spectral hashing algorithm is resistant to known attacks and it is not possible to find a collision under the complexity bound $O(2^{(n/2)})$ required for the birthday attack. It has pre-image resistance with complexity $O(2^n)$. Spectral hash also admits a random distribution which makes it a suitable candidate for an ideal cryptographic hash function.

6. Implementation Efficiency

The performance figures for the current implementation of s-hash are listed in Table 8. In our testing we found that the message digest length did not deterministically change the time required to compute the hash. In other words, the time variance for the different lengths was both negligible and inconsistent, and was not directly related to the size of the message digest. This is unsurprising considering the design of s-hash. If look-up tables for the various field operations are implemented memory usage is theoretically around 2KB; the reference implementation uses approximately 380KB.

Although s-hash can benefit from using extra memory, the amount of extra memory that it needs to reach its optimal speed is still very low and will scale linearly with the input message size. After optimizations are better implemented, the algorithm can run in less time and memory than the reference implementation does.

File Size	Speed (Mbits/S)	Cycles/Block
4 MB	38	29100
20 MB	38	29100
100 MB	38	29100

Table 8. Performance information for reference machine.

Processor	2.16GHz Intel Core Duo
Memory	1.5 GB 667MHz DDR2
OS	Mac OS X 10.5.3

Table 9. Reference Machine Specifications

6.1. **Hardware and 8-bit Processors.** S-hash has an advantage over other hashing algorithms when implemented on 8-bit processors or embedded type systems, because our optimal memory usage is low. Also, due to the design of s-hash, the operations on both prisms may be fully parallel and will compute in the same time, preventing wasted cycles. S-hash is also very well suited for 8-bit systems because it stores all of its data in fewer than 8 bit segments.

References

[1] C. P. Schnorr, "FFT-hashing: An efficient cryptographic hash function," in *rump session of the Crypto'91, Santa Barbara.* Aug. 1991, Springer, Berlin, Germany.

[2] K. Nyberg, "Differentially uniform mappings for cryptography," in *Advances in Cryptology, Proceedings Eurocrypt'93, LNCS 765, T. Helleseth, Ed.* 1994, pp. 55–64, Springer, Berlin, Germany.

[3] S. Lucks, "Design principles for iterated hash functions," in *Cryptology ePrint Archive 2004/253*, 2004.