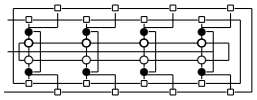
Attacks Against Permute-Transform-Xor Compression Functions and Spectral Hash

Ethan Heilman

August 26, 2009

Abstract

This paper presents an attack on the strong collision resistance of the Spectral Hash SHA-3 candidate. Spectral-Hash (shash) is a Merkle-Damgård based hash function, carefully designed to resist all known cryptographic attacks. To best of our knowledge, our attack is the only known attack against the shash algorithm. We exploit the fundamental structure of the algorithm, completely bypassing the hash function's formidable cryptographic protections. Our attack is presented in three stages. First, we define the family of functions which have the structure we wish to exploit. We call members of this family PTX functions. Next, we show that all PTX functions, including functions which use random oracles, are vulnerable to our collision attack. Finally, we reformulate the shash compression function showing that it is a PTX function and thus vulnerable. We present results on a practical implementation of our attack, generating collisions for shash in less than a second on a typical desktop computer.



1 Introduction

Spectral Hash[1] (shash) is a cryptographic hash function and a former candidate for selection as the NIST SHA-3 standard. Fundamentally, shash is a Merkle-Damgård construction[2] with two chaining variables. The authors of shash claim that because of the wide range of modern cryptographic protections, including affine transforms and discrete Fourier transforms, it is not possible to find a collision under the complexity bound of $O(2^{(n/2)})$. Unfortunately, we find that the shash algorithm is vulnerable to collisions in the chaining variables, allowing an attacker to generate collisions for shash in under a second.

On November 2008, Bjørstad found a collision and a preimage of zero[3] in the reference implementation of shash. The Spectral Hash team claimed that Bjørstad's findings were the result of an error in their reference implementation, rather than a weakness in their algorithm. An updated shash reference implementation was released which slightly changed the nature of the cycles and thus prevented the particular collision and preimage discovered by Bjørstad. Our attack shows that the weakness the Bjørstad attack depended on was not purely an implementation error. Rather, the Bjørstad attack depended on the same weakness we exploit in this paper (cycles in the internal state), but unfortunately made use of implementation specific details.

The collision attack presented in this paper, though similar in some respects to the Bjørstad attack, is a new result. Our attack is the only known attack on the corrected shash implementation and the only known attack on the shash algorithm itself. Furthermore, our attack generalizes across all hash functions that use a compression function in the PTX family.

In Section 2, we give a brief outline of the shash algorithm, with special attention paid to the cryptographic protections employed by the compression function.

In Section 3, we define a family of compression functions that we term PTX (Permute-Transform-Xor). We present collision attacks which work for all Merkle-Damgård constructs that use a compression function in the PTX family. By taking this approach, we show all functions which can be reformulated as PTX functions, including functions which use random oracles, are vulnerable. Then, we show that the shash compression function is a member of the PTX family; thus, the shash compression function is vulnerable to the same collision attacks.

Our attack works as follows: We choose inputs which induce cycles in one of the chaining variables (Section 3.2.1). Next, we exploit the strong relationship between the chaining variables to induce cycles

in both chaining variables (Section 3.2.2). Then, we use the cycles in the chaining variables to engineer collisions in the compression chain, which we exploit to generate collisions in the hash function (Section 3.2.4).

As the efficacy of our collision attacks depend on the order of some permutation π generated from a message block, in Section 4, we take a moment to explore this issue. Not only do we find that randomly generating message blocks is sufficient for our attacks, but we also present a method of generating message blocks to increase the efficacy of our attacks against shash.

In Section 5, we discuss the specific considerations encountered in carrying out our attacks against shash and results are presented on a practical implementation of our attack from Section 3.2.7. The implementation of our attack using a typical desktop computer shows that shash is broken both theoretically and practically.

Section 6 presents an additional weakness of shash, unrelated to our collision attack. Shash relies on two nonlinear functions to provide protection against linear and differential cryptanalysis. One of these functions, an Affine Transformation in $GF(2^4)$, has a flaw which prevents the function from increasing the cryptographic strength of shash. Although no differential or linear attacks against shash have been published, this weakness shows that shash may not provide the level of protection against differential and linear cryptanalysis that the authors intended.

1.1 A Word on Notation

A small amount of non-standard notation is used in this paper to express repeated concatenation and will now be introduced. We use the notation $a||b||c$ to express the concatenation of b onto a and c onto ab.

$$a||b||c = abc$$

We extend this notation so that $||a||^n$ represents n concatenations of a.

$$||a||^3 = a||a||a = aaa$$

A complex example:

$$a||b||c||^3 = a||b||c||c||c = abccc$$

$$1||2||3||..||7||8 = 1||2||3||4||5||6||7||8 = 12345678$$

2 The Spectral Hash Algorithm

In this section we provide a brief description of the shash algorithm. Since our attack exploits the structure of shash, without regard for the rest of the algorithm, we provide only a brief outline of the hash function. For example, we will not discuss the details of the finalization function used in shash, as it has no relevance to our attack. That being said, the description we provide is slightly more detailed than purely necessary to explain our attack. The reader may wish to skim this section, returning to fill in details as required. Furthermore we encourage any reader wanting a more complete description of the hash function to read the Spectral Hash NIST submission[1].

2.1 Overview

As Figure 1 shows, the Spectral Hash algorithm is built around a slightly modified Merkle-Damgård construction. Length padding[4] is used to strengthen shash against length extension attacks. A finalization function is used to generate the final hash from the output of the compression chain. Due to space constraints, the letter C in Figure 1 refers to the compression function ShashCompress.

The shash algorithm differs from a typical Merkle-Damgård construction in three ways. First, shash employs the large pipe scheme developed by Lucks [5] to increase the chaining state and thus, increase the difficulty of chaining variable collisions. To achieve the large pipe scheme, shash uses two chaining variables p and h. Second, the algorithm uses a finalization function to generate the hash from the output of the compression chain. Using the large pipe scheme makes the finalization function necessary because the chain state, which consists of p and h, is now larger than the hash output. Third, rather than use the same compression function on each message block, shash has a special compression function which it calls on the first message block. This initial compression function is covered in detail in Section 5.1.

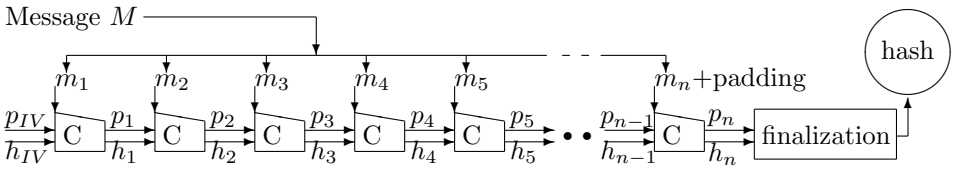


Figure 1: The Spectral Hash Algorithm.

2.2 The Variables m, p and h

The shash algorithm uses three variables m, p and h. The shash NIST submission documentation treats m, h and p as $4 \times 4 \times 8$ matrices that represent three dimensional prisms. While treating these variables as three dimensional objects is helpful in visualizing the shash algorithm, we found this notation burdensome and unnecessary in describing our attack. Instead, we prefer to unwrap these variables into one dimensional arrays. Thus, for the rest of this paper we treat m, h and p as arrays of 128 elements.

The message block variable m, is a 512 bit chunk of the message being hashed. Internally, shash treats m as an array of 128 elements, where each element is 4 bits. We prefer to treat m as a 512 bit number.

$$m \in \{0,1\}^{512}$$

The chaining variable p, is a sequence (permutation) of integers $0 - 127$.

$$p \in (a_0, a_1, a_2.., a_{127})$$

The variable p's initial value p_{IV}, is the sequence generated by sorting the numbers 0 to 127 from smallest to highest.

$$p_{IV} = 0, 1, 2, 3, ..127$$

Permutations are the only operations performed on p, making p a member of the symmetric group.

The chaining variable h, has exactly the same data structure as m. That is, h is a 512 bit number. Therefore, all possible values of m are possible values of h and all possible values of h are possible values of m. The initial value of h, h_{IV}, is all zero's.

$$h \in \{0,1\}^{512}$$
$$h_{IV} = \{0\}^{512}$$

The variables m, p and h describe completely the internal state of shash. At some points in this paper, we refer to variables p_{in}, p_{out}, h_{in}, and h_{out}. We do this merely as a way of distinguishing the input and output values of p or h. One could replace every instance of p_{in} and p_{out} with p and h_{in} and h_{out} with h.

2.3 Compression Function

ShashCompress(m, p, h)
1 $m \leftarrow AT(m)$
2 $p \leftarrow SwapControl1(m, p)$
3 $p \leftarrow SwapControl2(m, p)$
4 $m \leftarrow kDFT(m)$
5 $p \leftarrow SwapControl3(m, p)$
6 $m \leftarrow jDFT(m)$
7 $p \leftarrow SwapControl4(m, p)$
8 $m \leftarrow iDFT(m)$
9 $m \leftarrow NLST(h, m, p)$
10 $h \leftarrow m$
11 $p \leftarrow PlaneRotate(p)$
12 **return** h, p

The meat of shash is in its compression function, ShashCompress. A number of component functions are used to compose ShashCompress. We will discuss these component functions in the following paragraphs.

The affine transform AT is the first function called in ShashCompress. AT transforms the message block m by applying an affine transform in the field $GF(2^4)$. It is hoped by the authors that AT will lead to non-linearity in the compression function. In Section 6, we cover a weakness in AT.

After applying the affine transform, ShashCompress performs a series of transformations and permutations on m and p. The transformations consist of discrete Fourier transforms $[k, j, i]DFT$ applied to m. The DFT functions are designed by the authors to increase the hash function's resistance to Differential cryptanalysis. The permutations are caused by the functions $SwapControl[1, 2, 3, 4]$. These functions permute the elements of p using a permutation generated from m. The authors claim that the permutations increase the preimage resistance, because an attacker would have to guess the value of p to invert the compression function.

NLST(m, p, h)
1 $g \leftarrow d(p)$
2 **return** $g(m) \oplus h$

Next, the function NLST is called. NLST computes the final value of h. The function performs a series of non-linear equations and an xor. The non-linear equations use p to transform m to m'. We represent the non-linear equations as the function $d(p)$ that generates a function g, which is then applied to m to produce m'. The value m' is then xored against h, generating the final value of h.

$$\begin{aligned} g &\leftarrow d(p) \\ m' &\leftarrow g(m) \\ h &\leftarrow h \oplus m' \end{aligned}$$

The last step of ShashCompress is the function PlaneRotate. PlaneRotate permutes p to generate p's final value in the compression function. Unlike the permutation functions $SwapControl[1, 2, 3, 4]$ which permute p based on the value m, PlaneRotate performs the same permutation on p regardless of the value of p or m.

Notice that the compression function performs three general actions. It permutes p, it transforms m and it xors h. The next section introduces a family of functions which have this very same permute, transform and xor behavior. We will show that ShashCompress is a member of this family.

3 Collisions in PTX Compression Functions and Shash

In this section we introduce our attack against shash using a family of functions called Permute-Transform-Xor (PTX) functions. We take a guilt by association approach. First, we define a family of functions, which we call Permute-Transform-Xor (PTX). Second, we detail the mechanics of our collision attacks against all PTX functions. Finally, we show that shash is a member of the PTX family and thus, is vulnerable to our attacks against PTX functions.

3.1 The PTX family of Compression Functions

Permute-Transform-Xor (PTX) is a family of functions that we will use as compression functions in a Merkle-Damgård construction. The PTX family is useful to us because all functions in the PTX family are vulnerable to our attack. We define the PTX family by using a function, PTXCompress, which is composed of both defined and undefined functions. Since PTXCompress is composed of undefined functions, PTXCompress defines a set of functions. Our intention is to describe the set of functions vulnerable to our attack and to illustrate exactly the structures that the attack exploits, without muddying the waters with a discussion of the specific details of shash. We explain exactly how to carry out this attack on shash in Section 5.

A function is a member of PTX family, if and only if it can be formulated as the following function:

PTXCompress(m, p_{in}, h_{in})
1 $\pi \leftarrow f(m)$
2 $p_{out} \leftarrow \pi(p_{in})$
3 $g \leftarrow d(p_{out})$
4 $h_{out} \leftarrow g(m) \oplus h_{in}$
5 **return** h_{out}, p_{out}

Where π, p_{in}, p_{out}[1] are permutations of n unique elements. We constrain the variables h_{in}, h_{out} and m to be binary strings of the same length.

$$\pi, p_{in}, p_{out} \in (a_1, a_2, .., a_n)$$
$$h_{in}, h_{out}, m \in \{0,1\}^k$$

All PTX functions share the structure of generating a permutation π from the message block m. This permutation is used to permute p_{in} to p_{out}. Next, p_{out} is used to generate a function g. The generated function g maps some m to some m'. This m' is then xored against h_{in} to generate h_{out}.

We treat the functions f, d and g as arbitrary functions. Our attack will work without regard to the properties, cryptographic or otherwise, of f, d or g. To further illustrate this point, we will assume for the rest of this section that f, d and g are random oracles.

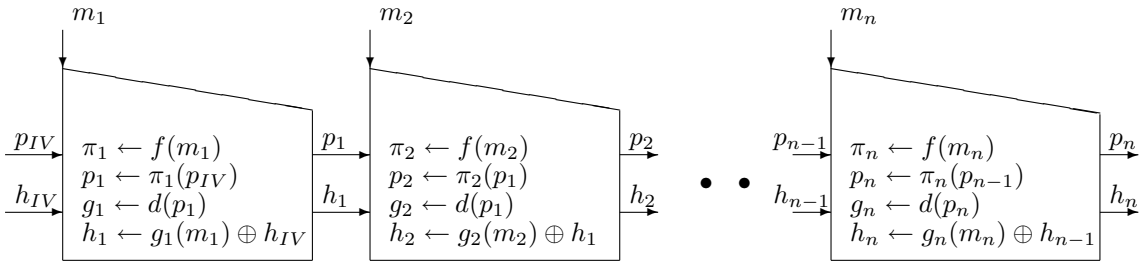


Figure 2: PTXHash: A Merkle-Damgård construction which uses a PTX compression function.

Figure 2 introduces PTXHash, which we define to be a Merkle-Damgård based hash function that uses a function in the PTX family as a compression function. We will be using PTXHash as the target of our collision attack in Section 3.2. In the context of a Merkle-Damgård construction, the variable m is a message block and the chaining variables are h and p.

3.2 Attacking Strong Collision Resistance

We can break strong collision resistance in PTXHash. Our approach is to induce cycles in the chaining variables, then exploit the cycles to generate collisions. In Section 3.2.1, we show that we can induce cycles in p. In Section 3.2.2, we show that if p cycles, h must cycle as well. Therefore, both chaining variables cycle, consequently producing collisions in the chaining variables. Figure 3 summarizes the cyclic relationship between a repeated message block and the chaining variable. In Section 3.2.3 we extend these chaining variable collisions to produce cyclic message chunks, namely a series of message blocks that cause both chaining variables to cycle. Cyclic message chunks are then used to compute 2nd preimages of chaining variables. Sections 3.2.4 - 3.2.8 build on this method of computing 2nd preimages of chaining variables to develop collision and multicollision attacks.

3.2.1 We can force p to cycle

Consider the case in which we provide a message M to $PTXHash$, where M consists of a concatenation of the same message block m_1 repeated n times (notation explained in Section 1.1).

$$M = m_1||m_1||m_1||..||m_1 = ||m_1||^n$$

The permutation π that is applied to p is entirely dependent on the value of m.

$$\pi = f(m)$$

As long as we supply the same message block m, f will produce the same π. The fact that the same π is generated each time allows us to cycle p. For any permutation π there exists some n such that $\pi^n = I$, where n is the order of π[6] and I is the identity permutation. That is, if the message block m_1 is repeated $n = order(\pi_1)$ times, p will cycle back to its initial state.

$$p_3 = \pi_1(\pi_1(\pi_1(p_{IV}))) = \pi_1^3(p_{IV})$$
$$p_n = \pi_1(\pi_1(..\pi_1(p_{IV})..)) = \pi_1^n(p_{IV}) = p_{IV}$$

To summarize, we can force p to cycle, if we repeat the same message block $order(\pi)$ times.

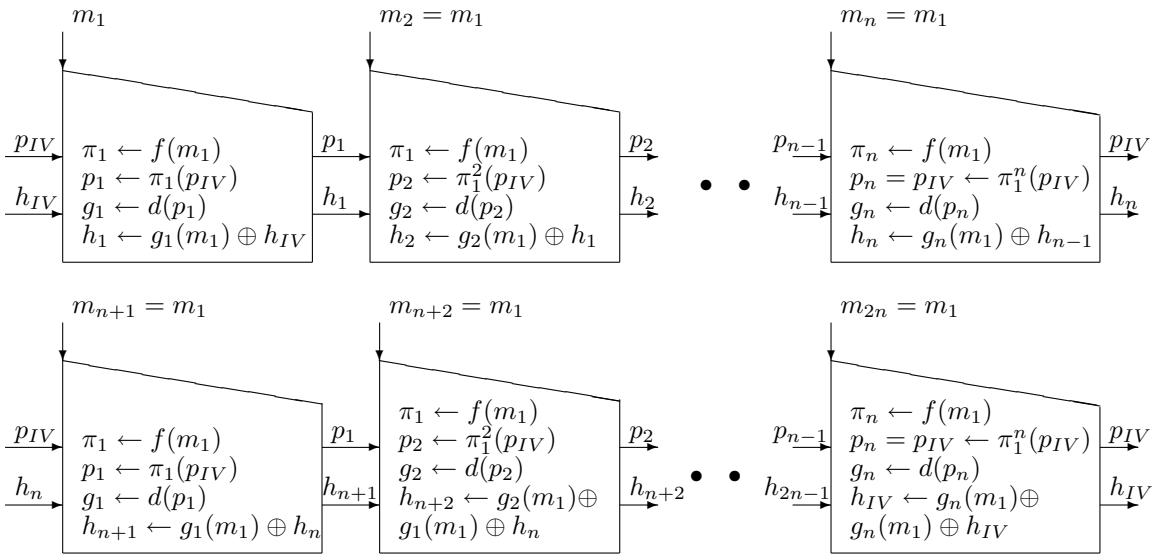


Figure 3: PTXHash supplied with the same message block $2n$ times, where $n = order(\pi_1)$.

3.2.2 If p cycles, h cycles.

The function g is entirely dependent on p, hence, if p returns to a previous value, so does g. Therefore, we can see that a cycle in p will cause a cycle in g.

$$g_1 \leftarrow d(p_1), g_2 \leftarrow d(p_2), g_3 \leftarrow d(p_3), .. \; p_n \text{ cycles to } p_1, \; g_1 \leftarrow d(p_1), g_2 \leftarrow d(p_2)...$$

Remember that we are repeating the same message box m_1. Thus, if g cycles, the values that g produces will cycle as well.

$$g_1(m_1), g_2(m_1), g_3(m_1), .. \; g_n \text{ cycles to } g_1, \; g_1(m_1), g_2(m_1)...$$

The output of each of these g functions is xored against h. The value of h can always be defined as an initial value h_{IV} xored against a series of g functions.

$$h = h_{IV} \oplus g_1(m_1) \oplus g_2(m_1) \oplus g_3(m_1)...$$

When g cycles a second time, the values generated on the second cycle will be the same as the values generated on the first cycle. These repeated values, when xored against h cancel themselves out. Consider the case in which n repetitions of the message block m_1 cause g to cycle. We show below that value of h after the first cycle of p, is h_n. We also show that after the second cycle of p, h has cycled back to its initial value of h_{IV}.

$$h_n = h_{IV} \oplus g_1(m_1) \oplus g_2(m_1)... \oplus g_n(m_1)$$
$$h_{2n} = h_n \oplus g_1(m_1) \oplus g_2(m_1)... \oplus g_n(m_1)$$
$$h_{2n} = h_{IV} \oplus g_1(m_1) \oplus g_2(m_1)... \oplus g_n(m_1) \oplus g_1(m_1) \oplus g_2(m_1)... \oplus g_n(m_1)$$
$$h_{2n} = h_{IV}$$

Therefore, if g cycles twice and the message block m is fixed, h cycles once (See Figure 3).

3.2.3 Cyclic Message Chunks, Collisions and 2nd Preimages of Chaining Variables

We refer to a series of message blocks as a message chunk, using the letter C to denote a message chunks. We use this notation throughout the paper, C always refers to a message chunk.

$$C = m_1||m_2||..||m_n$$

We can compute the effect that a message chunk has on the compression chain by iteratively compressing each of the message chunk's component message blocks.

$$h, p = \text{IteratedCompress}(C, p_{IV}, h_{IV}) = \text{Compress}(m_n, (..\text{Compress}(m_2, \text{Compress}(m_1, p_{IV}, h_{IV}))..))$$

In Section 3.2.1 we showed that by repeating a message block $order(\pi)$ times, p cycles. In Section 3.2.2 we showed that for every two cycles of p we have one cycle of h. Thus, when some message block is repeated $2 \times order(\pi))$ times, p cycles twice and h cycles once, causing a chaining variable collision in which the current state of the compression chain collides with an earlier state of the compression chain.

We refer to any message chunk which causes both chaining variables to collide with values held directly prior to the message chunk as a cyclic message chunk. That is, a cyclic message chunk is a message chunk which returns the chaining variables passed to it.

$$\begin{aligned}
&\pi_a = f(m_a) \\
&C_{cyclic} = ||m_a||^{2 \times order(\pi_a)} \\
&h_x, p_x \in \{0,1\}^k \\
&h_x, p_x = \text{IteratedCompress}(C_{cyclic}, h_x, p_x)
\end{aligned}$$

Cyclic message chunks can be used to create 2nd preimages of chaining variables and consequently chaining variable collisions. One is said to find a 2nd preimage of chaining variables when given a message chunk C_a, one can find another message chunk C_b, such that $C_a \neq C_b$ and both chunks generate the same values for the chaining variables.

$$\text{IteratedCompress}(C_a, h_x, p_x) = \text{IteratedCompress}(C_b, h_x, p_x)$$

To find a 2nd preimage of chaining variables for some message chunk C_a, append any cyclic message chunk on the end of C_a creating a message chunk C_b.

$$C_b = C_a||C_{cyclic}$$

The message chunk C_b will always be a preimage of C_a since C_{cyclic} cycles back to the chaining variables h_a and p_a produced by C_a.

$$\begin{aligned}
&h_a, p_a = IteratedCompress(C_a, h_x, p_x)) \\
&h_a, p_a = IteratedCompress(C_{cyclic}, h_a, p_a)) \\
&h_a, p_a = IteratedCompress(C_b, h_x, p_x))
\end{aligned}$$

Thus, any cyclic message chunk can be used to generate 2nd preimages and collisions in the chaining variables of all PTX functions for all possible message chunks.

Our method for using cyclic message chunks to generate 2nd preimages of chaining variables always generates a preimage which is longer than the original message chunk.

$$|C_a| < |(C_a||C_{cyclic})|$$

Consequently, we can only generate 2nd preimages of full messages against PTX functions not employing length padding. Shash employs length padding. In the following sections we will show how cyclic message chunks can be marshaled to generate collisions in PTX functions protected by length padding.

3.2.4 Generating Collisions

The intuition here is to leverage the properties of cyclic message chunks to generate collisions while ensuring that both resultant messages are of equal length and thereby avoid the protection offered by length padding. Using the cyclic message chunks as introduced in Section 3.2.3 we have developed three attacks which generate colliding messages: Move (Section 3.2.5), Substitute (Section 3.2.6) and Shuffle (Section 3.2.7). The attacks presented can be used in combination, or separately.

For each of our attacks there is a linear relationship between the length of the cyclic message chunks used and the length, $|M|$, of the colliding messages. We can express this relationship as a linear equation in which $C_1, C_2, ..C_n$ represent the cyclic message chunks used and the variables a and b depend on which attack or combinations of attacks are used.

$$|M| = a \times |(C_1||C_2||..||C_n)| + b$$

For each attack, we show that increasing the number cyclic message chunks enables us to produce many multicollisions. Furthermore, in Section 3.2.8 we show that by adding arbitrary message blocks we generate new collisions from the already discovered collisions. Thus, we can create new collisions, and/or new multicollisions by slightly increasing the colliding message length, $|M|$.

Our attacks are introduced in order of descending generality. Our first attack, Move, works for any cyclic message chunk. Our second attack, Substitute, works on any two or more cyclic message chunks of the same length. Our third and final attack, Shuffle, works on two or more cyclic message chunks of the same length that produce the same value for π. Accordingly, any cyclic message chunks that work for Shuffle also work for Substitute and any cyclic message chunks that work for Substitute work for Move.

3.2.5 Move Cyclic Message Chunks to Generate Collisions and Multicollisions

The Move attack is our most basic method for generating collisions. Move can create a collision from any cyclic message chunk. We merely change the location of a cyclic message chunk in a message to generate a colliding message. If the cyclic message chunk has more than two locations to occupy, Move can generate multicollisions. Generate some cyclic message chunk C_a composed of some message block m_a.

$$\pi_a = f(m_a)$$
$$C_a = m_a||m_a||..||m_a = ||m_a||^{2\times order(\pi_a)}$$

Then, choose an arbitrary message block m_1. Our first message, M_{a1}, is created by appending m_1 to C_a.

$$M_{a1} = C_a||m_1$$

Now create a colliding message M_{1a} by moving m_1 to the front the message.

$$M_{1a} = m_1||C_a$$

Cyclic message chunks cycle the value of h and p to the values they held directly prior to the cyclic message chunk, therefore moving a cyclic message chunk does not effect the final result of the compression chain.

$$h_1, p_1 = \text{Compress}(m_1, h_{IV}, p_{IV})$$
$$h_1, p_1 = \text{IteratedCompress}(m_1||C_a, h_{IV}, p_{IV}) = \text{IteratedCompress}(C_a||m_1, h_{IV}, p_{IV})$$
$$\text{PTXHash}(M_{a1}) = \text{PTXHash}(M_{1a})$$

Length padding offers no protection, as we have clearly not changed the length of the message.

$$|M_{a1}| = |M_{1a}| = |(C_a||m_1)| = |(m_1||C_a)|$$

Since we can move a cyclic message chunk without changing the result of the compression chain, if we have n cyclic message chunks, $C_1, C2, ..C_n$, all possible positions of these cyclic message chunks produce colliding messages; consequently we can generate $n!$ multicollisions.

$$\text{PTXHash}(m_1||C_1||C_2||..||C_n) = \text{PTXHash}(C_1||m_1||C_2||..||C_n) = \text{PTXHash}(C_2||m_1||C_1||..||C_n) = ...$$

3.2.6 Substitute Cyclic Message Chunks of Equal Length to Generate Collisions

Unlike the Move (see Section 3.2.5) attack which works for all cyclic message chunks, the Substitute method can only be to applied to cyclic message chunks which have the the same length.

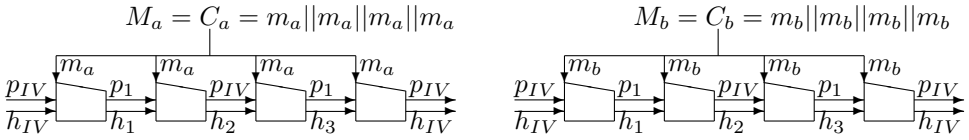


Figure 4: Two colliding messages created by two cyclic message chunks of equal size.

.

Two cyclic message chunks that share the same initial chaining variables also will output the same shared initial chaining variables generating a collision in the chaining variables.

$$h_{IV}, p_{IV} = \text{IteratedCompress}(C_a, h_{IV}, p_{IV})$$
$$h_{IV}, p_{IV} = \text{IteratedCompress}(C_b, h_{IV}, p_{IV})$$

If these two cyclic message chunks have equal length, then length padding offers no protection.

$$|C_a| = |C_b|$$

Therefore as Figure 4 shows, a cyclic message chunk can be substituted for another cyclic message chunk of the same length to generate colliding messages.

$$\text{PTXHash}(C_a) = \text{PTXHash}(C_b)$$

If we have n cyclic message chunks of the same length, we can generate n colliding messages. So, if $n > 2$, we can generate n multicollisions.

3.2.7 Shuffle cyclic message chunks of Equal Permutation to Generate Collisions

Our strategy here is to create two messages such that the chaining variable p continuously collides between messages, and then to exploit the commutative nature of h's construction to create a collision. Figure 5 demonstrates our method in the form of an example.

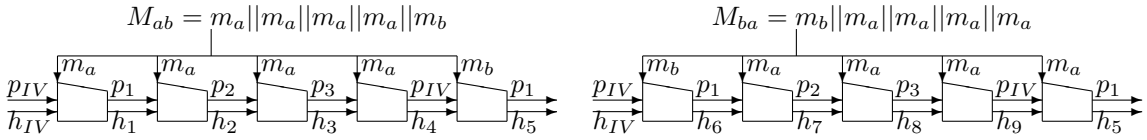


Figure 5: Two colliding messages created by switching the position of the first and last message block.

The Shuffle method requires two message blocks m_a and m_b that generate the same permutation π.

$$m_a \neq m_b$$
$$\pi = f(m_a) = f(m_b)$$

We use one of these message blocks, m_a to create a cyclic message chunk C_a.

$$C_a = ||m_a||^{2 \times order(\pi)}$$

The first of the two colliding messages, M_{ab} consists of the concatenation of the first half of a cyclic message chunk C_a and the second message block m_b.

$$C_a^{1/2} = ||m_a||^{\frac{2 \times order(\pi)}{2}} = |m_a||^{order(\pi)}$$
$$M_{ab} = C_a^{1/2}||m_b = m_a||m_a||..||m_a||m_b$$

We build a second message M_{ba} by switching the position of the first and last message block in M_{ab}.

$$M_{ba} = m_b||C_a^{1/2} = m_b||m_a||m_a||..||m_a$$

By choosing message blocks m_a and m_b, such that they generate the same π, we ensure that the values of p will be the same under both M_{ab} and M_{ba}, because the value of p is generated from both the previous value of p and the value of π.

$$p_1 = \pi(p_{IV}), p_2 = \pi(p_1), p_3 = \pi(p_2), p_4 = \pi(p_3)...$$

Thus, although p is changed after each compression, p can not distinguish between the two messages M_{ab} and M_{ba} as they consist entirely of message blocks which generate the same value of π.

The value of p is used to create a function g, which transforms m into m'. The value of m' is xored into h. As a result, compressing the same message block will not result in the same value being xored into h if the value of p is different.

$$g \leftarrow d(p_{out})$$
$$h_{out} \leftarrow g(m) \oplus h_{in}$$

Therefore, p prevents us from exploiting the commutative nature of xor by switching the positions of any two message blocks. However, due to C_a being half a cyclic message chunk, the value of p in M_{ab} cycles once, causing the first and last message blocks of m_{ab} to be compressed using the same value for p (as shown in Figure 5). Accordingly, if we switch the message blocks m_a and m_b in the first and last position of M_{ab} we generate a new message M_{ba}, for which the same values are xored into h, but in a different order.

$$h_{final} = h_{IV} \oplus g_1(m_a) \oplus g_2(m_a) \oplus .. \oplus g_{order(\pi)-1}(m_a) \oplus g_1(m_b)$$
$$h_{final} = h_{IV} \oplus g_1(m_b) \oplus g_2(m_a) \oplus .. \oplus g_{order(\pi)-1}(m_a) \oplus g_1(m_a)$$

As the xor operation is commutative it generates the same final value for h despite the switched message blocks.

We have caused a collision in the final value of both chaining variables, both messages are of equal length, hence both messages must result in the same hash value.

$$\text{PTXHash}(M_{ab}) = \text{PTXHash}(M_{ba})$$

The length of the colliding messages $|M|$, generated by the Shuffle method is linear to the order of the permutation π shared by m_a and m_b. We can express this relationship as

$$|M| = order(\pi) + 1$$

Hence, the Shuffle attack generates smaller colliding messages than either the Move or Substitute attacks.

We can generate multicollisions using Shuffle approach, by creating messages that have more than one m_b message block. The extra m_b message blocks allow us more than two ways to shuffle the message blocks, allowing us to produce more than two colliding messages. For example consider the following three colliding messages.

$$\text{PTXHash}(m_a||m_a||m_a||..||m_b||m_b) = \text{PTXHash}(m_b||m_a||m_a||..||m_a||m_b) = \text{PTXHash}(m_b||m_b||m_a||..||m_a||m_a)$$

Our Shuffle attack requires that we have atleast two message blocks which generate the same π. Finding such message blocks depends completely on both how $f(m)$ calculates π, and on the range of values π is capable of representing. The only way to find colliding values of π in a random oracle PTX function is the birthday attack. Accordingly, this means that finding colliding values of π in a random oracle PTX function is computationally impractical for all non-trivial sizes of π. Shash is not a random oracle and, as we show in Section 5.2, given any message block producing some permutation π, we can easily generate enormous numbers of other message blocks that produce the same π.

3.2.8 Find One Collision get Additional Collisions Free

Bonus collisions are collisions which can be generated from an already discovered pair of colliding messages M_c, and M_d using only randomness and concatenation.

$$m_{random} \in \{0,1\}^k$$
$$M_c \neq M_d$$
$$\text{Hash}(M_c) = \text{Hash}(M_d)$$
$$\text{Hash}(m_{random}||M_c) = \text{Hash}(m_{random}||M_d)$$

Bonus collisions are in no way required to be, or not be, multicollisions.

As our collision attacks are completely indifferent to the state of the chaining variables once a collision has been created in both chaining variables, that collision will be maintained across both messages as long as neither message diverges.

Consequently, we can take use any pair of colliding messages M_c and M_d and generate unlimited pairs of colliding messages by merely appending or prepending one or more arbitrary message blocks.

$$\text{Hash}(m_1||m_2||..||m_n||M_c||m_{n+1}||m_{n+2}||..||m_{n+i}) = \text{Hash}(m_1||m_2||..||m_n||M_d||m_{n+1}||m_{n+2}||..||m_{n+i})$$

It should be evident that bonus collisions are fundamentally more dangerous then just collisions because bonus collisions allow an attacker to generate colliding message pairs which contain arbitrary data of the attackers choosing.

3.3 The compression function of Shash is a PTX function

Now that we have shown that we can carry out collision attacks on PTX functions, we show that we can carry out such attacks on shash. To demonstrate that shash is vulnerable, we show that the shash compression function, ShashCompress, is a member of the PTX family by reformulating ShashCompress as a PTX function. Consider a table that contains all possible input output pairs for ShashCompress. If any of our changes alter this table then we fail to show that ShashCompress is a PTX function. Thus, we must be careful to show that our changes do not alter the input output pairs of ShashCompress. Our reformulation takes three steps. We show each step of the reformulation in Figure 6.

Step 1 splits each of the chaining variables into input and output variables, so that h becomes h_{in} and h_{out} and p becomes p_{in} and p_{out}. This is done to bring ShashCompress into agreement with the naming scheme used for chaining variables in the PTX definition. As this change is merely a renaming of internal variables, it has no effect on the input/output values of ShashCompress.

Step 2 consists of three modifications. First, we restate the message block variable m as a constant by creating three new variables m_1, m_2, m_3, each standing in for a different transformation of m. This change, like the change in step 1 is merely a relabeling of values and has no impact on the actual values themselves. The second change is to replace the call to NLST with the internals of the NLST function(Line 9 and 10). The third change is to the order in which we call the PlaneRotate function. We need to change the order because, in a PTX function, the value of p_{out} does not change after d is invoked, but in ShashCompress the value of p_{out} is altered by PlaneRotate after d is called. To get around this discrepancy, we allow PlaneRotate to alter the value of p_{out} before we call d, but we invoke InversePlaneRotate inside of d to retrieve the earlier value of p. The final value of p will be the same and d is able to recover the pre-plane-rotate value of p thus, the input/output relationship of ShashCompress is preserved despite rearrangement.

In Step 3, we refine the changes made in Step 2, so that ShashCompress is expressed completely in the form of a PTX function. We create a function f that produces the permutation π based on the message block m. We no longer permute p directly, but rather we call the newly created function f to generate the permutation π used to permute p. As far as the output of ShashCompress is concerned, it doesn't matter if the permutations are applied to p initially or used to generate a permutation which then permutes p. Our final change is to move the transformations variables of m into the internals of f and g.

STEP 1:

```
ShashCompress(m, p_in, h_in)
 1  m ← AT(m)
 2  p_out ← SwapControl1(m, p_in)
 3  p_out ← SwapControl2(m, p_out)
 4  m ← kDFT(m)
 5  p_out ← SwapControl3(m, p_out)
 6  m ← jDFT(m)
 7  p_out ← SwapControl4(m, p_out)
 8  m ← iDFT(m)
 9  m ← NLST(h_in, m, p_out)
10  h_out ← m
11  p_out ← PlaneRotate(p_out)
12  return h_out, p_out

NLST(m, p, h)
1  g ← d(p)
2  return g(m) ⊕ h
```

STEP 2:

```
ShashCompress(m, p_in, h_in)
 1  m_1 ← AT(m)
 2  m_2 ← kDFT(AT(m))
 3  m_3 ← jDFT(kDFT(AT(m)))
 4  p_out ← SwapControl1(m_1, p_in)
 5  p_out ← SwapControl2(m_1, p_out)
 6  p_out ← SwapControl3(m_2, p_out)
 7  p_out ← SwapControl4(m_3, p_out)
 8  p_out ← PlaneRotate(p_out)
 9  g ← d(InversePlaneRotate(p_out))
10  h_out ← g(iDFT(m_3)) ⊕ h_in
11  return h_out, p_out
```

STEP 3:

```
ShashCompress(m, p_in, h_in)
 1  π ← f(m)
 2  p_out ← π(p_in)
 3  g ← d(InversePlaneRotate(p_out))
 4  h_out ← g(ijkDFT(AT(m))) ⊕ h_in
 5  return h_out, p_out

f(m)
 1  m_1 ← AT(m)
 2  m_2 ← kDFT(AT(m))
 3  m_3 ← jDFT(kDFT(AT(m)))
 4  π ← SwapControl1(m_1)
 5  π ← SwapControl2(m_1, π))
 6  π ← SwapControl3(m_2, π)
 7  π ← SwapControl4(m_3, π)
 8  π ← PlaneRotate(π)
 9  return π
```

Figure 6: Reformulating the shash compression function as a PTX function.

None of these change alter the input/output table of ShashCompress. Furthermore these changes allow ShashCompress to be expressed as a PTX function (Figure 6). Therefore, we have shown that ShashCompress is a member of the PTX family of functions and vulnerable to the collision attack presented in Section 3.2.

4 The Order of π

Our attacks depend on an assumption that the order of some permutation π, generated by some message block m, will be of a reasonably small size, enough of the time, to make our attack practical. The function f is used to compute π from m

$$\pi \leftarrow f(m).$$

In this section we show that our assumption is completely sound.

As shown in Section 3.2.4, the length of a colliding message is linear to the length of its component cyclic message chunks. In our attack Move (see Section 3.2.5), any cyclic message chunk and any other message block can be easily composed to produce two colliding messages. Thus given some cyclic message chunk C_a, composed of some repeated message block m_a, and a random message block m_b, we can find two colliding messages of length

$$|M| = |C_a| + |m_b|.$$

In shash the message block size is 128 Bytes. So the length of such a colliding message, where $\pi_a = f(m_a)$ is

$$|M| = 128 \times 2 \times order(\pi_a) + 128.$$

In writing this section we were unsure where to draw the distinction between reasonable and unreasonable message length. Cryptographic hash functions are used on data sets well in excess of 100 MB (CDs 700 MB, DVDs 4.7 GB, etc) and as little as $8 - 16$ Bytes for passwords and small files. Unable to decide on a maximum reasonable message length, we instead choose a maximum low enough to be obviously reasonable, yet high enough to include a significant percentage of our random message blocks. We choose 256 KB as our reasonable message length constraint. Accordingly, we assume that all cyclic message chunks with a length of greater than 256 KB are too long Thus, we want to find all cyclic message chunks which have a length less than 256 KB, that is all cyclic message chunks for which $order(\pi) \leq 1000$.

We first considered justifying our assumption about order of π using the mathematical upper bound of a random permutation. Landau's function is defined as the maximal order element of the symmetric group S_n[7]. Thus, Landau's function is the strict upper bound of the order of a randomly sampled permutation of n items. In shash, where $n = 128$, the maximal order is 7216569360 or approximately $2^{32.75}$. While this may seem discouraging, it has been shown in the literature[8][9] that the maximal order of a symmetric group is

not at all representative of the other elements in that group. Thus, the maximal order does not provide an effective estimate of the order of a random permutation π.

$$\pi \in S_n$$

Therefore, keeping the upper bound in mind we will instead experimentally sample π to determine the difficulty of finding low order permutations.

We justify our assumptions about the order of π in both a completely random scenario, as is the case with a random oracle PTX function (the function f is a random oracle), and a scenario in which we supply random message blocks to shash which are then used to generate π (the function f as specified in shash). We will constrain ourselves to PTX functions for which π is a permutation of 128 elements, as this is the number of elements which shash uses for π.

Finally we present work on choosing shash message blocks such that the number of permutations which f applies to π reduced. Our hope is that these reduced permutation message blocks have an increased chance of producing very low order values of π.

4.1 Order of a Random π

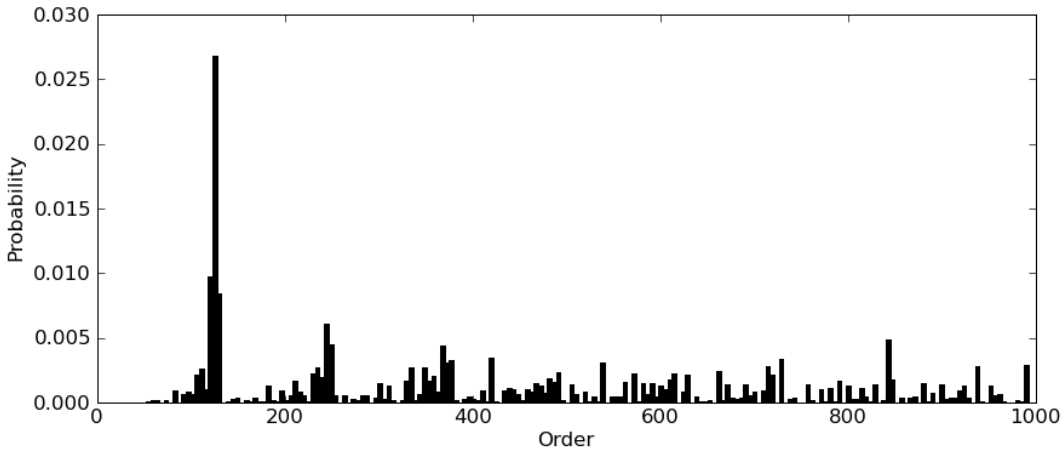


Figure 7: Distribution of $order(\pi) \leq 1000$ for a random oracle PTX function (π is randomly chosen).

For a random oracle PTX function, the value of π is chosen completely randomly, thus to determine the distribution of values we merely find the order of random elements sampled from the Symmetric group S_{128}.

$$\pi \in S_{128}$$

In our tests we took $100,000$ samples of random permutations. Of these samples, $19,235$ or roughly 19.2%, had an order of less than 1000. In Figure 7, we present the distribution of the samples that have an order of 1000 or less.

4.2 Shash and the Order of π for Random Message Blocks

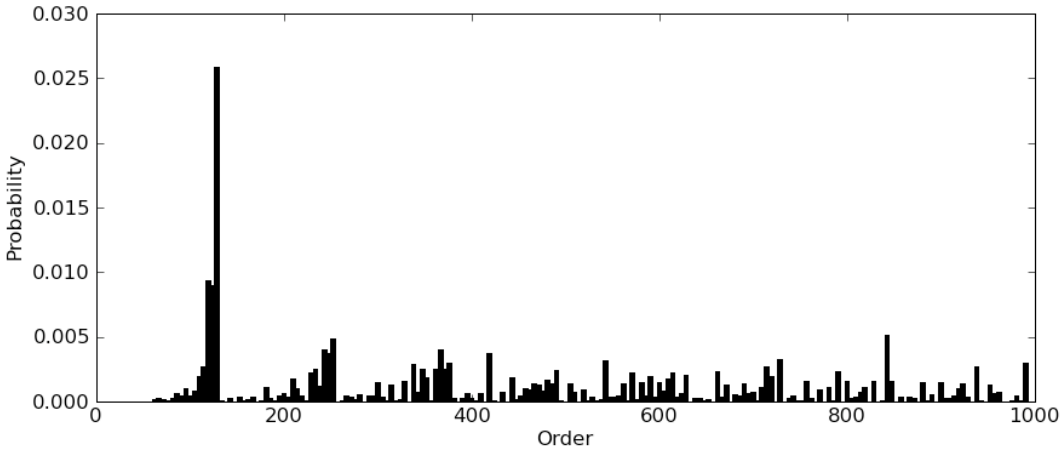


Figure 8: Distribution of $order(\pi) \leq 1000$ for shash (m is randomly chosen)

Shash does not generate π randomly, but rather performs computations on the message block to produce π. Thus, to test shash we can not randomly generate π, as in the previous section. Instead we must randomly generate message blocks, which shash will use to generate a permutation π. To test shash we randomly generated $100,000$ message blocks and tested the order of the resultant permutation π for each message block.

$$m \in \{0,1\}^{512}$$
$$\pi \leftarrow f(m)$$

For $100,000$ samples, $19,282$ or roughly 19.2% of the message blocks produced π having an order of less than 1000. Figure 9 shows that the distribution of the samples for shash is very similar to the distribution of the samples generated completely randomly. The lowest order of π that this was able to generate was 32.

4.3 Shash and the Order of π for Reduced Permutation Message Blocks

Here we briefly describe our attack on the function f which shash uses to compute π. The aim of our attack is to generate values of π of a lower order than produced by random sampling (Section 4.2). We present a method of generating message blocks for which the value of π will have undergone significantly fewer permutations. Finally we test this method and present our results.

Our intuition here is that the function f used to compute the permutation π from the message block m is unlike a random oracle.

$$\pi \leftarrow f(m)$$

As implemented in shash, the function f produces only limited diffusion/confusion and can be easily inverted[2] for valid values of π.

$$m \leftarrow f^{-1}(\pi)$$

Using the weakness in the AT function presented in Section 6, we are able to feed the functions SwapControl1 and SwapControl2 arbitrary values of our choosing. This allows us to solve for the equation of a message block such that the functions SwapControl1 and SwapControl2[3] do not introduce permutations into

[2]Creating a low order π from scratch and then inverting it to find the generating message block, m, is not a practical attack. Only a small fraction of potential values of π map to a message block. Although π in shash represents a permutation of 128 elements, shash message blocks are only capable of producing 2^{296} different values of π (See Section 5.2). Thus, the probability of choosing a value of π which maps to a message block is

$$\frac{2^{296}}{128!} = \frac{1}{3.30155566 \times 10^{127}}.$$

[3]A brief discussion of the AT, SwapControl, kDFT and PlaneRotate functions can be found in Section 2.3. For a description of the relationship that these functions have to π see our PTX reformation of shash in Section 3.3.

π and that the function SwapControl3 both does not introduce any cycles of size greater than 2 and that such cycles only occur at locations which PlaneRotate does not permute[4]. This equation can be found in Section A.2 of the appendix.

We build on this equation to generate reduced permutation message blocks. As the equation does not successfully solve for SwapControl4, we randomly choose values for the bits of the message block that SwapControl4 uses to permute π, hoping to guess correct values and thus generate a very low order permutation.

We generated $100,000$ reduced permutation message blocks, $14,385$ or roughly 14.3% percent had an order of less than 1000. As Figure 9 shows the reduced permutation method greatly improved our ability to generate very low order message blocks. It even found several message blocks of $order(\pi) = 12$, which is a third the size of the lowest order message block, $order(\pi) = 32$, found by randomly sampling m.

Even though our method leads to lower order values of π, it produces fewer values of $order(\pi) \leq 1000$ than the random methods. A full discussion of this seeming contradiction being beyond the scope and ambition of this paper, we will instead provide our best guess based on three observations and an assumption. The order of a permutation is the greatest common multiple (GCM) of the cycles in that permutation, thus to maximize the order of permutation, find the smallest cycles that are relatively prime to each other[5]. Small numbers are more likely to be relatively prime to each other than large numbers. We assume our method generates very low order permutations by restricting the size of the cycles. Therefore, we think that our method is unbalancing the message blocks so that they are more likely to generate either very small order permutations and very large order permutations.

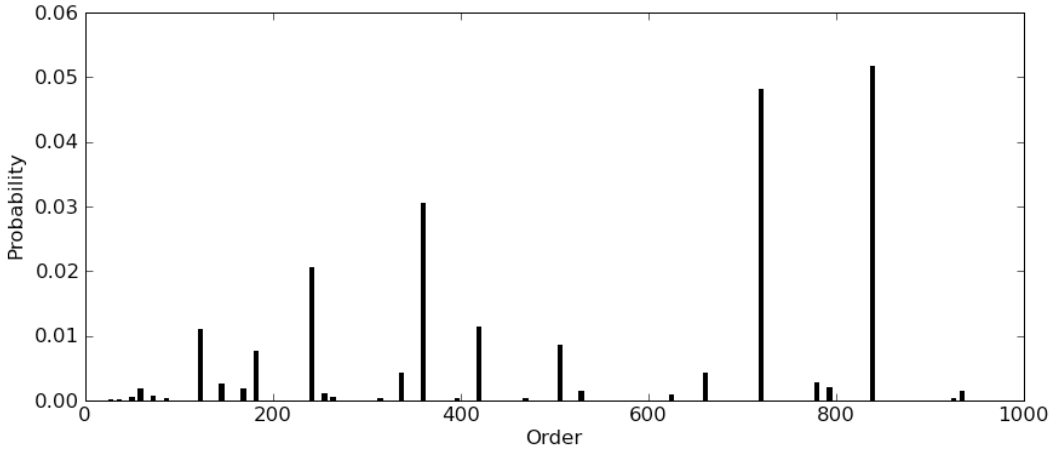


Figure 9: Distribution of $order(\pi) \leq 1000$ for shash (m chosen to reduce swaps).

4.4 Our Assumptions about the Order of Permutations are Sound

Essentially one in every five random message blocks generates a permutation π of a sufficiently low order. We showed that by merely picking random message blocks we can easily find message blocks which can be used to build cyclic message chunks of a small size ($\leq$ 256KB). This confirms our assumptions about the likely low order of a permutation for both random oracle PTX functions and shash. Moreover, we showed that by intelligently choosing the values of shash message blocks we can increase our chances of finding low order values of π and accordingly increase the efficacy of our shash collisions.

5 Practical Considerations

The attacks in Section 3.2 take a simple view of shash. While Section 3.2 is generally correct in its form, some complicating details are left out. We will address these details here. In Section 5.3 we present an implementation of this attack on the shash reference implementation.

[4]The full details of how we arrive at this equation, being particularly messy, long and dull, have been compassionately omitted to spare the readers patience and printer.

[5]Consider the gain in order caused by increasing a cycle by n elements, vs producing a second cycle of n elements.

5.1 Shash Uses Two Compression Functions

Shash has two compression functions. FirstShashCompress, which is invoked on the first message block and ShashCompress, which is invoked on all subsequent message blocks. FirstShashCompress is identical to ShashCompress except that FirstShashCompress performs an additional permutation π_{init} on p. Below, we represent FirstShashCompress as ShashCompress with an additional step.

$FirstShashCompress(m, p_{in}, h_{in})$
1 $\pi_{init} \leftarrow initSwap(m)$
2 **return** $ShashCompress(m, \pi_{init}(p_{in}), h_{in})$

To avoid complexities introduced by two different compression functions, we don't begin our attack until the second message block. Our collision attacks do not have any dependencies on specific prior values of h or p (see Section 3.2.8), so we just arbitrarily choose a first message block, m_{init}, before beginning our attack.

5.2 Generating Collisions in the Internal Permutation Variable π

Given some message block m_a that generates some permutation, $\pi_a = f(m_a)$, we easily produce some different message block m_b which generates some permutation $\pi_b = f(m_b)$ such that $\pi_b = \pi_a$. While message blocks that produce colliding values of π are not necessary for most of the collision attacks introduced in Section 3.2, they are a necessary condition for the Shuffle attack (Section 3.2.7), a sufficient condition for the Substitute attack (Section 3.2.6), and further degrade the security properties of shash.

```
FF,00,FF,00,FF,00,FF,00,FF,FF,FF,FF,FF,00,FF,00,
FF,00,FF,00,FF,00,FF,00,FF,FF,FF,FF,FF,00,FF,00,
FF,00,FF,00,FF,00,FF,00,FF,FF,FF,FF,FF,00,FF,00,
FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF,FF
```

Figure 10: Bitmask of the positions in shash message blocks that effect the value of π.

Only 368 bits of the 512 bits in a message block have any effect on π at all. The bits upon which the value of π is dependent occupy fixed positions regardless of the value of the message block. In Figure 10, we mark with F the nibbles (4-bit chunks) that are used to calculate the value of π.

To produce two messages blocks that both generate the same value of π, we just force both message blocks to have the same values for the nibbles that the value of π depends on. Hence, for a particular message block, we can generate $2^{144} - 1$ other message blocks that share the same π.

A closer analysis of how π is computed allows us to produce even more π colliding message blocks. The function SwapControl1 produces half as many permutations of π as the other SwapControl functions. This is because SwapControl1 marshals all of its input nibbles into pairs, xors these pairs together, and then uses the results of the xor to permute π. Given some three nibbles, n_1, n_2, and n_r where $n_1 \oplus n_2 = n_r$ we can easily find $2^4 - 1$ pair of nibbles, n_3 and n_4, that also xor to n_r. Incorporating this technique of replacing the input nibbles that SwapControl1 uses with other input nibbles that cause the same effect we can produce 2^{216} message blocks that generate the same π.

Producing message blocks that generate the same π creates some interesting implications for the security of shash beyond the collision attacks presented. For example if we fix p and find a set, S, of 512 message chunks that generate linearly independent values of h, we can describe all possible values of h as some subset of the set S xored together.

5.3 Implementing the Attack

We report on the collisions we generated in the latest shash reference implementation[6] using a typical desktop computer. The computer used for the tests was a Q6600 intel quad core processor with 4 GB of RAM, running ubuntu linux 8.10. The attack code was written in C, compiled with gcc and python using the sage library and compiler. Finding a collision using this attack takes roughly 300 milliseconds. We choose to perform the Shuffle attack from Section 3.2.7, because it both created the shortest colliding messages and it was the most complex of our attacks.

5.3.1 The Creating the First Message

Our first step is to generate some message block m_a. In this attack we can use almost any message block for m_a, but the larger the order of π the larger the colliding messages. The attack, Shuffle (see Section 3.2.7), that we will be performing produces colliding message of length $|M|$, where

$$|M| = 512 \times (order(\pi) + 2) \text{ bits.}$$

Finding message blocks of low order is the most computationally intensive part in all of our attacks as it requires randomly generating a message block, and then computing the $order(\pi)$ of that message block. That being said the process is nearly instant on a desktop computer. The reduced permutation method shown in Section 4.3 generates a message block having an $order(\pi) \leq 1000$) every 263 milliseconds. We used this method to generate the message block m_a as shown in Figure 11.

```
5B,41,7B,11,3B,4D,6B,70,5B,B1,7B,C5,0B,0F,2B,1F,
9B,CB,4B,54,5B,A3,7B,E4,3B,81,6B,35,6B,5C,3B,FC,
8B,09,EB,80,EB,1D,8B,36,5B,B1,7B,C5,EB,83,8B,04,
3B,B8,6B,D3,1B,D5,BB,A7,3B,81,6B,35,8B,66,EB,49
```

Figure 11: The message block m_a.

The second step is to compute some message chunk, C_a, such that the chaining variable p cycles. The order of the permutation π_a produced by m_a is 12, so we repeat m_a 12 times to produce C_a.

$$C_a = m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a = ||m_a||^{12}$$

The message chunk C_a is not a cyclic message chunk as only p cycles; both chaining variables must cycle in a cyclic message chunk.

The third step is to compute a second message block m_b which produces a permutation π_b such that $\pi_a = \pi_b$. We use our method, in Section 5.2, to generate a message block m_b (shown in Figure 12) that produces a colliding value of π

```
5B,5C,7B,9A,EB,48,8B,7B,5B,B1,7B,C5,0B,08,2B,CA,
5B,2B,7B,60,DB,1B,AB,3A,3B,81,6B,35,BB,8E,1B,09,
AB,36,DB,A4,4B,DF,9B,CE,5B,B1,7B,C5,4B,F6,9B,51,
3B,B8,6B,D3,1B,D5,BB,A7,3B,81,6B,35,8B,66,EB,49
```

Figure 12: The message block m_b.

Finally we append C_a and m_b to the end of our arbitrary chosen message block m_{init} (see Section 5.1) to create our first message M_{ab}.

$$M_{ab} = m_{init}||C_a||m_b$$

[6]There is some confusion about which reference implementation is the latest. The Spectral Hash website `http://cs.ucsb.edu/~koc/shash/Jan_2009_SHash.zip` provides the updated reference implementation. The NIST site `http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html` also makes available an updated implementation, that differs slightly. We were not able to determine which implementation is the most correct/update to date so to ensure applicability the attack was run against both implementations, with identical results.

5.3.2 The Creating the Second Message

We create the second message M_{ba} from the first message M_{ab} by switching the second message block, m_a, with the last message block, m_b.

$$M_{ab} = m_{init}||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_b$$
$$M_{ba} = m_{init}||m_b||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a||m_a$$
$$M_{ba} = m_{init}||m_b||C_a$$

5.3.3 The Collision

The Messages M_{ab} and M_{ba} produce the same hash value. We explain why in two stages, first we show that the chaining variable p is not altered, and second we show that because C_a causes p to collide at the second and last position, switching the message blocks at the second and last position does not produce a different final value of h.

Since the message block m_b produces the same value of π as m_a, the chaining variable p has no way of distinguishing between the message blocks m_a and m_b. Therefore m_a can be swapped with m_b without altering the value of the p.

Because the value of p cycles back to the value p_2, the value p_2 is supplied to the compression function for the message block in the second and the message block in the last position.

$$p_{ab} = p_{IV}, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_2$$
$$p_{ba} = p_{IV}, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_2$$

The function g is generated from p, so the cycle in p causes a cycle in value of g, thus generating same the function g_2 for both positions (see Section 3.2.2 for more details).

$$g_2 \leftarrow d(\pi_a(p_2))$$

As the second and last position use the same function g_2, the message block m_a will produce the same value $m'_a = g(m_a)$ in both positions. For the same reason, the message block m_b will also produce the same $m'_b = g(m_b)$ in both positions. The xor operation is commutative so the chaining variable h has no way of preserving the order of the values m' xored into it, thus h encodes the same value for

$$h_{ab} = h_{IV} \oplus m'_{init} \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_b$$

and

$$h_{ba} = h_{IV} \oplus m'_{init} \oplus m'_b \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a \oplus m'_a.$$

That is to say, the final value of the chaining variable h collides for both messages.

$$h_{ab} = h_{ba}$$

Since the chaining variables, h and p, collide for both messages and both messages are the same length, both messages must produce the same hash value for all hash sizes of the hash function.

$$shash(M_{ab}) = shash(M_{ba})$$

See Appendix Section A.1 for the complete colliding messages and digests.

6 Weakness in the Affine Transform Function

$0 \rightarrow 1,$	$4 \rightarrow 5,$	$8 \rightarrow 2,$	$12 \rightarrow 4,$
$1 \rightarrow 13,$	$5 \rightarrow 14,$	$9 \rightarrow 15,$	$13 \rightarrow 6,$
$2 \rightarrow 3,$	$6 \rightarrow 12,$	$10 \rightarrow 8,$	$14 \rightarrow 10,$
$3 \rightarrow 9,$	$7 \rightarrow 0,$	$11 \rightarrow 7,$	$15 \rightarrow 11,$

Figure 13: The inverse s-Box $AT^{-1}(m)$.

We present a weakness we discovered in the affine transform function used in shash. This weakness is not relevant to the collision attack. It is a completely independent result. Shash uses an affine transform function (AT) in the field $GF(2^4)$ to defend against differential and linear cryptanalysis. The affine transform is implemented as an s-box which is used to transform the message block.

AES uses an s-box in the field $GF(2^8)$ as a protection against linear and differential attacks. The authors of shash argue that since the s-box used in shash is similar to the AES s-box it should provide similar protection. Unfortunately, the shash s-box contains a critical flaw.

We will show that shash gains no additional protection from the s-box. We define shashNoBox as shash with the affine transform s-box removed. We want to find some input M and M' for which

$$shash(M) = shashNoBox(M').$$

To map from M' to M, we merely use the inverse s-box $AF^{-1}(M)$ shown in Figure 13. The s-box AT and the inverse s-box AT^{-1} cancel out, allowing us to map inputs and outputs of shashNoBox to the inputs and outputs of shash.

$$AT^{-1}(AT(M)) = M$$
$$shash(AT^{-1}(M')) = shashNoBox(M')$$

Consider the case in which an attacker has found a collision or preimage in the shashNoBox hash function. To translate the attack to shash, she merely has to run the inverse s-box on the colliding input or preimage. For instance, two messages M_a and M_b, known to collide in shashNoBox, can be mapped using AT^{-1} to generate two messages which must collide in shash.

$$M_a \neq M_b$$
$$shashNoBox(M_a) = shashNoBox(M_b)$$
$$shash(AT^{-1}(M_a)) = shash(AT^{-1}(M_b))$$

That is, shash is only as secure as shashNoBox. Therefore, the AT s-box as used in shash provides no additional cryptographic strength. AES does not have this weakness, because AES first xors the round key with the plaintext before applying the s-box[10]. Inverting the AES s-box would involve breaking/guessing the round key. Thus, the AES s-box is protected by the round key.

In AES, the s-box is applied each round, allowing the AES s-box many opportunities to mix and confuse the data[11]. Shash only applies the s-box once per message block, which results in much less substitution and a significantly weaker scheme. Therefore, even if the first issue was resolved by shielding the s-box behind some transformations or permutations, the second flaw prevents the s-box from offering comparable protection to the AES s-box.

While no linear or differential attacks have been demonstrated against shash, our results show that the affine transform function does not offer the protections, against linear and differential attacks, intended by the authors of shash.

7 Summary

In this paper we describe a theory of attack against the PTX family of compression functions. We show that shash uses a PTX compression function and is therefore vulnerable to our attacks. We present our research on exploiting this vulnerability to engineer collisions in the shash algorithm.

The fact that the pipe widening strategy [5] employed by shash provided no protection against chaining variable collisions, which according to the authors of shash is exactly the type of attack it was employed to foil, raises questions about the efficacy of pipe widening as a general defense against chaining variable collisions.

Furthermore, we show that any modifications of shash, even ones which we allow access to random oracles, are vulnerable as long as the compression function remains a PTX function. Thus, any attempt to strengthen shash against this attack must show that the improved compression function can not be reformulated as a PTX function. This generality, also opens the door to finding other hash functions that may use PTX compression functions.

We demonstrate a working example of our attack against the shash implementation. Our practical example shows that an attacker, using a typical desktop computer, can completely break strong collision resistance for all hash sizes in less than a second.

References

[1] G. Saldamlı, C. Demirkıran, M. Maguire, C. Minden, J. Topper, A. Troesch, C. Walker, Ç. K. Koç. Spectral hash. Submission to NIST, 2008. `http://www.cs.ucsb.edu/~koc/shash/sHash.pdf`.

[2] R.C. Merkle. A Certified Digital Signature. *Advances in Cryptology - CRYPTO '89 Proceedings*, 435(G):218–238, 1989.

[3] T. E. Bjørstad. Collision for the Reference Implementation of SpectralHash. NIST mailing list (local link), 2008. `http://ehash.iaik.tugraz.at/uploads/6/64/Spectralhash.txt`.

[4] I. Damgård. A Design Principle for Hash Functions. *Advances in Cryptology - CRYPTO '89 Proceedings*, 435(G):416–427, 1989.

[5] Stefan Lucks. Design principles for iterated hash functions. Cryptology ePrint Archive, Report 2004/253, 2004. `http://eprint.iacr.org/`.

[6] John A Beachy, William D. Blair. *ABSTRACT ALGEBRA*. Waveland press inc, Long Grove, IL, 2006.

[7] N. J. A. Sloane, Ed. The On-Line Encyclopedia of Integer Sequences, 2008. `http://www.research.att.com/~njas/sequences/A000793`.

[8] GOH W. M. Y. ; SCHMUTZ E. The expected order of a random permutation. *Bulletin of the London Mathematical Society*, 23:34–42, 1991.

[9] Richard Strong. The average order of a permutation. *Bulletin of the London Mathematical Society*, 5:10, 1998.

[10] Information Technology Laboratory (National Institute of Standards and Technology). *Announcing the Advanced Encryption Standard (AES) [electronic resource]*. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, 2001.

[11] Hans Dobbertin, Lars Knudsen, and Matt Robshaw. *The Cryptanalysis of the AES - A Brief Survey.* Springer Berlin / Heidelberg, 2005. `http://dx.doi.org/10.1007/11506447\_1`.

A Appendix

A.1 Collision

Message Digest (m1, m2):

```
224 =  80D3207C5F7E07C1299AA8254F74292BE7F86AE19FE0A72255BDB25A
256 =  80D3341789F7D03F02498D6A24B4DBB0926BD3DE2EB7873FE055C84D5DEACCB6
384 =  61032DCEC84FE19BF2F29077C0599C69255984A2E67BDE6C1219D5FADFB785B6ABC0F0FFF8095A7
       2093A33AFB4D296AD
512 =  D438099A673D9047760F33E6DBC5C41E3700E66369B2D15BCE1958AC73F72F9DE0A44E7A6FF5AFF
       63E0BDB5AF80F49EFBF102ABA1E50AA7A8E3A7F64FA49D52D
```

```
m1 =

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B5C7B9AEB488B7B5BB17BC50B082BCA5B2B7B60DB1BAB3A3B816B35BB8E1B09AB36DBA44BDF9BCE5BB17BC54BF69B513BB86BD31BD5BBA73B816B358B66EB49
```

```
Len = 7168

m2 =

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
5B5C7B9AEB488B7B5BB17BC50B082BCA5B2B7B60DB1BAB3A3B816B35BB8E1B09AB36DBA44BDF9BCE5BB17BC54BF69B513BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49
5B417B113B4D6B705BB17BC50B0F2B1F9BCB4B545BA37BE43B816B356B5C3BFC8B09EB80EB1D8B365BB17BC5EB838B043BB86BD31BD5BBA73B816B358B66EB49

Len = 7168
```

A.2 Equation of Reduced Permutation Message Blocks

$$
\begin{array}{llll}
AT^{-1}(& & & \\
n_0 \in \{0,1\}^4, & n_{32} \in \{0,1\}^4, & n_{64} \in \{0,1\}^4, & n_{96} \in \{0,1\}^4, \\
n_1 = 15, & n_{33} = 15 & n_{65} = 15, & n_{97} = 15, \\
n_2 \in \{0,1\}^4, & n_{34} \in \{0,1\}^4, & n_{66} \in \{0,1\}^4, & n_{98} \in \{0,1\}^4, \\
n_3 \in \{0,1\}^4, & n_{35} \in \{0,1\}^4, & n_{67} \in \{0,1\}^4, & n_{99} \in \{0,1\}^4, \\
n_4 = n_0 \oplus 15 & n_{36} = n_{32} \oplus 15, & n_{68} = n_{64} \oplus 15, & n_{100} = n_{96} \oplus 15, \\
n_5 = 15, & n_{37} = 15, & n_{69} = 15, & n_{101} = 15, \\
n_6 \in \{0,1\}^4, & n_{38} \in \{0,1\}^4, & n_{70} \in \{0,1\}^4, & n_{102} \in \{0,1\}^4, \\
n_7 \in \{0,1\}^4, & n_{39} \in \{0,1\}^4, & n_{71} \in \{0,1\}^4, & n_{103} \in \{0,1\}^4, \\
n_8 \in \{0,1\}^4, & n_{40} \in \{0,1\}^4, & n_{72} \in \{0,1\}^4, & n_{104} \in \{0,1\}^4, \\
n_9 \in= 15, & n_{41} = 15, & n_{73} = 15, & n_{105} = 15, \\
n_{10} \in \{0,1\}^4, & n_{42} \in \{0,1\}^4, & n_{74} \in \{0,1\}^4, & n_{106} \in \{0,1\}^4, \\
n_{11} \in \{0,1\}^4, & n_{43} \in \{0,1\}^4, & n_{75} \in \{0,1\}^4, & n_{107} \in \{0,1\}^4, \\
n_{12} = n_8 \oplus 15, & n_{44} = n_{40} \oplus 15, & n_{76} = n_{72} \oplus 15, & n_{108} = n_{104} \oplus 15, \\
n_{13} = 15, & n_{45} = 15, & n_{77} = 15, & n_{109} = 15, \\
n_{14} \in \{0,1\}^4, & n_{46} \in \{0,1\}^4, & n_{78} \in \{0,1\}^4, & n_{110} \in \{0,1\}^4, \\
n_{15} \in \{0,1\}^4, & n_{47} \in \{0,1\}^4, & n_{79} \in \{0,1\}^4, & n_{111} \in \{0,1\}^4, \\
n_{16} = 4, & n_{48} = 2, & n_{80} = 4, & n_{112} = 2, \\
n_{17} = 15, & n_{49} = 15, & n_{81} = 15, & n_{113} = 15, \\
n_{18} = 15 & n_{50} = 10, & n_{82} = 52, & n_{114} = 10, \\
n_{19} = 0, & n_{51} = 0, & n_{83} = 0, & n_{115} = 0, \\
n_{20} = 11, & n_{52} = 13, & n_{84} = 11, & n_{116} = 13, \\
n_{21} = 15, & n_{53} = 15, & n_{85} = 15, & n_{117} = 15, \\
n_{22} = 6, & n_{54} = 2, & n_{86} = 6, & n_{118} = 2, \\
n_{23} = 4, & n_{55} = 4, & n_{87} = 4, & n_{119} = 4, \\
n_{24} \in \{0,1\}^4, & n_{56} \in \{0,1\}^4, & n_{88} \in \{0,1\}^4, & n_{120} \in \{0,1\}^4, \\
n_{25} = 15, & n_{57} = 15, & n_{89} = 15, & n_{121} = 15, \\
n_{26} \in \{0,1\}^4, & n_{58} \in \{0,1\}^4, & n_{90} \in \{0,1\}^4, & n_{122} \in \{0,1\}^4, \\
n_{27} \in \{0,1\}^4, & n_{59} \in \{0,1\}^4, & n_{91} \in \{0,1\}^4, & n_{123} \in \{0,1\}^4, \\
n_{28} = n_{24} \oplus 15, & n_{60} = n_{56} \oplus 15, & n_{92} = n_{88} \oplus 15 & n_{124} = n_{120} \oplus 15, \\
n_{29} = 15, & n_{61} = 15, & n_{93} = 15, & n_{125} = 15, \\
n_{30} \in \{0,1\}^4, & n_{62} \in \{0,1\}^4, & n_{94} \in \{0,1\}^4, & n_{126} \in \{0,1\}^4, \\
n_{31} \in \{0,1\}^4, & n_{63} \in \{0,1\}^4, & n_{95} \in \{0,1\}^4, & n_{127} \in \{0,1\}^4, \\
) & & &
\end{array}
$$

The above equation is a solution for the values that a message block must possess to ensure that SwapControl1 and SwapControl2 perform no permutations, and SwapControl3 performs only limited permutations. We use this equation in Section 4.3. The variables n_0 through n_{127} represent nibbles in a message block. Note that the all the nibbles are transformed by the inverse affine transform function $AT^{-1}(m)$ from Section 6.