The Hash Function “Fugue”

Shai Halevi William E. Hall Charanjit S. Jutla
IBM T.J. Watson Research Center

September 15, 2009

Abstract

We describe Fugue, a hash function supporting inputs of length upto $2^{64} - 1$ bits and hash outputs of length upto 512 bits. Notably, Fugue is *not* based on a compression function. Rather, it is directly a hash function that supports variable-length inputs.

The starting point for Fugue is the hash function Grindahl, but it extends that design to protect against the kind of attacks that were developed for Grindahl, as well as earlier hash functions like SHA-1. A key enhancement is the design of a much stronger round function which replaces the AES round function of Grindahl, using better codes (over longer words) than the AES 4×4 MDS matrix. Also, Fugue makes judicious use of this new round function on a much larger internal state.

The design of Fugue is proof-oriented: the various components are designed in such a way as to allow proofs of security, and yet be efficient to implement. As a result, we can prove that current attack methods cannot find collisions in Fugue any faster than the trivial birthday attack. Although the proof is computer assisted, the assistance is limited to computing ranks of various matrices.

Contents

1 Introduction

Most hash functions to date are designed using the Merkle-Damgard paradigm [14, 6]: one first designs a compression function for fixed-length messages, and then extends it to variable-length messages by iterating the compression function for every block of the input.

This style of design has some drawbacks however: The most obvious problem is the message extension property of Merkle-Damgard, and over the years we have seen many other "generic" attacks, such as Joux's multi-collisions attack [8], Herding attacks [12], etc. At the heart of all these attacks lies the fact that the Merkle-Damgard paradigm is "wasteful" by design: having worked hard on hashing one block of the message, the design throws out all the internal state that was developed and leaves only the compressed output as a chaining variable for the next iteration.

A different approach for designing hash functions is to keep a larger evolving internal state, and to insert a "block" of the message into the state per iteration while applying a round function, and then keeping the entire state. After the whole message is processed, an extensive final transformation is applied to the state, and finally a part of the end state is used as the output.

One example of this style of design is the Grindahl hash function by Knudsen et al. [13]. For example, Grindahl-256 maintains an internal state in the form of a 4×13 matrix of bytes, and for each four-byte word of the input it first uses the input word to replace one of the columns, and then applies a transformation to the entire state. The transformation in Grindahl consists of essentially applying the round function of AES to the 13-column state. By relying on the AES round function, Grindahl inherits the efficiency and implementation flexibility of AES. Also the significant cryptanalytical effort that was invested in AES and its round function can be brought to bear on the security of Grindahl.

However, there are significant differences between block ciphers and hash functions, and the latter admit many new types of attacks. Indeed, many new types of attacks were introduced in the last few years for hash functions, including "message modification" [22, 23], "control/neutral bits" [3], "auxiliary differentials" [9], etc.. In particular, recent cryptanalysis of Grindahl due to Peyrin [20] suggests that the security of Grindahl is not as high as intended.

In Fugue we adopted the Grindahl approach of maintaining a large evolving state and using AES-like primitives to evolve it. To counter the risk of having to consider all the new types of attacks, we adopted in Fugue a proof-based approach to its design. That is, we designed the various components of Fugue to allow proofs of security. The main tools that we use in Fugue are the following:

- A larger state. For example, Fugue-256 uses a 4×30 matrix for its internal state.
- A souped-up variant of the AES round function [16, 5]. We still use the byte-substitution of AES, but we replace the linear Column-Mix with a Super-Mix operation. Recall that column-MIX operates on just one column. The Super-Mix operation, while being similar to column-MIX, also manages to affect the other three columns with little extra work (see Fig 4 in Section 1.5 and also see Section 4.2). This operation can be viewed as multiplying

a 16-byte vector by a 16×16 matrix, and we show that this matrix induces a number of MDS or nearly-MDS codes.

- Judicious use of the round function. In Fugue we do not apply the round function uniformly to the entire state. Rather, in each iteration we only apply the round function to select areas of the state, using cheap XOR operations to mix some other parts of the state, and yet other parts are left alone and not modified at all. We used a proof-guided approach to the decision of where to apply the round function and what parts of the state to mix using XORs. That is, we applied whatever operation was needed for the security claims to be *provable*.

As a result, Fugue is not only very secure, but quite efficient as well. The performance of Fugue-256 on contemporary machines is on par with SHA-256 (but Fugue can utilize parallelism much better than SHA-256). At the same time, we can prove that current attack techniques (including the techniques of Peyrin) do not apply to Fugue. Namely, we prove that using differential cryptanalysis with message modification and control/neutral byte analysis cannot be used to find collisions in Fugue any faster than a trivial birthday attack. We emphasize here that the proof is not based on ruling out certain possible attack scenarios by brute force search, but by a rigorous proof encompassing a whole range of strategies.

1.1 Speed estimates

The estimated (or measured) performance of Fugue-256 on 32-bit platforms, 64-bit platforms, 8-bit platforms and hardware platforms are given in Part II of this document (Page 29). In all cases, the performance of Fugue-256 is on par with that of SHA-256. The speed of Fugue-224 is the same as Fugue-256, while the speed of Fugue-348 is about 66% of Fugue-256, and the speed of Fugue-512 is about 50% of Fugue-256. The speed of a weak variant of Fugue-256, called wFugue-256, is about twice that of Fugue-256.

1.2 Statement of Expected Strength

We expect that the best attacks against Fugue are the generic ones. That is, the best collision attack against Fugue-X will have work factor of $2^{X/2}$, and the best pre-image and second-pre-image attacks will have work factor of 2^X. Also, we expect that the best distinguishing attack against PR-Fugue-X will have work factor 2^X. A detailed analysis of the security of Fugue is provided in Part III of this document (Section 7).

As for the weak version wFugue-256, our analysis suggests that limited-space attacks (say, upto 2^{96} bits) cannot find collisions in wFugue-256 with probability better than 2^{-96}. Here too we expect that the best attack has work factor of 2^{128}, but as opposed to Fugue-256 we cannot prove this work factor for wFugue-256.

1.3 Statement of Advantages and Limitations

Fugue inherits the flexible implementation profile of AES, so we expect that it can be implemented in many different environments. It can also be used with essentially any output size of upto 512 bits (in multiples of 32 bits), although in this document we only specified the required output sizes of 224, 256, 384 and 512. Another advantage of Fugue is the ability to share sub-components with AES (since we are using the same S-box).

But the biggest advantage of Fugue is its security. As we said before, Fugue comes with a proof of security, showing that current attack techniques cannot be used against it.

1.4 Main Idea

Since there is no secret key in a hash construct, in order to achieve a collision, an adversary can follow the development of the state for two different inputs and stop early, or backtrack and modify the inputs adaptively. In fact, since the initial state is fixed, it might even be able to pre-compute a strategy for this backtracking in a table. Further, since not all of the input can be consumed into the state at one time, the input is incorporated into the state incrementally, which gives the adversary another adaptive strategy to choose the later inputs as a function of the current state(s).

The problem of starting from a fixed initial state is much easier to handle, as from a hash function design perspective one could make the state transformations non-linear enough, so that such initial strategies either require huge tables, say of size greater than 2^{96} bits, or are computationally in-effective, beyond a few rounds of initial input blocks.

However, the second problem of countering a multi-input-block adaptive collision attack is more difficult, as partial collisions obtained in the previous round can be used to adaptively choose inputs in the next round to achieve even better partial collisions, till one actually finds a full collision.

In the design of Fugue, our aim is to prove that no matter how good a partial collision is obtained, finding a full collision, even with adaptive input control, is an extremely low probability event. A key design feature is a new maximum distance separable (MDS) linear code, which is extremely efficient to encode. Recall that even AES's mixing is based on an MDS code, though the length of that code is only 8 bytes (in standard form it has 4 message bytes, and 4 parity bytes), and its minimum distance is 5. Our MDS code has length 16 bytes, and has minimum distance 13.

Before we explain how these codes are used, we give a simple overview of Fugue-256 in Fig 1. The internal state is of size 30 words (each word being 32 bits). Further, it has two types of rounds: the *input* rounds, and the *final round*. In each input round, only one word of input is consumed. After all the inputs are iteratively consumed, a *final* extensive round of non-linear transformations is performed, before outputting a subset of the state as the hash value.

Thus, one can obtain a collision either by finding a collision in the 30 word state itself after some input rounds, in which case it is called an *internal collision*, or by finding a collision in

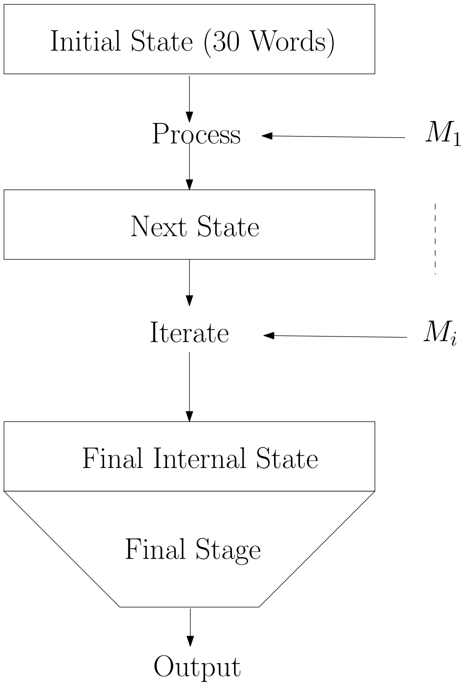

Figure 1: Fugue-256 Structure

the subset of the state that is output at the end of the final round, in which case it is called an *external collision*.

In this introduction, we will mainly focus on internal collisions, as the choice of input words in each input round may allow one to converge on an internal collision, and hence it may seem as a simpler problem than finding external collisions. Further, we will consider a simplified version of Fugue for the moment, so as to illustrate the collision resistance property of Fugue. This simplified version is shown in Fig 2, where we consider a differential trail, i.e. all quantities shown in the figure are differentials. A round begins by replacing the first column of the 30 column state by an input word. Next, the 30 column state is rotated to the right by three columns. After that, the processing focuses on just the left four columns, and just as in AES, replaces all 16 bytes using an (invertible, but non-linear) S-Box substitution. This is followed by an invertible linear transformation on these four columns over GF(2^8).

As can be seen in Fig 2(f), we are considering a round which led to an internal collision, i.e. all zero differences. This implies, and as is shown in the figure, that in state (e) the second, third and the fourth columns must have zero differences. Assume that in state (b) there are some non-zero differences. Then, since both the S-box and the linear transformation are invertible, it follows that in state (e), one of the Δb's must be non-zero. Thus, we have a situation that the linear transformation on the four left columns of state (d) led to one non-zero column and three zero columns in state (e). Consider an error-correcting linear code with 12 parity check equations, such that these equations are the ones which compute the 12 output bytes of the linear transformation corresponding to the right three columns. Then, the four left columns in state (d) form a *codeword* in this code.

The linear transformation we use in Fugue has the property that this code is an MDS code, and hence has minimum distance 13 (note that the dimension of the code is $16 - 12$, as it has 12 linearly independent parity check equations, and hence its maximum possible minimum distance is $16-(16-12)+1 = 13$). Thus, in state (d) there were at least 13 non-zero byte differences, which means that at least 13 S-Box substitutions in state (c) were on pairs of bytes whose differences were non-zero, and hence assuming that the S-Box had good differential properties, achieving internal collision has low probability.

Notice that, this implies that the linear transformation we use is not the same as the AES linear transformation (which is built out of AES's MixColumn and Row Rotations), as the AES linear transformation does not have this MDS property over 16 byte length codes. In the next sub-section we describe how our linear transformation is built and how it remains highly efficient.

One might wonder, if we can extend the above argument of using an MDS code to modify, say, eight columns of the state. If such an MDS code can be obtained, then that would imply a minimum distance of 32 - (32-28)+1 = 29. However, finding MDS codes of such large length, i.e. much more than 16 bytes, is a difficult task, especially since we want the codes to be extremely efficient to encode. However, what we do show is that for each $t \geq 0$, there is an efficient linear code of length $(4+3t)\times 4$, such that it has dimension four, and minimum distance $(4+3t)\times 4-(t+1)\times 3 = 13 + 8t$.

Before, we describe how this is accomplished, we remark that we use these codes to scale security against internal collisions, as well as external collisions, both within a fixed hash output length, and through different hash lengths. So, as an example, we consider the case $t = 1$. Thus, we want a code of length 28 bytes (or elements of GF(2^8)). We build this code by composition. Thus, we start with a linear transformation of 16 bytes to 16 bytes, given by a 16×16 matrix $\mathbf{N}$ of GF(2^8) elements, which transforms a 16 byte (column) vector u to u' by multiplication on the left, i.e.

$$u' = \mathbf{N} \cdot u$$

The subset of vectors u, which under the linear transformation $\mathbf{N}$ result in vectors with the last 12 bytes zero, form a linear code, whose parity check equations are given by the last 12 rows of $\mathbf{N}$. Suppose, this 12×16 sub-matrix has the property that all its 12×12 sub-matrices are non-singular, then a simple application of linear algebra shows that the linear code is MDS. Hence, every non-zero vector u in this subset can have at most three zero bytes.

If we start with vectors of 28 bytes, denoted $u[0..27]$, we first apply the linear transformation $\mathbf{N}$ to 16 bytes $u[12..27]$, say resulting in $u'[12.27]$. Next, we apply the same transformation $\mathbf{N}$ on 16 bytes $u[0..11]$, $u'[12..15]$, say resulting in $u''[0..15]$. Now consider the linear transformation $\overline{\mathbf{N}}$ which takes 28 bytes $u[0..27]$ to 28 bytes $u''[0..15]$, $u'[16..27]$. Then, $\overline{\mathbf{N}}$ has the desired property. First note that the transformation is invertible, and hence if its input is non-zero, then so is it's output. If we are considering non-zero inputs which lead to outputs with all but the first four bytes zero, then the first four bytes of the output must be non-zero, i.e. $u''[0..3]$ is non-zero, and $u''[4..15]$ and $u'[16..27]$ are zero. Thus, from the MDS property of $\mathbf{N}$, it follows that there are

$$\boxed{\begin{array}{llll c lllll} \Delta a_0 & 0 & 0 & 0 & & 0 & 0 & \Delta u_0 & \Delta v_0 & \Delta w_0 \\ \Delta a_1 & 0 & 0 & 0 & & 0 & 0 & \Delta u_1 & \Delta v_1 & \Delta w_1 \\ \Delta a_2 & 0 & 0 & 0 & \cdots & 0 & 0 & \Delta u_2 & \Delta v_2 & \Delta w_2 \\ \Delta a_3 & 0 & 0 & 0 & & 0 & 0 & \Delta u_3 & \Delta v_3 & \Delta w_3 \end{array}} \qquad \text{(a)}$$

Replace first column with input word

$$\boxed{\begin{array}{llll c lllll} \Delta x_0 & 0 & 0 & 0 & & 0 & 0 & \Delta u_0 & \Delta v_0 & \Delta w_0 \\ \Delta x_1 & 0 & 0 & 0 & & 0 & 0 & \Delta u_1 & \Delta v_1 & \Delta w_1 \\ \Delta x_2 & 0 & 0 & 0 & \cdots & 0 & 0 & \Delta u_2 & \Delta v_2 & \Delta w_2 \\ \Delta x_3 & 0 & 0 & 0 & & 0 & 0 & \Delta u_3 & \Delta v_3 & \Delta w_3 \end{array}} \qquad \text{(b)}$$

Rotate to the right by three columns

$$\boxed{\begin{array}{llll c lllll} \Delta u_0 & \Delta v_0 & \Delta w_0 & \Delta x_0 & & 0 & 0 & 0 & 0 & 0 \\ \Delta u_1 & \Delta v_1 & \Delta w_1 & \Delta x_1 & & 0 & 0 & 0 & 0 & 0 \\ \Delta u_2 & \Delta v_2 & \Delta w_2 & \Delta x_2 & \cdots & 0 & 0 & 0 & 0 & 0 \\ \Delta u_3 & \Delta v_3 & \Delta w_3 & \Delta x_3 & & 0 & 0 & 0 & 0 & 0 \end{array}} \qquad \text{(c)}$$

S-Box substitution on left 4 columns

$$\boxed{\begin{array}{llll c lllll} \Delta \tilde{u}_0 & \Delta \tilde{v}_0 & \Delta \tilde{w}_0 & \Delta \tilde{x}_0 & & 0 & 0 & 0 & 0 & 0 \\ \Delta \tilde{u}_1 & \Delta \tilde{v}_1 & \Delta \tilde{w}_1 & \Delta \tilde{x}_1 & & 0 & 0 & 0 & 0 & 0 \\ \Delta \tilde{u}_2 & \Delta \tilde{v}_2 & \Delta \tilde{w}_2 & \Delta \tilde{x}_2 & \cdots & 0 & 0 & 0 & 0 & 0 \\ \Delta \tilde{u}_3 & \Delta \tilde{v}_3 & \Delta \tilde{w}_3 & \Delta \tilde{x}_3 & & 0 & 0 & 0 & 0 & 0 \end{array}} \qquad \text{(d)}$$

linear transformation on left 4 columns

$$\boxed{\begin{array}{llll c lllll} \Delta b_0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 \\ \Delta b_1 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 \\ \Delta b_2 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\ \Delta b_3 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 \end{array}} \qquad \text{(e)}$$

Replace first column with input word

$$\boxed{\begin{array}{llll c lllll} 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 \end{array}} \qquad \text{(f)}$$

Figure 2: One Round of “Simplified Fugue” leading to an Internal Collision

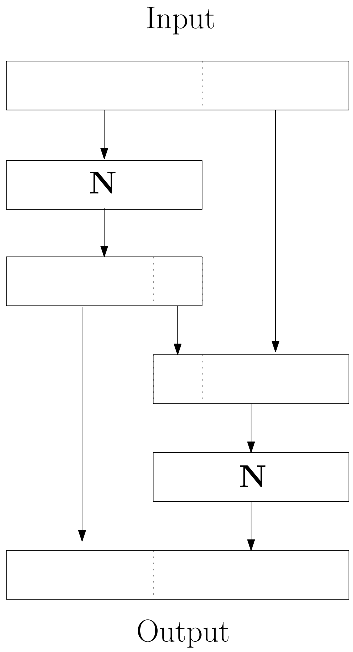

Figure 3: The composite linear transformation $\overline{\mathbf{N}}$

at most three bytes zero amongst $u[0..11]$ and $u'[12..15]$ combined. Thus, $u'[12..15]$ is non-zero. Moreover, $u'[16..27]$ is zero, and hence again by the MDS property of **N**, it follows that there are *at most three bytes zero* in $u[12..27]$. Thus, there are at most 6 bytes zero in $u[0..27]$. Thus, the linear code whose parity check equations are the $(28 - 4)$ rows of the 28×28 matrix representing the linear transformation $\overline{\mathbf{N}}$, has minimum distance $28 - 6 = 22$. The composite linear transformation is illustrated in Figure 3.

We now describe how an efficient linear transformation like **N** is obtained.

1.5 Super-Mix Transformation

If the linear transformation **N** is implemented by just the (left) matrix multiplication by **N**, it is likely to be extremely in-efficient, unless it is very sparse. Even if it is sparse, one may need to store $GF(2^8)$ multiplication tables corresponding to all the different entries in **N**. Finally, there remains the question of determining such a sparse matrix with the required MDS property. A brute force search over such matrices seems intractable.

Our approach starts with the AES mixing transformation [16], which is a linear transformation from vectors of four $GF(2^8)$ elements to vectors of four $GF(2^8)$ elements, and is given by a 4×4 matrix **M** over $GF(2^8)$. In fact, AES does indeed do a 16 byte to 16 byte linear transformation, but does so by just independently transforming groups of four bytes by **M**. Thus, this transformation is not even close to having the MDS property we require of **N**.

$$\begin{pmatrix} M_{00} & M_{10} & M_{20} & M_{30} \\ M_{01} & M_{11} & M_{21} & M_{31} \\ M_{02} & M_{12} & M_{22} & M_{32} \\ M_{03} & M_{13} & M_{23} & M_{33} \end{pmatrix} \otimes \begin{bmatrix} u_0 & v_0 & w_0 & x_0 \\ u_1 & v_1 & w_1 & x_1 \\ u_2 & v_2 & w_2 & x_2 \\ u_3 & v_3 & w_3 & x_3 \end{bmatrix} =$$

AES:

$$\left[\ \boxed{\begin{array}{l} T_0[u_0]+ \\ T_1[u_1]+ \\ T_2[u_2]+ \\ T_3[u_3] \end{array}}\ \boxed{\begin{array}{l} T_0[v_0]+ \\ T_1[v_1]+ \\ T_2[v_2]+ \\ T_3[v_3] \end{array}}\ \boxed{\begin{array}{l} T_0[w_0]+ \\ T_1[w_1]+ \\ T_2[w_2]+ \\ T_3[w_3] \end{array}}\ \boxed{\begin{array}{l} T_0[x_0]+ \\ T_1[x_1]+ \\ T_2[x_2]+ \\ T_3[x_3] \end{array}}\ \right]$$

Fugue (*hint*):

Fugue (*detail*):

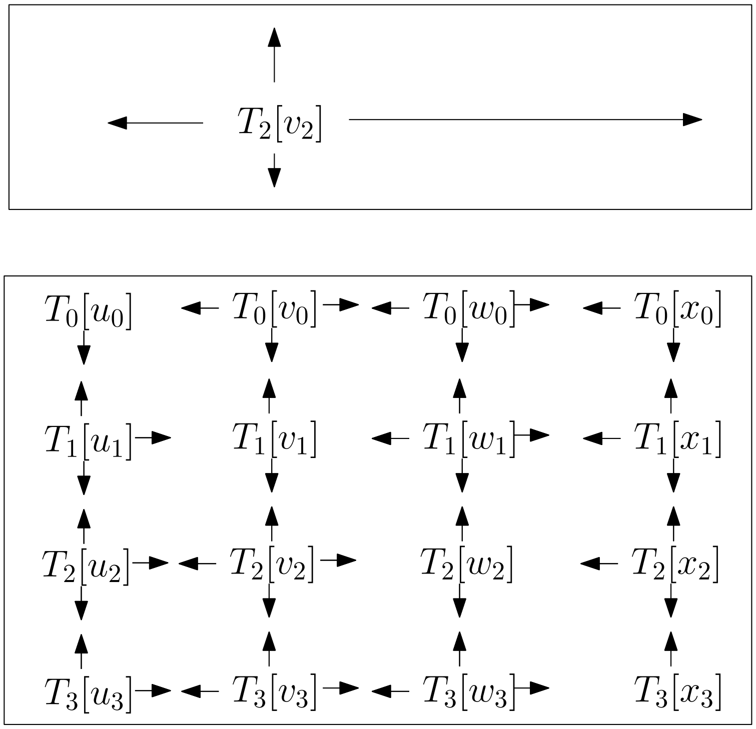

Figure 4: AES Column Mix vs. Fugue Super-Mix

The AES 4×4 transformation however has a nice property that it can be implemented with four 8-bit to 32-bit tables. Thus, let v be a four byte column vector $\langle v_0, v_1, v_2, v_3 \rangle$. Let $\mathbf{M}$'s four columns be called $\mathbf{M}_0$, $\mathbf{M}_1$, $\mathbf{M}_2$, and $\mathbf{M}_3$. Then,

$$\mathbf{M} \cdot v = v_0 \cdot \mathbf{M}_0 \; + \; v_1 \cdot \mathbf{M}_1 \; + \; v_2 \cdot \mathbf{M}_2 \; + \; v_3 \cdot \mathbf{M}_3$$

Thus, if we pre-compute a 32-bit value table T_0, with 256 entries indexed by i, corresponding to $i \cdot \mathbf{M}$, and similarly pre-compute T_1, T_2 and T_3, then

$$\mathbf{M} \cdot v = T_0[v_0] \; + \; T_1[v_1] \; + \; T_2[v_2] \; + \; T_3[v_3]$$

which is implemented with four table lookups and three 32-bit exclusive-or operations.

In Fugue, the 16 byte to 16 byte linear transformation is obtained as follows (see figure 4). The 16 bytes are first viewed as a 4×4 matrix of bytes $\mathbf{V}$, which is same as how the 16 byte state is represented in AES. In other words, the AES mixing transformation can be seen as a matrix multiplication, i.e. $\mathbf{V}' = \mathbf{M} \cdot \mathbf{V}$. However, as noticed above, that mixes the columns in $\mathbf{V}$ independently. Instead, in Fugue, the contribution of each byte in the input matrix $\mathbf{V}$ to the output matrix $\mathbf{V}'$ is to both its column, and its row (except for the diagonal entries, which continue to affect only their columns). Thus, using the alternate interpretation of the previous paragraph, the contribution of a byte v_2 in a column of $\mathbf{V}$ is not only $T_2[v_2]$ to its column in the output, but also $T_2[v_2]$ transposed, to its row in the output. The row in this case is row two, as v_2 is from row two of $\mathbf{V}$.

Although, one can derive some intuition of why this leads to good mixing, it is not true that all matrices $\mathbf{M}$ yield the 16 byte transformation $\mathbf{N}$ as desired, namely with the MDS property mentioned above. Indeed, the AES mixing matrix does not satisfy this property. However, a straightforward brute force search over all circulant 4×4 matrices yield many such matrices which result in an MDS $\mathbf{N}$. However, we picked the first one with some more desirable properties, as will become clear later. We later indicate in Section 16 the exact criterion used.

Acknowledgements. We would like to thank Hugo Krawczyk for several helpful discussions.

Part I

Specification of Fugue

2 Basic Conventions

Bits, Bytes, and Numbers. This document is mostly stated in terms of bytes, which are represented in hexadecimal notation (e.g., **80** is a byte corresponding to the decimal number 128). A sequence of bytes is always denoted with the first byte on the left, for example **00 01 80** is a sequence of three bytes, with the first being **00**, the second **01**, and the last **80**.

When considering bits, we follow the big-endian convention set by NIST and consider the most-significant bit as the first bit in a byte. (For example, the byte **80** represents a single 1 bit followed by seven 0 bits.) In a few places we need to talk also about multi-byte integers, and in these cases we also use big endian convention, namely the first byte in a representation of an integer is the most significant byte. (For example, a three-byte representation of the decimal number $384 = 256 + 128$ is the same sequence **00 01 80** from above.)

Matrices, Columns, and Words. Throughout this document we view the internal state of the hash function Fugue as a matrix with byte entries. We use the following notations for matrices: For an $m \times n$ matrix M, the rows are numbered 0 to $m-1$ (and displayed top to bottom), and the columns will be numbered 0 to $n-1$ (and displayed left to right). The i-th row of M is denoted M^i, and the j-th column of M is denoted M_j. The element in the i-th row and j-th column of M (which is a byte) is denoted M^i_j. A sub-sequence of columns of a matrix M, numbered i through j, will be denoted by $M_{i..j}$.

Very often in this document we view a column in a matrix as a fundamental unit, and we call it a *Word*. Since the state-matrices that we consider have four rows, then a word in this document is always a four-byte entity.

We view the bytes in a matrix in *column order*, where in each columns the bytes are ordered from top (index 0) to bottom (index $m-1$). For example, the following 4×3 matrix

$$A = \begin{pmatrix} \mathbf{00} & \mathbf{04} & \mathbf{08} \\ \mathbf{01} & \mathbf{05} & \mathbf{09} \\ \mathbf{02} & \mathbf{06} & \mathbf{0a} \\ \mathbf{03} & \mathbf{07} & \mathbf{0b} \end{pmatrix}$$

is represented as the sequence of bytes **00 01 02 03 04 05 06 07 08 09 0a 0b**, and its second column is the word $A_1 =$ **04 05 06 07**.

3 Galois Field GF(2^8)

The Galois Field GF(2^8), is the finite field of 256 elements, whose elements are represented as bytes and have one to one correspondence with degree-7 binary polynomials. The mapping from bytes to binary polynomials is given by considering the least significant bit of the byte as representing the free coefficient, and in general the i'th least-significant bit as the coefficient of x^i. For example, the byte **13** (corresponding to the bit sequence 00010011) represents the polynomial $x^4 + x + 1$, and the byte **3c** (binary 00111100) represents the polynomial $x^5 + x^4 + x^3 + x^2$.

The additive unity of the field is the zero polynomial, i.e. **00**, and the multiplicative unity of the field is the polynomial 1, i.e. **01**. Throughout this specification, the field multiplication will be polynomial multiplication modulo the irreducible polynomial

$$x^8 + x^4 + x^3 + x + 1.$$

As an example, when multiplying the byte corresponding to 8-bit binary number $a_7a_6...a_0$ (or the polynomial $a_7x^7 + a_6x^6 + ... + a_0$) by the byte **02** (i.e. the polynomial x), we get the polynomial $a_6x^7 + a_5x^6 + a_4x^5 + (a_3 \oplus a_7)x^4 + (a_2 \oplus a_7)x^3 + a_1x^2 + (a_0 \oplus a_7)x + a_7$, where $\oplus$ is the exclusive-or operation. (In other words, multiplying a byte $\mathbf{a}$ by **02** in GF(2^8), we get $2 \times \mathbf{a}$ when $\mathbf{a} < \mathbf{80}$, and $(2 \times \mathbf{a}) \oplus \mathbf{1b}$ otherwise.)

Below and throughout the document we use "$\cdot$" to denote field multiplication in GF(2^8), and "+" to denote field addition in GF(2^8)(which is the same as exclusive-or). The same symbols will be used to denote multiplication and addition of matrices or vectors over GF(2^8), as well as scalar multiplication of field elements with matrices or vectors over GF(2^8). As is common in many programming languages $\mathbf{x} += \mathbf{y}$ will denote the assignment $\mathbf{x} = \mathbf{x} + \mathbf{y}$.

4 Specification of Fugue-256

The main component in the hash function Fugue (called "SMIX" below) is a mapping from 16 bytes to 16 bytes, which resembles the round functions of the block cipher AES [16]. As in AES, it is sometimes convenient to view the 16 bytes as a 4×4 matrix (but we will also view them just as a column vector of 16 bytes). Also as in AES, the SMIX mapping is a permutation, consisting of byte-substitution followed by a linear transformation. (However, in Fugue we never need to compute the inverse map.)

4.1 Substitution Box

The substitution box (**S-box**) that is used in Fugue is identical to the one used in AES. The S-box is a non-linear permutation over bytes, which is composed of two mappings: The first map treats the 8-bit quantity as an element of GF(2^8) (as specified in section 3), and takes the multiplicative inverse if the element is non-zero, and otherwise just maps to **00**. The second map, treats the resulting GF(2^8) element as an 8-bit bit vector and performs the following affine transformation (over GF(2)).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

The S-box is explicitly given in the appendix in Table 12.

4.2 Super-Mix

Similarly to the AES round function, the SMIX transformation in Fugue takes a 4×4 matrix of bytes, passes each byte through the S-box transformation, and then applies a linear transformation to the result. This linear transformation is called the "Super-mix" transformation. A major difference between Fugue and AES is that in AES each column of the matrix is mixed separately (via the Column Mix transformation), whereas in Fugue there is cross-mixing between the column. As described later in this document, the Super-Mix in Fugue utilizes stronger codes (over longer words) than the 4-byte MDS code of AES, thus providing better diffusion and better protection against differential attacks (at a modest cost).

The linear transformation in Fugue is called the "Super-Mix". It can be viewed as putting the 16 bytes in one column vector and multiplying it by the 16×16 matrix N that is specified in Equation 3 below. To better understand this transformation, it helps to see how it can be built from a simpler 4×4 matrix (denoted $\mathbf{M}$). Like in AES, the matrix $\mathbf{M}$ will be a *circulant* matrix, but Fugue uses a different matrix than the 4×4 Column Mix matrix of AES. Specifically, we use

$$\mathbf{M} = \begin{pmatrix} \mathbf{01} & \mathbf{04} & \mathbf{07} & \mathbf{01} \\ \mathbf{01} & \mathbf{01} & \mathbf{04} & \mathbf{07} \\ \mathbf{07} & \mathbf{01} & \mathbf{01} & \mathbf{04} \\ \mathbf{04} & \mathbf{07} & \mathbf{01} & \mathbf{01} \end{pmatrix} \tag{1}$$

As a warm-up, consider the procedure that one would use to implement an AES-like Column Mix transformation, and later we explain how to modify this procedure to get the Super-Mix of Fugue. Let us denote the input 4×4 matrix by $\mathbf{U}$. Given the input matrix $\mathbf{U}$ and the mixing matrix $\mathbf{M}$, the AES Column Mix procedure is just $\mathbf{V} = \mathbf{M} \cdot \mathbf{U}$, i.e. a straightforward matrix multiplication over GF(2^8). Thus, the j-th column of $\mathbf{V}$ can be obtained from the j-th column of $\mathbf{U}$ as the sum

$$\mathbf{V}_j \;=\; \mathbf{U}_j^0 \cdot \mathbf{M}_0 \;+\; \mathbf{U}_j^1 \cdot \mathbf{M}_1 \;+\; \mathbf{U}_j^2 \cdot \mathbf{M}_2 \;+\; \mathbf{U}_j^3 \cdot \mathbf{M}_3 \;=\; \sum_{i=0}^{3} \mathbf{U}_j^i \cdot \mathbf{M}_i.$$

In Super-Mix, each of the terms $\mathbf{U}_j^i \cdot \mathbf{M}_i$ is not only added to the output column $\mathbf{V}_j$, but if $i \neq j$ then it is also transposed and added to the output row $\mathbf{V}^i$. Then (again similarly to AES), we apply a "row shift" operation to the matrix, in which the i-th row is rotated to the left by i positions. Namely, given a 4×4 input matrix $\mathbf{U}$ we compute:

$$\text{Super-Mix}(\mathbf{U}) = ROL\left(\mathbf{M}\cdot\mathbf{U}\ + \begin{pmatrix} \sum_{j\neq 0}\mathbf{U}_j^0 & 0 & 0 & 0 \\ 0 & \sum_{j\neq 1}\mathbf{U}_j^1 & 0 & 0 \\ 0 & 0 & \sum_{j\neq 2}\mathbf{U}_j^2 & 0 \\ 0 & 0 & 0 & \sum_{j\neq 3}\mathbf{U}_j^3 \end{pmatrix}\cdot\mathbf{M}^{\mathrm{T}}\right) \tag{2}$$

where

- $\mathbf{M}^{\mathrm{T}}$ is the transpose of the matrix $\mathbf{M}$, i.e. $(\mathbf{M}^{\mathrm{T}})_j^i = \mathbf{M}_i^j$,
- the transformation "ROL" takes a 4×4 matrix, and rotates the i-th row to the left by i bytes, i.e. $\mathrm{ROL}(\mathbf{W})_j^i = \mathbf{W}^i_{(j-i) \bmod 4}$

In other words, if $\mathbf{W}$ denotes the intermediate matrix before the "ROL" transformation in Super-Mix($\mathbf{U}$), then

$$\mathbf{W}_j^i = \sum_k (\mathbf{M}_k^i \cdot \mathbf{U}_j^k) + \mathbf{M}_i^j \cdot (\sum_{k\in[0..3], k\neq i} \mathbf{U}_k^i)$$

Equivalently, the Super-Mix transformation is given by (left) multiplication by the following 16×16 matrix $\mathbf{N}$, when the 4×4 matrix of input (and output) is considered as a 16-byte column vector, with the $(i + 4j)$-th byte of the vector corresponding to the byte in the i-th row and the j-th column of the matrix (i.e. the input and output matrices are scanned column-wise).

$$\mathbf{N} = \begin{pmatrix}
1 & 4 & 7 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 4 & 7 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 7 & 1 & 1 & 4 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 4 & 7 & 1 & 1 \\[2ex]
0 & 0 & 0 & 0 & 0 & 4 & 7 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 4 & 7 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 7 & 1 & 0 & 4 \\
4 & 7 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\[2ex]
0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 6 & 4 & 7 & 1 & 7 & 0 & 0 & 0 \\
0 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 1 & 6 & 4 & 7 \\
7 & 1 & 6 & 4 & 0 & 0 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 \\
0 & 0 & 0 & 7 & 4 & 7 & 1 & 6 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\[2ex]
0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 5 & 4 & 7 & 1 \\
1 & 5 & 4 & 7 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 4 & 0 & 0 \\
0 & 0 & 4 & 0 & 7 & 1 & 5 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 \\
0 & 0 & 0 & 4 & 0 & 0 & 0 & 4 & 4 & 7 & 1 & 5 & 0 & 0 & 0 & 0
\end{pmatrix} \tag{3}$$

$$\mathbf{N}^{-1} = \left(\begin{array}{cccccccccccccccc}
15 & 49 & 16 & bc & bf & ca & 76 & f5 & a5 & 2f & 2b & 57 & b2 & 3c & 3d & 45 \\
9c & be & 59 & df & 56 & a9 & de & fe & 26 & f5 & 8c & fb & 77 & c0 & 2a & 39 \\
e2 & 15 & 45 & 16 & 96 & 1f & a6 & 3b & 4d & f7 & f4 & 91 & 4b & f4 & a1 & c1 \\
5e & fd & 69 & 6f & ca & 2a & 3e & f6 & 9f & a7 & c6 & 5b & cf & cf & 62 & 31 \\[1em]
6f & 5e & fd & 69 & f6 & ca & 2a & 3e & 5b & 9f & a7 & c6 & 31 & cf & cf & 62 \\
bc & 15 & 49 & 16 & f5 & bf & ca & 76 & 57 & a5 & 2f & 2b & 45 & b2 & 3c & 3d \\
df & 9c & be & 59 & fe & 56 & a9 & de & fb & 26 & f5 & 8c & 39 & 77 & c0 & 2a \\
16 & e2 & 15 & 45 & 3b & 96 & 1f & a6 & 91 & 4d & f7 & f4 & c1 & 4b & f4 & a1 \\[1em]
45 & 16 & e2 & 15 & a6 & 3b & 96 & 1f & f4 & 91 & 4d & f7 & a1 & c1 & 4b & f4 \\
69 & 6f & 5e & fd & 3e & f6 & ca & 2a & c6 & 5b & 9f & a7 & 62 & 31 & cf & cf \\
16 & bc & 15 & 49 & 76 & f5 & bf & ca & 2b & 57 & a5 & 2f & 3d & 45 & b2 & 3c \\
59 & df & 9c & be & de & fe & 56 & a9 & 8c & fb & 26 & f5 & 2a & 39 & 77 & c0 \\[1em]
be & 59 & df & 9c & a9 & de & fe & 56 & f5 & 8c & fb & 26 & c0 & 2a & 39 & 77 \\
15 & 45 & 16 & e2 & 1f & a6 & 3b & 96 & f7 & f4 & 91 & 4d & f4 & a1 & c1 & 4b \\
fd & 69 & 6f & 5e & 2a & 3e & f6 & ca & a7 & c6 & 5b & 9f & cf & 62 & 31 & cf \\
49 & 16 & bc & 15 & ca & 76 & f5 & bf & 2f & 2b & 57 & a5 & 3c & 3d & 45 & b2
\end{array}\right)$$

Figure 5: The matrix N^{-1}

The Super-Mix transform is invertible, as verified by a computer program that yielded the inverse for **N** as described in Figure 5.

4.3 The Hash Function F-256

In this section we specify the hash function **F**-256 that underlies Fugue-256. The function **F**-256 takes as input a byte string of length multiple of four bytes, and an initial vector (IV) of 32 bytes, and outputs a hash value of 32 bytes. The hash function **F**-256 maintains a **state** of 30 four byte columns, starting with an initial state, which is set using the IV.

The input stream of $4m$ bytes ($m \geq 0$) is parsed as m four-byte words, and fed one word at a time into a *round transformation* **R** that modifies the state. After all the input has been processed, the state undergoes another transformation by a *final round* **G**. Subsequently, eight columns of the state are used as the output of **F**-256.

4.3.1 The Round Transformation R

The round transformation **R** takes a 30 column state **S**, and one four-byte word $I = I^0 I^1 I^2 I^3$, and produces a new 30 column state. Using the notation specified in Section 2, the 30 column state can be identified with a 4×30 matrix (e.g. the first column of the state being $\mathbf{S}_0$, etc.).

The transformation $\mathbf{R}$ is best specified by the way it modifies the state $\mathbf{S}$ using the input word I. It consists of the following sequence of steps (all of which are described below):

$$\mathbf{TIX}(I); \mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX}; \mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX};$$

- The step $\mathbf{TIX}(I)$ (short for xor, truncate, insert and xor) stands for the following sequence of steps:

$$\begin{aligned}&\mathbf{S}_{10} \mathrel{+}= \mathbf{S}_0;\\ &\mathbf{S}_0 = I \text{ (i.e., } \mathbf{S}_0^i = I^i \text{ for } i = 0, 1, 2, 3);\\ &\mathbf{S}_8 \mathrel{+}= \mathbf{S}_0;\\ &\mathbf{S}_1 \mathrel{+}= \mathbf{S}_{24}.\end{aligned}$$

- The step **ROR3** just rotates the state to the right by three columns, i.e. simultaneously set $\mathbf{S}_i = \mathbf{S}_{i-3}$, where the subscript subtraction is performed modulo 30.
- The step **CMIX**, which stands for *column mix*, is

$$\begin{aligned}&\mathbf{S}_0 \mathrel{+}= \mathbf{S}_4;\ \mathbf{S}_1 \mathrel{+}= \mathbf{S}_5;\ \mathbf{S}_2 \mathrel{+}= \mathbf{S}_6;\\ &\mathbf{S}_{15} \mathrel{+}= \mathbf{S}_4;\ \mathbf{S}_{16} \mathrel{+}= \mathbf{S}_5;\ \mathbf{S}_{17} \mathrel{+}= \mathbf{S}_6;\end{aligned}$$

- The step **SMIX**, just operates on the first four columns of the state, which can be viewed as a 4×4 matrix $\mathbf{W}$. First, each byte of the matrix $\mathbf{W}$ undergoes an S-box substitution. Next, the resulting matrix undergoes the Super-Mix linear transformation. Thus, if S-Box[$\mathbf{W}$] denotes the matrix obtained by substituting each byte $\mathbf{W}_j^i$ by S-box[$\mathbf{W}_j^i$], then **SMIX** stands for

$$\mathbf{S}_{0..3} = \text{Super-Mix}(\text{S-box}[\mathbf{S}_{0..3}]).$$

(Recall that the additions above are addition of vectors of four bytes in GF(2^8), and hence is same as 32-bit exclusive-or.) The sequence of steps **ROR3**;**CMIX**;**SMIX** will also be referred to as a *sub-round*. Thus, a round in **F**-256 can be seen as a **TIX** step followed by two sub-rounds.

4.3.2 The Final Round G

The final round **G** takes a 30 column state **S** and produces a final 30 column state. It is best described by how it affects the state **S**, which is as follows.

```
repeat 5 times
    {
    ROR3; CMIX; SMIX
    ROR3; CMIX; SMIX
    }
repeat 13 times
    {
    $S_4 += S_0; S_{15} += S_0;$ ROR15; SMIX;
    $S_4 += S_0; S_{16} += S_0;$ ROR14; SMIX;
    }
  $S_4 += S_0; S_{15} += S_0;$
```

where **RORn** stands for rotating the state **S** to the right by n columns, i.e. simultaneously setting $\mathbf{S}_i = \mathbf{S}_{i-n}$ for all $i = 0..29$, and the subtraction in the subscript is modulo 30.

4.3.3 Initial State

The state **S** is initialized by setting its first 22 columns to *zero*, and the last 8 columns to the given IV. That is, the 32-byte IV is parsed as 8 four-byte words $\mathrm{IV}_0, \ldots, \mathrm{IV}_7$, and for all $j = 0...7$ we set $\mathbf{S}_{22+j} = \mathrm{IV}_j$. (In matrix notations, we set $\mathbf{S}_{0..21} = 0$ and $\mathbf{S}_{22..29} = \mathrm{IV}$.)

4.3.4 Hash Output

After the final round **G**, the following stream of eight words of the state **S** is output as the *hash value.*

$$\mathbf{S}_1\ \mathbf{S}_2\ \mathbf{S}_3\ \mathbf{S}_4\quad \mathbf{S}_{15}\ \mathbf{S}_{16}\ \mathbf{S}_{17}\ \mathbf{S}_{18}.$$

Note that the first word in the output is $\mathbf{S}_1$, and *not* $\mathbf{S}_0$. For example, if the final state begins with the five columns below

$$\begin{pmatrix} \mathbf{00} & \mathbf{04} & \mathbf{08} & \mathbf{0c} & \mathbf{10} & \\ \mathbf{01} & \mathbf{05} & \mathbf{09} & \mathbf{0d} & \mathbf{11} & \\ \mathbf{02} & \mathbf{06} & \mathbf{0a} & \mathbf{0e} & \mathbf{12} & \cdots \\ \mathbf{03} & \mathbf{07} & \mathbf{0b} & \mathbf{0f} & \mathbf{13} & \end{pmatrix}$$

then the first 16 bytes of the output would be

04 05 06 07 08 09 ... 12 13

4.3.5 Complete Specification of the Hash Function F-256

On input a stream of $4m$ bytes ($m \geq 0$) that are parsed as m four-byte words P_1, P_2,...,P_m, and the initial vector of 32 bytes (parsed as 8 four-byte words IV_0, IV_1,...,IV_7), the hash function **F**-256 operates as follows.

```
for $j = 0..21$, $\mathbf{S}_j = 0$;
for $j = 0..7$, Set $\mathbf{S}_{(22+j)} = \mathrm{IV}_j$.
for $i = 1..m$
{   $\mathbf{TIX}(P_i)$;
    repeat 2 times $\{\mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX};\}$
}
repeat 10 times $\{\mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX};\}$
repeat 13 times
{   $\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0; \mathbf{S}_{15} \mathrel{+}= \mathbf{S}_0; \mathbf{ROR15}; \mathbf{SMIX};$
    $\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0; \mathbf{S}_{16} \mathrel{+}= \mathbf{S}_0; \mathbf{ROR14}; \mathbf{SMIX};$
}
$\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0; \mathbf{S}_{15} \mathrel{+}= \mathbf{S}_0;$

Output $\mathbf{S}_1\ \mathbf{S}_2\ \mathbf{S}_3\ \mathbf{S}_4\quad \mathbf{S}_{15}\ \mathbf{S}_{16}\ \mathbf{S}_{17}\ \mathbf{S}_{18}$.
```

4.4 The Hash Function Fugue-256

The hash function Fugue-256 takes as input bit sequences of arbitrary length, upto $2^{64}-1$ bits, and returns a 256-bit output. It computes the output by padding the input with zeros to a multiple of 32-bits (four bytes), parsing it as a sequence of bytes, appending an 8-byte representation of the original input length, and applying **F**-256 to the result, along with a fixed initial vector (IV). The fixed IV that is used by Fugue-256 is defined (in matrix notations) as

$$\mathrm{IV256} = \begin{pmatrix} \mathbf{e9} & \mathbf{66} & \mathbf{e0} & \mathbf{d2} & \mathbf{f9} & \mathbf{fb} & \mathbf{91} & \mathbf{34} \\ \mathbf{52} & \mathbf{71} & \mathbf{d4} & \mathbf{b0} & \mathbf{6c} & \mathbf{f9} & \mathbf{49} & \mathbf{f8} \\ \mathbf{bd} & \mathbf{13} & \mathbf{f6} & \mathbf{b5} & \mathbf{62} & \mathbf{29} & \mathbf{e8} & \mathbf{c2} \\ \mathbf{de} & \mathbf{5f} & \mathbf{68} & \mathbf{94} & \mathbf{1d} & \mathbf{de} & \mathbf{99} & \mathbf{48} \end{pmatrix}$$

That is, the first word in IV256 is $\mathrm{IV256}_0$ = **e9 52 bd de**, the second word is $\mathrm{IV256}_1$ = **66 71 13 5f**, and so on upto the last word $\mathrm{IV256}_7$ = **34 f8 c2 48**. This fixed IV was obtained by running **F**-256 with an all-zero IV and the one-word input **00 00 01 00** (representing the decimal number 256).

In more details, let the input to Fugue-256 be a bit sequence X of length n bits ($n \leq 2^{64}-1$). We first append to X sufficiently many 0-bits to make its length a multiple of 32. That is, if n is a multiple of 32 then append nothing, otherwise append $32 - (n \bmod 32)$ zero bits.[1]

Let the resulting (possibly padded) string of bits be denoted X', and we view X' as a sequence of m bytes (in big-endian convention), $X' = B_0 B_1 \ldots B_{m-1}$, where m is a multiple of four. Next, the length n is represented as an eight-byte integer (in big-endian convention) and appended to X', to form the encoded stream X'' (of length $m + 8$ bytes). Then, **F**-256 is applied to the input X'' and the fixed IV value IV256 from above, returning a 32-byte output. This output is viewed as a 256-bit value (using again big-endian convention) and returned as the output of Fugue-256.

Example. Consider the 35-bit input

$$X = 10101001\ 10111000\ 11000111\ 11010110 \quad 010$$

We append to it 29 zero bits to form a 64-bit padded stream

$$X' \;=\; 10101001\ 10111000\ 11000111\ 11010110 \quad 01000000\ 00000000\ 00000000\ 00000000$$

which is viewed as an 8-byte stream X' = **a9 b8 c7 d6 40 00 00 00**. Next, the bit-length 35 (hexadecimal **23**) is represented as an eight-byte integer and appended to X' to form the encoded stream

$$X'' = \mathbf{a9\ b8\ c7\ d6 \quad 40\ 00\ 00\ 00 \quad 00\ 00\ 00\ 00 \quad 00\ 00\ 00\ 23}$$

and **F**-256 is applied to X''.

4.5 Pseudo-Random Function PR-Fugue-256

The function PR-Fugue-256 takes as input a binary string of length between 0 and $2^{64} - 1$, and a key of length 32 bytes, and produces as output a 32 byte value. Just as in Fugue-256, the input is first padded with zero bits to a length multiple of 32 bits, then the length of the original input is appended as an 8-byte integer, before running **F**-256 on the resulting encoded stream. The only difference between PR-Fugue-256 and Fugue-256 is that PR-Fugue-256 calls the underlying function **F**-256 using the 32-byte key as the IV value, instead of the fixed IV value IV256.

4.6 Compression Function C-Fugue-256

We define the compression function C-Fugue-256 as a backward-compatibility mode for applications that must use a compression function in a Merkle-Damgard mode. We stress that this is not the optimal way of using Fugue (from both performance and security perspectives), but it still offers an appropriate drop-in substitution for applications that need it.

---

[1] Note that as specified in Section 2 (and in accordance with the convention set in the Known-Answer-Test document from NIST), if the length of X is not an integral number of bytes then the padding to byte boundaries is done by adding zero bits in the *least significant* bit-positions of the byte.

The function C-Fugue-256 takes as input a binary string of length exactly 512 bits and an initial vector of 32 bytes, and produces an output of 32 bytes. The input is treated as a stream of 64 bytes, and the output of C-Fugue-256 is just the output of **F**-256 on this input and the given initial vector. (Note that the input is not padded, as it is already of the correct length.)

4.7 Other Modes of Operation

Fugue-256 can be used as a drop-in replacement for SHA-256 in many other modes of operation, including HMAC [17] and randomized hashing [18], without resorting to the backward-compatibility mode C-Fugue-256. For example, HMAC-Fugue-256 takes an input X and key K, and computes

$$\text{HMAC-Fugue-256}(K, X) \;=\; \text{Fugue-256}(K \oplus \mathsf{opad} \mid \text{Fugue-256}(K \oplus \mathsf{ipad} \mid X)\ )$$

We note that when the key is 32-byte long, then PR-Fugue-256 is a more efficient way of using Fugue to get a pseudo-random function.

5 Specification of Parameterized Fugue

The hash function Fugue-256 that was defined in the previous section is just once instance of a parameterized design. In this section we present this parameterized design, and call out the specific parameter setting for Fugue-224, Fugue-256, Fugue-384, and Fugue-512, as required by NIST (as well as a weakened version of Fugue-256 that may be more amenable to cryptanalysis). In its most generic form, an instance of Fugue depends on the following five parameters:

Output size n: the number of four-byte words in the output of Fugue, which is also the number of four-byte words in the IV. In this document we assume that $n \leq 16$ (i.e., the output size can be at most 512 bits).[2] For example, for Fugue-256 we have $n = 8$.

Work load k: the number of sub-rounds per round transformation. Recall that for every four-byte input word we apply a round transformation consisting of several sub-rounds (where the main component of a sub-round is the nonlinear SMIX permutation). Hence the parameter k specifies the word-load factor, i.e., how many SMIX-es we apply per four-byte input word. In Fugue-256 we have $k = 2$.

State size s: the number of four-byte columns in the internal state. We require that s be divisible by 3 and by $\lceil n/4 \rceil$, and moreover that $s \geq \mathsf{max}(6k, 2n)$. In Fugue-256 we have $s = 30$.

TIX-less rounds r: the number of rounds in the first phase of the final transformation **G**. These rounds are the same as the round-transformation **R** that is used to process the inputs, except that they contain no **TIX** step. In Fugue-256 we have $r = 5$.

---

[2] There is no technical reason that forbids longer outputs, but the specification becomes increasingly awkward for longer outputs.

Final rounds t: the number of rounds in the second phase of the final transformation **G**. In Fugue-256 we have $t = 13$.

5.1 The parameterized function $\mathbf{F}[n, k, s, r, t]$

The parameterized hash function $\mathbf{F}[n, s, k, r, t]$ takes an input stream of $4m$ bytes (for some $m \geq 0$) and an IV of $4n$ bytes. Just like **F**-256, it begins by initializing a $4 \times s$ state matrix **S** using the IV, then applies one round transformation **R** for every four-byte input word (in order), then applies a final transformation **G** (which itself consists of two transformations — G1 and then G2), and finally outputs part of the resulting final state. Below we denote the m four-byte input words by $P_0, P_1, \ldots, P_{m-1}$ and the n IV words by IV_0,IV_1, … ,IV_{n-1}.

Initialize State.

```
For $j = 0$ to $s - 1 - n$, set $\mathbf{S}_j = 0$.
For $j = 0$ to $n - 1$, set $\mathbf{S}_{s-n+j} = \mathrm{IV}_j$.
```

The Round Transformation $\mathbf{R[s,k](P)}$:

```
$\mathbf{TIX}[s, k](P)$;
Repeat $k$ times:
    { ROR3;CMIX[$s$];SMIX; }
```

where

- The step **SMIX** is same as that defined for **F**-256, and is independent of the parameters.
- **ROR3** is same as before, i.e. simultaneously set $\mathbf{S}_i = \mathbf{S}_{i-3}$, for all i = 1 to (s-1). In general **RORn** will stand for simultaneously setting $\mathbf{S}_i = \mathbf{S}_{i-n}$, for all i = 1 to (s-1). All arithmetic of the column indices of the state **S** is done modulo s.
- The step $\mathbf{TIX}[s, k](P)$ takes one four-bytes work of input P, and is defined as follows:

```
$\mathbf{S}_{6k-2} \mathrel{+}= \mathbf{S}_0$;
$\mathbf{S}_0 = P$;
$\mathbf{S}_8 \mathrel{+}= \mathbf{S}_0$;
For $i =$ 0 to $k - 2$ step 1 :
    $\mathbf{S}_{3i+1} \mathrel{+}= \mathbf{S}_{s-3k+3i}$
```

Note that, if k is one, then the loop is not executed at all.

- The definition of **CMIX**[s] is:

  ```
  $\mathbf{S}_0 + = \mathbf{S}_4$;  $\mathbf{S}_{s/2} \quad + = S_4$;
  $\mathbf{S}_1 + = \mathbf{S}_5$;  $\mathbf{S}_{s/2+1} + = \mathbf{S}_5$;
  $\mathbf{S}_2 + = \mathbf{S}_6$;  $\mathbf{S}_{s/2+2} + = \mathbf{S}_6$;
  ```

The final transformation G consists of a first phase $G1$ followed by second phase $G2$. The first phase consists just of r rounds of TIX-less round transformations, namely:

- $\mathbf{G1}[\mathbf{k}, \mathbf{s}, \mathbf{r}]$: *Repeat* rk times: { **ROR3**; **CMIX**[s]; **SMIX**; }

The second phase needs some more explanation: Recall that $\mathbf{F}[n, k, s, r, t]$ needs to produce $4n$ bytes of output, which it does by using n of the columns in the final state. These n columns are partitioned into groups of four, and each group is taken from a different part of the state. Below we denote the number of groups of four columns by $N = \lceil n/4 \rceil$ (and recall that we require that the number of columns s is divisible by N).

If we need only four (or less) columns ($N = 1$) then we take $\mathbf{S}_{1..4}$ or a prefix of them. If we need 5-8 columns ($N = 2$) then we take $\mathbf{S}_{1..4}\mathbf{S}_{\frac{s}{2}..\frac{s}{2}+3}$ or a prefix of them. Similarly to get 9-12 columns ($N = 3$) we take $\mathbf{S}_{1..4}\mathbf{S}_{\frac{s}{3}..\frac{s}{3}+3}\mathbf{S}_{\frac{2s}{3}..\frac{2s}{3}+3}$ (or a prefix), etc.

Roughly speaking, a round of **G2** applies SMIX to each of the above groups (with some additional mixing), and rotates the entire state by one columns to the left.[3] The additional mixing roughly takes the leftmost column that resulted from the previous SMIX and adds it to one columns in each of these groups. A more accurate description follows:

- $\mathbf{G2}[\mathbf{n}, \mathbf{s}, \mathbf{t}]$: Denote the number of groups as above by $N = \lceil n/4 \rceil$, and also denote $p = s/N$:
  - **If** $N = 1$ then:

    ```
    Repeat $t$ times: {  $\mathbf{S}_4$ += $\mathbf{S}_0$; $\mathbf{ROR(s-1)}$; SMIX; }
    $\mathbf{S}_4$ += $\mathbf{S}_0$;
    ```

  - **If** $N = 2$ then:

    ```
    Repeat $t$ times: { $\mathbf{S}_4$ += $\mathbf{S}_0$; $\mathbf{S}_p$   += $\mathbf{S}_0$; $\mathbf{ROR(p)}$;     SMIX;
                        $\mathbf{S}_4$ += $\mathbf{S}_0$; $\mathbf{S}_{p+1}$ += $\mathbf{S}_0$; $\mathbf{ROR(p-1)}$; SMIX; }
    $\mathbf{S}_4$ += $\mathbf{S}_0$; $\mathbf{S}_p$ += $\mathbf{S}_0$;
    ```

  - **If** $N = 3$ then:

    ```
    Repeat $t$ times: { $\mathbf{S}_4$ += $\mathbf{S}_0$; $\mathbf{S}_p$   += $\mathbf{S}_0$; $\mathbf{S}_{2p}$   += $\mathbf{S}_0$; $\mathbf{ROR(p)}$;     SMIX;
                        $\mathbf{S}_4$ += $\mathbf{S}_0$; $\mathbf{S}_{p+1}$ += $\mathbf{S}_0$; $\mathbf{S}_{2p}$   += $\mathbf{S}_0$; $\mathbf{ROR(p)}$;     SMIX;
                        $\mathbf{S}_4$ += $\mathbf{S}_0$; $\mathbf{S}_{p+1}$ += $\mathbf{S}_0$; $\mathbf{S}_{2p+1}$ += $\mathbf{S}_0$; $\mathbf{ROR(p-1)}$; SMIX; }
    $\mathbf{S}_4$ += $\mathbf{S}_0$; $\mathbf{S}_p$ += $\mathbf{S}_0$; $\mathbf{S}_{2p}$ += $\mathbf{S}_0$;
    ```

---

[3] More accurately, we apply SMIX one column left of these groups, for example to $\mathbf{S}_{0..3}$ rather than $\mathbf{S}_{1..4}$.

- **If** $N = 4$ then:

```
Repeat $t$ times:
{   $\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_p \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{2p} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{3p} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{ROR(p)}$;  $\mathbf{SMIX}$;
    $\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{p+1} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{2p} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{3p} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{ROR(p)}$;  $\mathbf{SMIX}$;
    $\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{p+1} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{2p+1} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{3p} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{ROR(p)}$;  $\mathbf{SMIX}$;
    $\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{p+1} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{2p+1} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{3p+1} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{ROR(p-1)}$;  $\mathbf{SMIX}$;
}
$\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_p \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{2p} \mathrel{+}= \mathbf{S}_0$;  $\mathbf{S}_{3p} \mathrel{+}= \mathbf{S}_0$;
```

Output. After the transformation **G2**, the output of $\mathbf{F}[n,k,s,r,t]$ consists of the columns $\mathbf{S}_{1..4}\mathbf{S}_{p..p+3}\mathbf{S}_{2p..2p+3}$ ... Namely, if $n \leq 4$ then we just output $\mathbf{S}_{1..n}$, and if $n > 4$ then we output as follows:

```
Output $\mathbf{S}_{1..4}$ ;
For $i = 1$ to $N-2$, Output $\mathbf{S}_{ip\,..\,ip+3}$ ;
Output as many of the columns $\mathbf{S}_{s-p\,..\,s-p+3}$ as needed
```

5.1.1 A Complete Specification of $\mathbf{F}[n,k,s,r,t]$

With the above description, the complete specification of $\mathbf{F}[n,k,s,r,t]$ is as follows: On input consisting of m four-byte words ($m \geq 0$), denoted $P_0, P_1, \ldots P_{m-1}$, and an initial vector of n four-byte words, denoted IV_0,IV_1,...,IV_{n-1}, the hash function $\mathbf{F}[n,s,k,r]$ maintains a $4 \times s$ state matrix $\mathbf{S}$ and operates as follows:

```
For $j = 0..(s-n-1)$, $\mathbf{S}_j = 0$;
For $j = 0..(n-1)$, $\mathbf{S}_{(s-n+j)} = \mathrm{IV}_j$.
For $i = 1..m$
{ $\mathbf{TIX}[s,k](P_i)$;
  Repeat $k$ times : $\{\mathbf{ROR3}; \mathbf{CMIX}[s]; \mathbf{SMIX};\}$
}
$\mathbf{G1}[k,s,r]$;
$\mathbf{G2}[n,s,t]$;

If $n \leq 4$ then Output $\mathbf{S}_{1..n}$;
else
    Output $\mathbf{S}_{1..4}$;
    For $i = 1..(N-2)\{$ Output $\mathbf{S}_{ip\,..\,ip+3};\}$
    Output as many of the columns $\mathbf{S}_{s-p\,..\,s-p+3}$ as needed.
```

5.2 Parameter Specifications for Different Output Lengths

For the output lengths that are required by NIST, we specify the following setting of parameters:

- **F**-224 is $\mathbf{F}[n = 7,\ s = 30, k = 2, r = 5, t = 13]$.
- **F**-256 is $\mathbf{F}[n = 8,\ s = 30, k = 2, r = 5, t = 13]$.
- **F**-384 is $\mathbf{F}[n = 12, s = 36, k = 3, r = 6, t = 13]$.
- **F**-512 is $\mathbf{F}[n = 16, s = 36, k = 4, r = 8, t = 13]$.

In general, we recommend state size $s = 30$ for $n \leq 8$, and $s = 36$ for $n \in [9, 16]$. In addition, we define a weak version of **F**-256 (denoted w**F**-256) as $\mathbf{F}[n = 8, s = 30, k = 1, r = 5, t = 5]$. Note that the size of the state remains the same, but the work-load (i.e., number of SMIX-es per input word) is cut to just one rather than two. The number of rounds in the second phase of the final transformation **G** is also reduced to 5 from 13.

5.3 Fugue-224 and related functions

The function **F**-224 is exactly the same as the function **F**-256, except that the output is truncated to the first 28 bytes. In other words, instead of outputting $\mathbf{S}_{1..4}\mathbf{S}_{15..18}$, the output of **F**-224 is only $\mathbf{S}_{1..4}\mathbf{S}_{15..17}$.

The hash function Fugue-224 does the padding on the given input just as Fugue-256 does, but uses a different fixed initial value. Specifically, the fixed initial value IV224 is defined (in matrix notations) as

$$\text{IV224} = \begin{pmatrix} \mathbf{f4} & \mathbf{62} & \mathbf{ee} & \mathbf{e0} & \mathbf{8b} & \mathbf{9a} & \mathbf{bd} \\ \mathbf{c9} & \mathbf{86} & \mathbf{39} & \mathbf{74} & \mathbf{50} & \mathbf{43} & \mathbf{8d} \\ \mathbf{12} & \mathbf{f7} & \mathbf{e0} & \mathbf{e3} & \mathbf{a7} & \mathbf{d2} & \mathbf{67} \\ \mathbf{0d} & \mathbf{57} & \mathbf{1c} & \mathbf{cb} & \mathbf{2f} & \mathbf{15} & \mathbf{9a} \end{pmatrix}$$

That is, the first word in IV224 is $\text{IV224}_0 =$ **f4 c9 12 0d**, the second word is $\text{IV224}_1 =$ **62 86 f7 57**, and so on upto the last word $\text{IV224}_7 =$ **bd 8d 67 9a**. This IV was obtained by running **F**-224 with an all-zero IV and the one-word input **00 00 00 e0** (representing the decimal number 224).

5.4 Fugue-384 and related functions

Recall that the function **F**-384 was defined as $\mathbf{F}[12, 36, 3, 6, 13]$. Below we provide an explicit specification for this function: **F**-384 takes as input a stream of $4m$ bytes (for some $m \geq 0$) and an initial vector IV of 48 bytes, and produces an output of 48 bytes, using a 4×36 internal state matrix. The the relevant basic transformations of the state are as follows:

1. $\mathbf{TIX}(I)$ is the following sequence of steps:

$$\begin{aligned}&\mathbf{S}_{16} \mathrel{+}= \mathbf{S}_0;\\ &\mathbf{S}_0 = I;\\ &\mathbf{S}_8 \mathrel{+}= \mathbf{S}_0;\\ &\mathbf{S}_1 \mathrel{+}= \mathbf{S}_{27};\ \mathbf{S}_4 \mathrel{+}= \mathbf{S}_{30};.\end{aligned}$$

2. The transformation **CMIX** is

$$\begin{aligned}&\mathbf{S}_0 \mathrel{+}= \mathbf{S}_4;\ \mathbf{S}_1 \mathrel{+}= \mathbf{S}_5;\ \mathbf{S}_2 \mathrel{+}= \mathbf{S}_6;\\ &\mathbf{S}_{18} \mathrel{+}= \mathbf{S}_4;\ \mathbf{S}_{19} \mathrel{+}= \mathbf{S}_5;\ \mathbf{S}_{20} \mathrel{+}= \mathbf{S}_6;\end{aligned}$$

On input of a stream of m 4-byte words ($m \geq 0$) $P_1, P_2 \ldots, P_m$, and an initial vector of 12 four-byte words IV_0,IV_1,...,IV_{11}, the hash function **F**-384 operates as follows.

```
For $j = 0..23$,  set $\mathbf{S}_j = 0$;
For $j = 0..11$,  set $\mathbf{S}_{(24+j)} = \mathrm{IV}_j$.
For $i = 1..m$
{  $\mathbf{TIX}(P_i)$;
   Repeat 3 times  : $\{\mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX};\}$
}
Repeat 18 times  : $\{\mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX};\}$
Repeat 13 times :
{
   $\mathbf{S}_4 += \mathbf{S}_0$;  $\mathbf{S}_{12} += \mathbf{S}_0$;  $\mathbf{S}_{24} += \mathbf{S}_0$;  $\mathbf{ROR12}$;  $\mathbf{SMIX}$;
   $\mathbf{S}_4 += \mathbf{S}_0$;  $\mathbf{S}_{13} += \mathbf{S}_0$;  $\mathbf{S}_{24} += \mathbf{S}_0$;  $\mathbf{ROR12}$;  $\mathbf{SMIX}$;
   $\mathbf{S}_4 += \mathbf{S}_0$;  $\mathbf{S}_{13} += \mathbf{S}_0$;  $\mathbf{S}_{25} += \mathbf{S}_0$;  $\mathbf{ROR11}$;  $\mathbf{SMIX}$;
}
$\mathbf{S}_4 += \mathbf{S}_0$;  $\mathbf{S}_{12} += \mathbf{S}_0$;  $\mathbf{S}_{24} += \mathbf{S}_0$;

Output $\mathbf{S}_{1..4}\mathbf{S}_{12..15}\mathbf{S}_{24..27}$
```

The function Fugue-384 takes as input a binary string of length between zero and $2^{64} - 1$, and outputs a stream of 48 bytes. The input string is padded and length-encoded just as for Fugue-256 (see Section 4.4), and the function **F**-384 is invoked on the encoded stream of bytes (of length multiple of 4), and the following fixed initial vector IV384:

$$\mathrm{IV384} = \begin{pmatrix} \mathbf{aa} & \mathbf{31} & \mathbf{a0} & \mathbf{00} & \mathbf{21} & \mathbf{74} & \mathbf{fa} & \mathbf{47} & \mathbf{e5} & \mathbf{a9} & \mathbf{bc} & \mathbf{5c} \\ \mathbf{61} & \mathbf{25} & \mathbf{1d} & \mathbf{60} & \mathbf{5e} & \mathbf{1b} & \mathbf{69} & \mathbf{3e} & \mathbf{02} & \mathbf{9c} & \mathbf{95} & \mathbf{10} \\ \mathbf{ec} & \mathbf{2e} & \mathbf{b4} & \mathbf{09} & \mathbf{f4} & \mathbf{5e} & \mathbf{3e} & \mathbf{b0} & \mathbf{ae} & \mathbf{25} & \mathbf{51} & \mathbf{95} \\ \mathbf{0d} & \mathbf{1f} & \mathbf{c7} & \mathbf{85} & \mathbf{4a} & \mathbf{9c} & \mathbf{9a} & \mathbf{40} & \mathbf{8a} & \mathbf{e0} & \mathbf{7c} & \mathbf{a1} \end{pmatrix}$$

This fixed IV was obtained by running **F**-384 with an all-zero IV and the one-word input **00 00 01 80** (representing the decimal number 384).

The function PR-Fugue-384, takes as input a binary string of length between 0 and $2^{64} - 1$, and a key of length 48 bytes, and produces as output a 48 byte value. The input is first padded and length-encoded, just as for Fugue-384, and then the function **F**-384 is invoked on this encoded input, along with the given key as the 48 byte IV. The output of this invocation of **F**-384, is the output of PR-Fugue-384.

The function C-Fugue-384, takes as input a binary string of length exactly 512 bits, and an initial vector of 48 bytes, and produces an output of 48 bytes. The input is treated as a stream of 64 bytes, and the output of C-Fugue-384 is same as the output of **F**-384 on this input, and the given initial vector. Note that the input is not padded, as it is already of the correct length.

5.5 Fugue-512 and related functions

Recall that the function **F**-512 was defined as $\mathbf{F}[16, 36, 4, 8, 13]$. Below we provide an explicit specification for this function: **F**-512 takes as input a stream of $4m$ bytes (for some $m \geq 0$) and an initial vector IV of 64 bytes, and produces an output of 64 bytes, using a 4×36 internal state matrix. The the relevant basic transformations of the state are as follows:

1. $\mathbf{TIX}(I)$ is the following sequence of steps:

$$\begin{aligned}&\mathbf{S}_{22} \mathrel{+}= \mathbf{S}_0;\\ &\mathbf{S}_0 = I;\\ &\mathbf{S}_8 \mathrel{+}= \mathbf{S}_0;\\ &\mathbf{S}_1 \mathrel{+}= \mathbf{S}_{24};\ \mathbf{S}_4 \mathrel{+}= \mathbf{S}_{27};\ \mathbf{S}_7 \mathrel{+}= \mathbf{S}_{30};\end{aligned}$$

2. The transformation **CMIX** is

$$\begin{aligned}&\mathbf{S}_0 \mathrel{+}= \mathbf{S}_4;\ \mathbf{S}_1 \mathrel{+}= \mathbf{S}_5;\ \mathbf{S}_2 \mathrel{+}= \mathbf{S}_6;\\ &\mathbf{S}_{18} \mathrel{+}= \mathbf{S}_4;\ \mathbf{S}_{19} \mathrel{+}= \mathbf{S}_5;\ \mathbf{S}_{20} \mathrel{+}= \mathbf{S}_6;\end{aligned}$$

On input of a stream of m 4-byte words ($m \geq 0$) $P_1, P_2 \ldots, P_m$, and an initial vector of 16 four-byte words IV_0,IV_1,...,IV_{15}, the hash function **F**-512 operates as follows.

```
For $j = 0..19$, set $\mathbf{S}_j = 0$;
For $j = 0..15$, set $\mathbf{S}_{(20+j)} = \mathrm{IV}_j$.
For $i = 1..m$
{ $\mathbf{TIX}(P_i)$;
  Repeat 4 times : $\{\mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX}; \}$
}
Repeat 32 times : $\{\mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX}; \}$
Repeat 13 times :
{
  $\mathbf{S}_4 += \mathbf{S}_0;\ \mathbf{S}_9 += \mathbf{S}_0;\ \mathbf{S}_{18} += \mathbf{S}_0;\ \mathbf{S}_{27} += \mathbf{S}_0;\ \mathbf{ROR9};\ \mathbf{SMIX};$
  $\mathbf{S}_4 += \mathbf{S}_0;\ \mathbf{S}_{10} += \mathbf{S}_0;\ \mathbf{S}_{18} += \mathbf{S}_0;\ \mathbf{S}_{27} += \mathbf{S}_0;\ \mathbf{ROR9};\ \mathbf{SMIX};$
  $\mathbf{S}_4 += \mathbf{S}_0;\ \mathbf{S}_{10} += \mathbf{S}_0;\ \mathbf{S}_{19} += \mathbf{S}_0;\ \mathbf{S}_{27} += \mathbf{S}_0;\ \mathbf{ROR9};\ \mathbf{SMIX};$
  $\mathbf{S}_4 += \mathbf{S}_0;\ \mathbf{S}_{10} += \mathbf{S}_0;\ \mathbf{S}_{19} += \mathbf{S}_0;\ \mathbf{S}_{28} += \mathbf{S}_0;\ \mathbf{ROR8};\ \mathbf{SMIX};$
}
$\mathbf{S}_4 += \mathbf{S}_0;\ \mathbf{S}_9 += \mathbf{S}_0;\ \mathbf{S}_{18} += \mathbf{S}_0;\ \mathbf{S}_{27} += \mathbf{S}_0;$

Output $\mathbf{S}_{1..4}\mathbf{S}_{9..12}\mathbf{S}_{18..21}\mathbf{S}_{27..30}$
```

The function Fugue-512 takes as input a binary string of length between zero and $2^{64} - 1$, and outputs a stream of 64 bytes. The input string is padded and length-encoded just as for Fugue-256 (see Section 4.4), and the function **F**-512 is invoked on the encoded stream of bytes (of length multiple of 4), and the following fixed initial vector IV512:

$$\mathrm{IV512} = \begin{pmatrix} \mathbf{88} & \mathbf{e6} & \mathbf{c5} & \mathbf{ac} & \mathbf{d9} & \mathbf{b6} & \mathbf{06} & \mathbf{4a} & \mathbf{aa} & \mathbf{dd} & \mathbf{ca} & \mathbf{43} & \mathbf{25} & \mathbf{95} & \mathbf{da} & \mathbf{e1} \\ \mathbf{07} & \mathbf{16} & \mathbf{d3} & \mathbf{9a} & \mathbf{15} & \mathbf{ee} & \mathbf{e8} & \mathbf{92} & \mathbf{c6} & \mathbf{b2} & \mathbf{e6} & \mathbf{7f} & \mathbf{ea} & \mathbf{1f} & \mathbf{6e} & \mathbf{3e} \\ \mathbf{a5} & \mathbf{af} & \mathbf{e4} & \mathbf{b0} & \mathbf{f1} & \mathbf{cc} & \mathbf{02} & \mathbf{ef} & \mathbf{e2} & \mathbf{13} & \mathbf{58} & \mathbf{20} & \mathbf{78} & \mathbf{dd} & \mathbf{d1} & \mathbf{35} \\ \mathbf{7e} & \mathbf{75} & \mathbf{db} & \mathbf{27} & \mathbf{17} & \mathbf{54} & \mathbf{0b} & \mathbf{d1} & \mathbf{c9} & \mathbf{98} & \mathbf{38} & \mathbf{3f} & \mathbf{e7} & \mathbf{d6} & \mathbf{1d} & \mathbf{67} \end{pmatrix}$$

This fixed IV was obtained by running **F**-512 with an all-zero IV and the one-word input **00 00 02 00** (representing the decimal number 512).

The function PR-Fugue-512, takes as input a binary string of length between 0 and $2^{64} - 1$, and a key of length 64 bytes, and produces as output a 64 byte value. The input is first padded and length-encoded, just as for Fugue-384, and then the function **F**-512 is invoked on this encoded input, along with the given key as the 64 byte IV. The output of this invocation of **F**-512, is the output of PR-Fugue-512.

The function C-Fugue-512, takes as input a binary string of length exactly 512 bits, and an initial vector of 64 bytes, and produces an output of 64 bytes. The input is treated as a stream of 64 bytes, and the output of C-Fugue-512 is same as the output of **F**-512 on this input, and the given initial initial vector. Note that the input is not padded, as it is already of the correct length.

5.6 A Weaker Version of Fugue-256

Recall that the function w**F**-256 was defined as $\mathbf{F}[8, 30, 1, 5, 5]$. Below we provide an explicit specification for this function: **F**-256 takes as input a stream of $4m$ bytes (for some $m \geq 0$) and an initial vector IV of 32 bytes, and produces an output of 32 bytes, using a 4×30 internal state matrix. The the relevant basic transformations of the state are as follows:

1. **TIX**(I) is the following sequence of steps: $\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0$; $\mathbf{S}_0 = I$; $\mathbf{S}_8 \mathrel{+}= \mathbf{S}_0$;.
2. The transformation **CMIX** is: $\mathbf{S}_{0..2} \mathrel{+}= \mathbf{S}_{4..6}$; $\mathbf{S}_{15..17} \mathrel{+}= \mathbf{S}_{4..6}$;

On input of a stream of m 4-byte words ($m \geq 0$) $P_1, P_2 \ldots, P_m$, and an initial vector of 8 four-byte words IV_0,IV_1,...,IV_7, the hash function **F**-256 operates as follows.

```
For $j = 0..21$, set $\mathbf{S}_j = 0$;
For $j = 0..7$, set $\mathbf{S}_{(22+j)} = \text{IV}_j$;
For $i = 1..m$
{ $\mathbf{TIX}(P_i)$;
  $\mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX}$;
}
Repeat 5 times : {$\mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX};$}
Repeat 5 times :
{
  $\mathbf{S}_4 += \mathbf{S}_0$; $\mathbf{S}_{15} += \mathbf{S}_0$; $\mathbf{ROR15}$; $\mathbf{SMIX}$;
  $\mathbf{S}_4 += \mathbf{S}_0$; $\mathbf{S}_{16} += \mathbf{S}_0$; $\mathbf{ROR14}$; $\mathbf{SMIX}$;
}
$\mathbf{S}_4 += \mathbf{S}_0$; $\mathbf{S}_{15} += \mathbf{S}_0$;

Output $\mathbf{S}_{1..4}\mathbf{S}_{15..18}$
```

Part II

Implementation

6 Software and Hardware Efficiency

6.1 Implementing Fugue on 32-bit Machines

The main construct in Fugue is the **SMIX** step. It itself is composed of the non-linear S-Box, and the Super-Mix linear transformation. Fugue has been designed, with ideas from AES to get a fast implementation on 32-bit machines. Indeed, the S-Box substitution followed by the linear transformation can be pre-computed in four 32-bit tables of 256 entries each. From the high level description of the Super-Mix transformation in Section 1.5, it should be clear how these four tables can be used to implement a highly optimized version of Fugue.

Moreover, since Fugue over different hash lengths, can be defined in a parameterized fashion, with the same underlying building blocks, and namely **SMIX**, the optimized implementations scale nicely. The performance figures for 32-bit Intel Machines are shown in Table 1, and compared with Brian Gladman's highly optimized SHA-2 implementations [2].

One of the main limitations on Intel's 32-bit machines is the limited availability of general purpose registers. Therefore, an assembly inline C code, which instructs the machine to reserve registers for specific tasks in the Super-Mix transformation can lead to a much better implementation than just an ANSI C implementation in older machines.

6.2 Implementing Fugue on 64-bit Machines

Our optimized implementation of Fugue for 32-bit Intel Architectures, runs fairly well on the 64-bit Intel Architectures. We have not yet capitalized on the 64-bit word size, although we suspect that the ANSI C implementation does take advantage of the extra general purpose registers.

6.3 Implementing Fugue on 8-bit Architectures

Implementing Fugue on an 8-bit processor is rather straightforward. We can use more or less the same implementation as for AES, with only small changes to account for the differences between the AES round function and the **SMIX** of Fugue.

The RAM requirements include the 120-byte state of Fugue-256 or Fugue-224 (or a 144-byte state for Fugue-384 and Fugue-512), a few more counters and indexes, and 16-20 bytes for scratch calculations, all of which should fit comfortably in under 200 bytes of RAM.

In terms of performance, the similarity between the **SMIX** of Fugue and the AES round function makes is easy to relate the performance of Fugue to that of AES. Specifically, in AES each byte (after the S-box substitution) is multiplied by 1, 2, and 3 (over GF(2^8)) and these

Table 1: Various Software Implementation Speeds (MBytes/sec)

No. of Input bytes		256	1K	16K
X86_32[†]				
ANSI C Fugue-256	Intel (WIN) Compiler v11.1[1]	55	75	90
	gcc 3.4.4 (cygwin)[2]	20	43	59
ASM C[a] Fugue-256	gcc 3.4.4 (cygwin)[2]	23	48	69
ANSI C Fugue-384	Intel (WIN) Compiler v11.1[1]	27	45	60
ANSI C Fugue-512	Intel (WIN) Compiler v11.1[1]	23	37	45
ANSI C SHA-256[b]	Intel (WIN) Compiler v11.1[3]			126
X86_64[†]				
ANSI C Fugue-256	gcc 4.3.3 (LINUX)[4]	64	90	97
ANSI C Fugue-384	gcc 4.3.3 (LINUX)[4]	36	57	64
ANSI C Fugue-512	gcc 4.3.3 (LINUX)[4]	32	42	48.5
ANSI C SHA-256[b]	gcc 4.3.3 (LINUX)[5]			120

[†] Intel Core 2 T7700 (90nm) 2.4GHz

[a] non-SSE Assembly Inline

[b] Brian Gladman's Code [2]

[1] icl /O2 /arch:ia32

[2] gcc -O3 -fomit-frame-pointer

[3] icl /O2 /arch:SSSE3

[4] gcc -O3 -frename-registers

[5] gcc -O3 -msse2

different multiples are xor-ed into four different bytes. In **SMIX**, each byte is multiplied by 1,4, and 7 and these different multiples are usually xor-ed into seven different bytes (except the bytes on the diagonal that are only xor-ed into four bytes). Fugue has some additional overhead since one may need to explicitly perform the row-shift (at least occasionally), we need also to implement the **CMIX** operation (which consists of 24 byte-xors) and the **TIX** operation (consisting of four byte movements and 12 to 24 byte-xors), and we also need some index manipulation to implement the column-rotate operations. (On the other hand, we do not have the key-add operation that takes 16 byte-xors in AES and of course we do not have to implement key-scheduling.)

All in all, comparing an implementation of Fugue to an AES implementation *that pre-computes the key-schedule*, we believe that a sub-round of Fugue should take less that twice the cycles of an AES round function. Hence the final transformation **G** of Fugue-256 and Fugue-224 (which consists of $5 + 13 \times 2 = 36$ applications of **SMIX**) should take about the same time as AES-256 encryption of five blocks (using an implementation that pre-computes the key-scheduling), and the round transformation (that consists of two **SMIX**-es) should take about 25% of the time of one AES-256 encryption. Using these estimates, we conclude that computing the underlying **F**-256 on a 16-byte message will be about six times slower than computing AES-256 on the same message, whereas for long messages the speed of **F**-256 will approach that of AES-256.

In [11] it was reported that an implementation of AES-128 on the 8-bit 6805 CPU core would take about 15000 cycles to encrypt one block, including the time for key-setup. Assuming that without the key-schedule we will have 10000 cycles for AES-128 encryption (or 14000 for AES-256 encryption), and using our estimate from above for the relative speed of Fugue vs. AES, we estimate that hashing a 128-bit message with Fugue-256 would take about 84000 cycles on the same 6805 CPU core. For longer messages, the performance of Fugue will improve to less than 20000 cycles per sixteen bytes of message.

6.4 Hardware Implementations of Fugue

We implemented Fugue-256 in the Hardware Design Language Verilog, and optimized the implementation for three different memory-speed tradeoffs. The Verilog code can be complied into gates using various synthesis tools. In our case, we targeted the IBM Standard Cell Library for IBM's Copper (Cu-8, 90ns, bulk silicon) technology. We then used approximation tools to estimate the actual cell count required to implement these various tradeoffs. For comparison sake, we also implemented SHA-256 with the same library and for the same technology.

The estimated cell count, the number of flip-flops, the clock speed possible, ad the resulting core speed, and the asymptotic hash speed per word of input are shown in Table 2. The implementations of Fugue-256 include three main variants: (a) all four columns of the **SMIX** transformation are handled together (SUPER4), (b) **SMIX** transformation is handled two columns at a time (SUPER2), and (c) **SMIX** transformation is handled one column at a time. Further, the basic S-Box, which is identical to the AES S-Box, and which is the most expensive component to implement, can have various implementations. The suffix "_P" in the Table refers to the IBM data path library cells to optimize S-Box lookups. The suffix "_F" refers to a similar implementation but

Table 2: Hardware Estimates for Various Fugue-256 and SHA-256 Implementations

Core	#Cells	#Flip-flops	Clocks/step	Clock Speed	Core Speed	Hash Speed
F-256:			(clk/smix)	(ns/clk)	(ns/smix)	(ns/word)
SUPER4_P	109854	1002	1	1.15	1.15	2.3
SUPER4_F	108990	1002	1	1.30	1.30	2.60
SUPER4_L	73774	1002	1	2.00	2.00	4.00
SUPER2_R	67465	1228	3	1.15	3.45	6.90
SUPER2_L	65062	1131	2	2.00	4.00	8.00
SUPER1_R	60420	1181	5	1.15	5.75	11.50
SUPER1_L	59216	1132	4	2.00	8.00	16.00
SHA-256:			(clk/step)	(ns/clk)	(ns/step)	(ns/word)
HASHCORE	46685	1114	2	1.30	2.60	10.4

with a different tradeoff. The suffix "_L" refers to a composite field arithmetic implementation of the AES S-Box [21]. The suffix "_R" refers to a pipelined implementation of the composite field implementation.

6.5 Other Implementations of Fugue

We expect Fugue implementations to benefit from many of the current trends in computer architectures. Specifically, below we comment briefly about the inherent parallelizability of Fugue and the potential for significant speedup using Multi-media extension to common architectures.

Parallelism. Just like the AES round function, the Fugue **SMIX** operation consists of sixteen nearly-independent threads. Namely, each of the sixteen bytes that are involved can be used independently to compute a 16-byte vector (with upto seven non-zero bytes) and the result of the **SMIX** transformation is just the xor of all these sixteen vectors. A multi-threaded architecture can take advantage of this inherent parallelism by assigning different input bytes to different threads.

Advanced byte-manipulation. Fugue implementations will also benefit from architectures such as AltiVec or SSE that allow parallel lookups and byte permutations in vectors. Using these tools was reported to allow an order of magnitude improvement for the software performance of the AES-based Whirlpool [7], and the performance gains for Fugue should similar. In fact, Fugue stands to benefit even more from byte-permutation instructions than AES, since the **SMIX** transformation includes also a matrix-transpose operation (and an explicit row-shift).

Part III

Security Analysis

7 The Super-Mix Matrix and Related Linear Codes

The 16×16 matrix $\mathbf{N}$ representing the Super-Mix linear transformation is associated with some linear codes that will be useful in the analysis of Fugue. Specifically, viewing (some of) the rows of $\mathbf{N}$ as parity check equations leads to linear codes that are either MDS or close to it. Recall that a linear code of dimension k and length n (i.e., an $[n,k]$-code) has a minimum distance of at most $n-k+1$, and a linear code that attains this minimum distance is called *maximum distance separable*, or **MDS** code. Below we provide some background on linear codes and then discuss the codes that are associated with the Super-Mix matrix $\mathbf{N}$.

7.1 Linear Codes

An $[n,k]$ linear code over a field F is given by $n-k$ linearly independent parity check equations over F in n variables. Representing these equations by a $(n-k) \times n$ parity check matrix H, the corresponding code $\mathcal{C}$ consists of all the column vectors c of length n such that $H \cdot c = 0$. Since the rank of H is at most $n-k$, there must exist a non-zero column vector c of weight $n-k+1$ (or less) such that $H \cdot c = 0$. Hence the minimum distance of $\mathcal{C}$ is at most $n-k+1$. (We remind the reader that the weight of a vector is the number of non-zero entries in it, and for linear codes the minimum distance of a code is same as the minimum weight of its non-zero codewords.) Moreover, if H has any $(n-k) \times (n-k)$ sub-matrix of less than full rank, then there is a non-zero vector c of weight at most $n-k$ such that $H \cdot c = 0$. It follows that the minimum distance of $\mathcal{C}$ is $n-k+1$ if and only if every $(n-k) \times (n-k)$ sub-matrix of H is of full rank. As we mentioned above, such codes are called MDS.

Below we will also be interested in codes that are not MDS, but *almost*-MDS, i.e. where the minimum distance is close to $n-k+1$. Recalling that the minimum distance of $\mathcal{C}$ equals the size of *the smallest non-empty set* of columns that are *NOT linearly independent*, motivates the following definitions:

Definition 7.1 *Fix an $[n,k]$ linear code $\mathcal{C}$ with a parity-check matrix H, and an integer $m \leq n-k$. The $\mathsf{min\text{-}rank}_m$ of $\mathcal{C}$ is the minimum rank among all $(n-k) \times m$ sub-matrices of H. We further define $\mathsf{maxmin\text{-}rank}(\mathcal{C})$ as the largest m for which $\mathsf{min\text{-}rank}_m = m$.*

Observing that $\mathsf{min\text{-}rank}_m = m$ if and only if every size-m subsets of columns of H is linearly independent, we have the following lemma:

Lemma 7.2 *The minimum distance of a linear code $\mathcal{C}$ is exactly $\mathsf{maxmin\text{-}rank}(\mathcal{C}) + 1$.* ■

Below we will sometimes find it useful to consider also $\mathsf{min\text{-}rank}_m$ for integers m larger than the minimum distance of $\mathcal{C}$. Specifically, we use it to bound the number of codewords in $\mathcal{C}$ with certain specified indexes fixed to zero.

Lemma 7.3 *Let $\mathcal{C}$ be a $[n,k]$-code, let I be some fixed subset of the indexes 1 through n, and denote $m = n - |I|$. Then the dimension of the linear subspace $L = \{c \in \mathcal{C} :\ c^i = 0 \text{ for all } i \in I\}$ is at most $m - \mathsf{min\text{-}rank}_m(\mathcal{C})$.*

Proof: Below we denote by c^I the vector c restricted to the indexes in I, and by $c^{\bar{I}}$ the vector c excluding all the indexes in I. Consider now H, the $(n-k) \times n$ parity-check matrix of $\mathcal{C}$. We denote by H_I the sub-matrix of H consisting only of the columns corresponding to indexes in I, and by $H_{\bar{I}}$ we denote the sub-matrix obtained by excluding all these columns.

Clearly for every vector c we have $H \cdot c = H_I \cdot c^I + H_{\bar{I}} \cdot c^{\bar{I}}$, which means that when $c^I = 0$ then $H \cdot c = H_{\bar{I}} \cdot c^{\bar{I}}$. Hence we can re-write our linear subspace as $L = \{c :\ c^I = 0 \text{ and } H_{\bar{I}} \cdot c^{\bar{I}} = 0\}$. Moreover, since the entries corresponding to I must be fixed to zero, then this space has the same dimension as its projection on $\bar{I}$, namely

$$\dim(L) = \dim\left(\left\{c^{\bar{I}} :\ H_{\bar{I}} \cdot c^{\bar{I}} = 0\right\}\right)$$

Now, $H_{\bar{I}}$ is an $(n-k) \times m$ sub-matrix of H, so it has rank at least $\mathsf{min\text{-}rank}_m(\mathcal{C})$, which means that its null space has dimension at most $m - \mathsf{min\text{-}rank}_m(\mathcal{C})$. ■

7.2 Linear Codes Related to the Super-Mix Matrix N

Recall that we view the input (and output) of the Super-Mix transformation alternatively as a 4×4 matrix or a 16-element column vector (which is left-multiplied by the 16×16 matrix $\mathbf{N}$). The mapping between these representations is done by scanning the 4×4 matrices column-wise.

One of the key properties of the Super-Mix is that if a non-zero output has three zero columns, then the corresponding input has very few zeroes among its 16 bytes. This can be phrased as a coding property by using sub-matrices of the Super-Mix matrix $\mathbf{N}$ as parity check matrices. More generally, we look at linear codes corresponding to the output of the Super-Mix having one, two, or three zero columns.

Three zero columns. We first define four length-n linear codes over GF(2^8), each corresponding to one of the four columns in the output of the Super-Mix being zero. Specifically, for $m = 0, 1, 2, 3$ we denote by $\bar{I}_m$ the indexes corresponding to all bytes *except the ones in column m*, when scanning the 4×4 output matrix column-wise.

$$\begin{aligned}
\bar{I}_0 &= \{4,5,6,7,\quad 8,9,10,11,\quad 12,13,14,15\} \\
\bar{I}_1 &= \{0,1,2,3,\quad 8,9,10,11,\quad 12,13,14,15\} \\
\bar{I}_2 &= \{0,1,2,3,\quad 4,\ 5,\ 6,\ 7,\quad 12,13,14,15\} \\
\bar{I}_3 &= \{0,1,2,3,\quad 4,\ 5,\ 6,\ 7,\quad 8,\ 9,10,11\}
\end{aligned}$$

With these notations, the linear subspaces of inputs to the Super-Mix that result in all but one of the columns of the output being zero are

$$\begin{array}{rcl} \mathcal{C}_0 & = & \{c: \ \mathbf{N}^{\bar{I}_0} \cdot c = 0\} \\ \mathcal{C}_1 & = & \{c: \ \mathbf{N}^{\bar{I}_1} \cdot c = 0\} \\ \mathcal{C}_2 & = & \{c: \ \mathbf{N}^{\bar{I}_2} \cdot c = 0\} \\ \mathcal{C}_3 & = & \{c: \ \mathbf{N}^{\bar{I}_3} \cdot c = 0\} \end{array} \tag{4}$$

(where $\mathbf{N}^X$ is the sub-matrix of $\mathbf{N}$ consisting of only the rows indexed by X).

Since $\mathbf{N}$ is non-singular then all the parity check equations are linearly independent, hence all of these codes have dimension 4 (i.e., they are all $[16, 4]$ codes). Further, we used computer program to find the minimum distance of these codes, and found that $\mathcal{C}_0$ is an MDS code, namely it has minimum distance $16 - 4 + 1 = 13$. The code $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ are almost MDS, in the sense that their minimum distance is 12.

Two zero columns and one zero column. In the analysis of Fugue we also consider some specific subsets one two or one output columns that are zero. The codes corresponding to these subsets are defined similarly to the code of the three-zero-columns above. Specifically, we set:

$$\begin{array}{rcl} \bar{I}_{0,1} & = & \{\ 8,\ 9, 10, 11, 12, 13, 14, 15\} \\ \bar{I}_{1,2} & = & \{\ 0,\ 1,\ 2,\ 3,\ 12, 13, 14, 15\} \\ \bar{I}_{0,1,2} & = & \{12, 13, 14, 15\} \end{array}$$

where $\bar{I}_S$ indexes all the bytes *except the ones in the columns corresponding to* S, and then define the linear codes:

$$\begin{array}{rcl} \mathcal{C}_{0,1} & = & \{c: \ \mathbf{N}^{\bar{I}_{0,1}} \cdot c = 0\} \\ \mathcal{C}_{1,2} & = & \{c: \ \mathbf{N}^{\bar{I}_{1,2}} \cdot c = 0\} \\ \mathcal{C}_{0,1,2} & = & \{c: \ \mathbf{N}^{\bar{I}_{0,1,2}} \cdot c = 0\} \end{array} \tag{5}$$

Clearly, $\mathcal{C}_{0,1}$ and $\mathcal{C}_{1,2}$ are $[16, 8]$-codes and $\mathcal{C}_{0,1,2}$ is a $[16, 12]$-code, and we found their minimum distances to be 6,7, and 2, respectively. We also used a computer program to find the min-rank$_m$ values for all these codes for all values of m from the maxmin-rank and up. These are summarized in Table 3.

Finally, we mention, that the 4×4 matrix $\mathbf{M}$ in section 4.2, which is the alternate to the AES mixing matrix, continues to be an MDS matrix, i.e. it has the nice Mix-Column properties of AES. In other words, for any non-zero (column) vector v of four GF(2^8) elements, the number of non-zero entries in v and the vector $(\mathbf{M} \cdot v)$ together, is at least 5.

7.3 Implications for the Differential Properties of SMIX

The codes $\mathcal{C}_i$ and their properties are used extensively in the differential analysis of Fugue. Here we provide some illustrative examples of their use, more details are found in Section 10.

Table 3: Min-Rank$_m$ values for Various Linear Codes

m	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathcal{C}_0$	-	-	-	-	-	-	-	-	-	-	-	12	12	12	12	12
$\mathcal{C}_1$	-	-	-	-	-	-	-	-	-	-	11	11	12	12	12	12
$\mathcal{C}_2$	-	-	-	-	-	-	-	-	-	-	11	11	12	12	12	12
$\mathcal{C}_3$	-	-	-	-	-	-	-	-	-	-	11	11	12	12	12	12
$\mathcal{C}_{0,1}$	-	-	-	-	5	5	6	6	7	7	8	8	8	8	8	8
$\mathcal{C}_{0,2}$	-	-	3	3	4	4	5	6	7	7	7	8	8	8	8	8
$\mathcal{C}_{0,3}$	-	-	-	-	5	5	6	6	7	7	7	8	8	8	8	8
$\mathcal{C}_{1,2}$	-	-	-	-	-	6	6	7	7	7	8	8	8	8	8	8
$\mathcal{C}_{1,3}$	-	-	-	-	5	5	6	6	7	7	8	8	8	8	8	8
$\mathcal{C}_{2,3}$	-	-	3	3	4	5	6	6	7	7	7	8	8	8	8	8
$\mathcal{C}_{0,1,2}$	1	1	2	2	2	2	3	3	3	3	4	4	4	4	4	4

† The shaded entries are maxmin-ranks (i.e., one less than the corresponding minimum distances).

Example 1: three zero output columns. Consider the situation just before the successful conclusion of a collision-finding attack: the attacker provided two different messages that induced an almost-collision on the internal state of Fugue: only one column is different between the two internal states, and this difference will be written over in the next TIX step.

Looking at the last **SMIX** operation in processing the two messages, the four output columns for the first message are called $v = [v_0, v_1, v_2, v_3]$, and for the second message $v' = [v'_0, v'_1, v'_2, v'_3]$. The column that will be written over in the next TIX are v_0 and v'_0, so we know that these are the only columns that differ between these two states. Namely we have $v_0 + v'_0 \neq 0$ but $v_i + v'_i = 0$ for $i = 1, 2, 3$.

Let us denote the corresponding input columns to this last **SMIX** step for the two messages by $u = [u_0, u_1, u_2, u_3]$ and $u' = [u'_0, u'_1, u'_2, u'_3]$, and the columns after byte-substitution but before the linear Super-Mix are denoted $\hat{u} = [\hat{u}_0, \hat{u}_1, \hat{u}_2, \hat{u}_3]$ and $\hat{u}' = [\hat{u}'_0, \hat{u}'_1, \hat{u}'_2, \hat{u}'_3]$, respectively. Since Super-Mix is linear, then we know that

$$N \cdot (\hat{u} + \hat{u}') = v + v' \;=\; [v_0 + v'_0 \;\; 0 \;\; 0 \;\; 0]$$

Therefore $\hat{u} + \hat{u}'$ (when viewed as a byte-vector) belongs to the code $\mathcal{C}_0$, that has minimum weight 13. In other words, and at least 13 of the bytes must differ between $\hat{u}$ and $\hat{u}'$ and at most three bytes can be the same. Since the S-box is a permutation, then the same applies also to the byte vectors u and u', namely they agree in at most 3 bytes. This implies in particular that all the four-byte words $u_0 + u'_0$, $u_1 + u'_1$, $u_2 + u'_2$, and $u_3 + u'_3$ must be non-zero.

Moreover, the above argument also implies that in the last **SMIX** step, the S-box was applied to at least 13 non-zero differences, yet the attacker was able to make the output difference in all these 13 (or more) non-zero positions somehow "hit" a codeword of $\mathcal{C}_0$.

Example 2: Counting zeros in the SMIX input. The argument from above about "hitting" a $\mathcal{C}_0$ codeword can be sharpened further: Suppose that the attacker was able to freely choose the input difference $u + u'$, how should it go about choosing this difference? In particular, how many bytes should u and u' agree on (from the attacker's perspective)?

The argument above says that $u + u'$ can have at most 3 zero bytes, but they can have less. A-priory it may seem that having u and u' agree on exactly three bytes would be the best choice for the attacker: Zero input difference is always mapped by the S-box to zero output difference, whereas obtaining any other input/output difference would take some work, so maximizing the number of zeros in the input sounds like a good strategy.

However, recall that the attacker does not need to "hit" one particular output difference of the S-box, all it needs is *any* codeword of $\mathcal{C}_0$. Moreover, from Table 3 we can get bounds on the number of $\mathcal{C}_0$ codeword that have zeros in any fixed set of positions. In particular, we get the following bounds:

- If u and u' agree on exactly three bytes (say, in positions i_1, i_2, i_3), then the attacker needs to "hit" $\mathcal{C}_0$ codewords that equal zero in these three positions. Looking up the entry in Table 3 corresponding to $\mathcal{C}_0$ and $m = 16 - 3 = 13$, we see that $\text{min-rank}_{13}(\mathcal{C}_0) = 12$. Applying Lemma 7.3 we get that dimension of these codewords is at most $13 - 12 = 1$. Hence there are at most 256 such codewords.
- If u and u' agree on exactly two bytes (say, in positions i_1, i_2), then the attacker needs to "hit" $\mathcal{C}_0$ codewords that equal zero in these two positions. Looking up the entry in Table 3 corresponding to $\mathcal{C}_0$ and $m = 16-2 = 14$, we see that $\text{min-rank}_{14}(\mathcal{C}_0) = 12$, so the dimension of these codewords is at most $14 - 12 = 2$. Hence there are at most 2^{16} such codewords.
- Similarly, if u and u' agree on exactly one byte then we check that $\text{min-rank}_{15}(\mathcal{C}_0) = 12$, and conclude that the attacker must "hit" a codeword taken from a space of dimension at most $15 - 12 = 3$, so there are at most 2^{24} such codewords.
- If u and u' disagree on all bytes then the dimension of codewords is 4 (which is the entire code) and we have 2^{32} codewords.

Given these bounds, we can now use the following heuristic to determine the best strategy for the attacker: since the best non-zero differential characteristic of the AES S-box has probability 2^{-6}, we assign "probability" of 2^{-6k} for hitting any particular codeword with k non-zero bytes. Hence by choosing three zero-bytes in $u + u'$ the attacker can get the "probability" of hitting a $\mathcal{C}_0$ codeword as high as $256 \times 2^{-6\times 13} = 2^{-70}$. By choosing a two-byte-zero difference it can be as much as $2^{16-6\times 14} = 2^{-68}$, one-byte-zero gives $2^{24-6\times 15} = 2^{-66}$, and all non-zero gives $2^{32-6\times 16} = 2^{-64}$. Hence the "optimal strategy" for the attacker may be to choose $u+u'$ to be non-zero everywhere.[4]

8 Diffusion Properties of Fugue-256

An important advantage of the **SMIX** transformation of Fugue over the AES round function is that the Super-Mix transformation entails stronger mixing than the Column-Mix of AES (since the four columns are not mixed independently). This results in very strong diffusion properties, on which we elaborate in this section. All the data in this section was obtained by running computer programs that track the diffusion of bytes through the execution of **F**-256.

8.1 Diffusion of input bytes

One consequence of the strong mixing in Fugue is that each input byte in Fugue propagates very quickly to the entire state. By the time that input word P_i is entered into **F**-256, the input bytes from P_{i-3} already influenced all the state bytes in a non-linear fashion.

This is illustrated in **Table 4**, where the diffusion of the third byte P^3 is depicted. (The byte P^3 has the slowest initial diffusion among the four input bytes). This table follows the influence of that input byte via three round functions (that include six **SMIX**-es). Before every **SMIX**, it records for each byte of the state whether or not it is influenced by that input byte, and moreover the "level of non-linearity" of that influence. Namely, if we write an expression for the value of that state byte in terms of the input byte P^3 (with all the other relevant bytes treated as constants), what is the largest level of nesting that the S-box[$\star$] appears in this expression. Equivalently, if we track all the paths through which the input byte P^3 influences that state-byte, what is the largest number of **SMIX**-es that appear on any of these paths. A blank entry means that P^3 has no influence over that state byte, a 0-entry means this state byte depends linearly on P^3 (i.e., $B = \mathsf{linear}(P^3)$), 1-entry means that $B = \mathsf{linear}(P^3) + \mathsf{linear}(\text{S-box}[\mathsf{linear}(\mathsf{P}^3)])$, etc.

Note the numbering scheme in this table, which we use often throughout this report: The last **SMIX** is always denoted SMIX[0], and this is the SMIX *just before* a **TIX** step. The **SMIX**-es just before the previous **TIX** steps are then denoted SMIX[−1], SMIX[−2], etc. The other **SMIX**-es (i.e., the ones *just after* the **TIX** steps) are denoted SMIX[−0.5], SMIX[−1.5], etc. The reason for this numbering scheme is that in our analysis we would often begin at the end of a (presumably successful) collision attack that ends with a **TIX** step, and then examine how the state evolves backward through the **SMIX**-es. We explicitly marked the location of the **TIX** steps to make the notations easier to follow.

One can observe that before every other **SMIX**, the column $\mathbf{S}_3$ of the state does not depend

---

[4] This argument is a bit simplistic, ignoring the fact that most non-zero input/output differences only have probability 2^{-7} or zero. It is possible to somewhat reduce the probabilities above using a more elaborate argument, but we do not know how to improve them by much.

Table 4: Diffusion of input byte P^3 in **F**-256

```
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
<TIX>Before SMIX[-3.5]
[0]
[1]
[2]
[3]           0                       0
Before SMIX[-3]
[0]        1           1                                1
[1]     1           1                                1
[2]  1           1                                1
[3]           1                                0
<TIX>before SMIX[-2.5]
[0]  2  2  2     2  2  2        1           2     2  2  2        1
[1]  2  2  2     2  2  2     1              2     2  2  2     1
[2]  2  2  2     2  2  2  1                 2     2  2  2  1
[3]  2  2  2     2  2  2                    2     2  2  2
before SMIX[-2]
[0]  3  3  3  3  3  3  3  2  2  2        1        3  3  3  2  2  2        1
[1]  3  3  3  3  3  3  3  2  2  2     1           3  3  3  2  2  2     1
[2]  3  3  3  3  3  3  3  2  2  2  1              3  3  3  2  2  2  1
[3]  3  3  3  3  3  3  3  2  2  2                 3  3  3  2  2  2
<TIX>before SMIX[-1.5]
[0]  4  4  4     4  4  4  3  3  3  2  2  2  4     4  4  4  3  3  3  2  2  2        1
[1]  4  4  4     4  4  4  3  3  3  2  2  2  4  1  4  4  4  3  3  3  2  2  2     1
[2]  4  4  4     4  4  4  3  3  3  2  2  2  4     4  4  4  3  3  3  2  2  2  1
[3]  4  4  4     4  4  4  3  3  3  2  2  2  4     4  4  4  3  3  3  2  2  2
before SMIX[-1]
[0]  5  5  5  5  5  5  5  4  4  4  3  3  3  2  2  5  5  5  4  4  4  3  3  3  2  2  2        1
[1]  5  5  5  5  5  5  5  4  4  4  3  3  3  2  2  5  5  5  4  4  4  3  3  3  2  2  2     1
[2]  5  5  5  5  5  5  5  4  4  4  3  3  3  2  2  5  5  5  4  4  4  3  3  3  2  2  2  1
[3]  5  5  5  5  5  5  5  4  4  4  3  3  3  2  2  5  5  5  4  4  4  3  3  3  2  2  2
<TIX>before SMIX[-0.5]
[0]  6  6  6     6  6  6  5  5  5  4  4  4  6  3  6  6  6  5  5  5  4  4  4  3  3  3  2  2  2
[1]  6  6  6     6  6  6  5  5  5  4  4  4  6  3  6  6  6  5  5  5  4  4  4  3  3  3  2  2  2
[2]  6  6  6     6  6  6  5  5  5  4  4  4  6  3  6  6  6  5  5  5  4  4  4  3  3  3  2  2  2
[3]  6  6  6     6  6  6  5  5  5  4  4  4  6  3  6  6  6  5  5  5  4  4  4  3  3  3  2  2  2
before SMIX[0]
[0]  7  7  7  7  7  7  7  6  6  6  5  5  5  4  4  7  7  7  6  6  6  5  5  5  4  4  4  3  3  3
[1]  7  7  7  7  7  7  7  6  6  6  5  5  5  4  4  7  7  7  6  6  6  5  5  5  4  4  4  3  3  3
[2]  7  7  7  7  7  7  7  6  6  6  5  5  5  4  4  7  7  7  6  6  6  5  5  5  4  4  4  3  3  3
[3]  7  7  7  7  7  7  7  6  6  6  5  5  5  4  4  7  7  7  6  6  6  5  5  5  4  4  4  3  3  3
<TIX>
     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
```

Recorded for each state byte is the largest number of **SMIX**-es through which it is influenced by the input P^3.

on the input byte that we track. (Indeed that column is just set to equal the newer input word from that round.) On the other hand, all the other input bytes into the **SMIX**-es are always influenced by every input byte (with a monotonically increasing "level of non-linearity").

8.2 Diffusion of state bytes

A related measure of diffusion is how state bytes influence each other through the round function. This is tracked in **Table 5**. Starting from the state of **F**-256 just prior to an input word, this table tracks how many of these "initial state bytes" influence (in a non-linear fashion) each byte of the state at some later time. Namely, consider freezing the execution just after some **SMIX**, and focusing on one particular byte of the state. The value of that byte can be written as some expression involving all the "initial state bytes" and the input bytes that came later, and we ask how many of these "initial state bytes" appear in the expression inside at least one S-box[$\star$]. Equivalently, if we we track all the paths through which all these "initial state bytes" influences the state-byte that we focus on, then how many of these initial bytes have paths of influence that go through at least one **SMIX**.

Again, we can observe that the bytes that are output by an SMIX, are influenced by a rapidly increasing number of the "initial state bytes". Specifically, considering the **SMIX** transformation just prior to **TIX**(P_i), the bytes that are output by this **SMIX** depend (non-linearly) on:

- 14 or more of the bytes from the state just prior to **TIX**(P_{i-1}),
- 41 or more of the bytes from the state just prior to **TIX**(P_{i-2}),
- 65 or more of the bytes from the state just prior to **TIX**(P_{i-3}),
- 89 or more of the bytes from the state just prior to **TIX**(P_{i-4}),
- 109 or more of the bytes from the state just prior to **TIX**(P_{i-5}).

8.3 Diffusion in the TIX-less rounds G1

When analyzing "external collisions" (cf. Section 10) and the properties of Fugue as a PRF, it is also important to consider the diffusion properties of the final transformation **G**. We begin by looking at the first phase of that final transformation, namely the five "TIX-less" rounds.

These rounds are tracked in **Table 6**. Starting right *after* the last **TIX** step and ending after the end of the **G**1 transformation (i.e., after the five TIX-less rounds), this table records how many "initial bytes" influence each of the "final bytes" (after **G**1), and with what "level of non-linearity." Specifically, the final bytes are listed column by column, and for each byte there are three numbers: First is the number of initial bytes that this final byte depends on in a non-linear fashion (i.e., via at least one **SMIX**). Next is the number of initial bytes that this final byte depends on via at least five **SMIX**-es, and finally the number of initial bytes that it depends on

Table 5: Diffusion statistics of the round transformations **R** in **F**-256

```
After SMIX[-4.5]
   6   5   5   2   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   6   5   2   5   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   6   2   5   5   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   3   6   6   6   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
After SMIX[-4.0]
  18  17  17  14   5   5   2   0   0   0   0   0   0   0   0   5   5   2   0   0   0   0   0   0   0   0   0   0   0   0
  18  17  14  17   5   2   5   0   0   0   0   0   0   0   0   5   2   5   0   0   0   0   0   0   0   0   0   0   0   0
  18  14  17  17   2   5   5   0   0   0   0   0   0   0   0   2   5   5   0   0   0   0   0   0   0   0   0   0   0   0
  15  18  18  18   6   6   6   0   0   0   0   0   0   0   0   6   6   6   0   0   0   0   0   0   0   0   0   0   0   0
After SMIX[-3.5]
  34  29  29  19  17  17  14   5   5   2   0   0   0  18   0  17  17  14   5   5   2   0   0   0   0   0   0   0   0   0
  31  30  23  33  17  14  17   5   2   5   0   0   0  18   0  17  14  17   5   2   5   0   0   0   0   0   0   0   0   0
  31  20  33  30  14  17  17   2   5   5   0   0   0  18   0  14  17  17   2   5   5   0   0   0   0   0   0   0   0   0
  28  34  31  31  18  18  18   6   6   6   0   0   0  15   0  18  18  18   6   6   6   0   0   0   0   0   0   0   0   0
After SMIX[-3.0]
  42  45  45  42  29  29  19  17  17  14   5   5   2   0   0  29  29  19  17  17  14   5   5   2   0   0   0   0   0   0
  45  44  41  41  30  23  33  17  14  17   5   2   5   0   0  30  27  33  17  14  17   5   2   5   0   0   0   0   0   0
  45  41  41  44  20  33  30  14  17  17   2   5   5   0   0  20  33  30  14  17  17   2   5   5   0   0   0   0   0   0
  42  42  45  45  34  31  31  18  18  18   6   6   6   0   0  34  31  31  18  18  18   6   6   6   0   0   0   0   0   0
After SMIX[-2.5]
  58  53  53  47  45  45  42  29  29  19  17  17  14  42   5  45  45  42  29  29  19  17  17  14   5   5   2   0   0   0
  55  54  47  57  44  41  41  30  23  33  17  14  17  45   2  44  41  41  30  27  33  17  14  17   5   2   5   0   0   0
  55  44  57  54  41  41  44  20  33  30  14  17  17  45   5  41  41  44  20  33  30  14  17  17   2   5   5   0   0   0
  52  58  55  55  42  45  45  34  31  31  18  18  18  42   6  42  45  45  34  31  31  18  18  18   6   6   6   0   0   0
After SMIX[-2.0]
  66  69  69  66  53  53  47  45  45  42  29  29  19  17  17  53  53  47  45  45  42  29  29  19  17  17  14   5   5   2
  69  68  65  65  54  47  57  44  41  41  30  23  33  17  14  54  51  57  44  41  41  30  27  33  17  14  17   5   2   5
  69  65  65  68  44  57  54  41  41  44  20  33  30  14  17  44  57  54  41  41  44  20  33  30  14  17  17   2   5   5
  66  66  69  69  58  55  55  42  45  45  34  31  31  18  18  58  55  55  42  45  45  34  31  31  18  18  18   6   6   6
After SMIX[-1.5]
  82  77  77  71  69  69  66  53  53  47  45  45  42  66  29  69  69  66  53  53  47  45  45  42  29  29  19  17  17  14
  79  78  71  81  68  65  65  54  47  57  44  41  41  69  23  68  65  65  54  51  57  44  41  41  30  27  33  17  14  17
  79  68  81  78  65  65  68  44  57  54  41  41  44  69  33  65  65  68  44  57  54  41  41  44  20  33  30  14  17  17
  76  82  79  79  66  69  69  58  55  55  42  45  45  66  31  66  69  69  58  55  55  42  45  45  34  31  31  18  18  18
After SMIX[-1.0]
  90  93  93  90  77  77  71  69  69  66  53  53  47  45  45  77  77  71  69  69  66  53  53  47  45  45  42  29  29  19
  93  92  89  89  78  71  81  68  65  65  54  47  57  44  41  78  75  81  68  65  65  54  51  57  44  41  41  30  27  33
  93  89  89  92  68  81  78  65  65  68  44  57  54  41  41  68  81  78  65  65  68  44  57  54  41  41  44  20  33  30
  90  90  93  93  82  79  79  66  69  69  58  55  55  42  45  82  79  79  66  69  69  58  55  55  42  45  45  34  31  31
After SMIX[-0.5]
 102 101 101  95  93  93  90  77  77  71  69  69  66  90  53  93  93  90  77  77  71  69  69  66  53  53  47  45  45  42
 102 101  94 101  92  89  89  78  71  81  68  65  65  93  47  92  89  89  78  75  81  68  65  65  54  51  57  44  41  41
 102  91 101 101  89  89  92  68  81  78  65  65  68  93  57  89  89  92  68  81  78  65  65  68  44  57  54  41  41  44
  99 102 102 102  90  93  93  82  79  79  66  69  69  90  55  90  93  93  82  79  79  66  69  69  58  55  55  42  45  45
After SMIX[0]
 110 113 113 110 101 101  95  93  93  90  77  77  71  69  69 101 101  95  93  93  90  77  77  71  69  69  66  53  53  47
 113 112 109 109 101  94 101  92  89  89  78  71  81  68  65 101  98 101  92  89  89  78  75  81  68  65  65  54  51  57
 113 109 109 112  91 101 101  89  89  92  68  81  78  65  65  91 101 101  89  89  92  68  81  78  65  65  68  44  57  54
 110 110 113 113 102 102 102  90  93  93  82  79  79  66  69 102 102 102  90  93  93  82  79  79  66  69  69  58  55  55
```

Recorded for each state-byte is the number of “initial bytes” with non-linear influence on that state-byte.

Table 6: Diffusion statistics of five TIX-less rounds in **F**-256

```
      byte 0        byte 1        byte 2        byte 3
 col   >0  >4  >9    >0  >4  >9    >0  >4  >9    >0  >4  >9
 ---  --- --- ---   --- --- ---   --- --- ---   --- --- ---
[ 0] (120 100  40) (120 100  40) (120 100  40) (120 100  40)
[ 1] (120 100  40) (120 100  40) (120 100  40) (120 100  40)
[ 2] (120 100  40) (120 100  40) (120 100  40) (120 100  40)
[ 3] (120 100  40) (120 100  40) (120 100  40) (120 100  40)
[ 4] (120 100  40) (120 100  40) (120 100  40) (120 100  40)
[ 5] (120  88  28) (120  88  28) (120  88  28) (120  88  28)
[ 6] (120  88  28) (120  88  28) (120  88  28) (120  88  28)
[ 7] (117  76  16) (116  76  16) (113  76  16) (114  76  16)
[ 8] (117  76  16) (113  76  16) (113  76  16) (117  76  16)
[ 9] (114  76  16) (113  76  16) (116  76  16) (117  76  16)
[10] (105  64   0) (105  64   0) (102  64   0) (106  64   0)
[11] (105  64   0) (102  64   0) (105  64   0) (106  64   0)
[12] (102  64   0) (105  64   0) (105  64   0) (106  64   0)
[13] ( 93  52   0) ( 93  52   0) ( 90  52   0) ( 94  52   0)
[14] ( 93  52   0) ( 90  52   0) ( 93  52   0) ( 94  52   0)
[15] (120 100  40) (120 100  40) (120 100  40) (120 100  40)
[16] (120  88  28) (120  88  28) (120  88  28) (120  88  28)
[17] (120  88  28) (120  88  28) (120  88  28) (120  88  28)
[18] (117  76  16) (116  76  16) (113  76  16) (114  76  16)
[19] (117  76  16) (113  76  16) (113  76  16) (117  76  16)
[20] (114  76  16) (113  76  16) (116  76  16) (117  76  16)
[21] (105  64   0) (105  64   0) (102  64   0) (106  64   0)
[22] (105  64   0) (102  64   0) (105  64   0) (106  64   0)
[23] (102  64   0) (105  64   0) (105  64   0) (106  64   0)
[24] ( 93  52   0) ( 93  52   0) ( 90  52   0) ( 94  52   0)
[25] ( 93  52   0) ( 90  52   0) ( 93  52   0) ( 94  52   0)
[26] ( 90  52   0) ( 93  52   0) ( 93  52   0) ( 94  52   0)
[27] ( 81  40   0) ( 81  40   0) ( 78  40   0) ( 82  40   0)
[28] ( 81  40   0) ( 78  40   0) ( 81  40   0) ( 82  40   0)
[29] ( 78  40   0) ( 81  40   0) ( 81  40   0) ( 82  40   0)
```

Recorded for each entry are the number of bytes that influence it through at least one, five, and ten **SMIX**-es.

via at least ten **SMIX**-es. (Note that ten **SMIX**-es is the "level of non-linearity" that one gets from a full application of AES-128.) Some notable features that can be seen from that table are:

- Every "final byte" depends non-linearly on at least 78 of the "initial bytes", and also depends on at least 40 "initial bytes" via five or more **SMIX**-es.
- Every "final byte" in columns 0-4 (and 15) depends non-linearly on all the 120 "initial bytes", and moreover it depends on 100 of them via at least five **SMIX**-es and on 40 of them via at least ten **SMIX**-es.

8.4 Diffusion in the entire final transformation G

We also consider the diffusion of the entire final transformation, including both the TIX-less rounds **G**1 and the final rounds **G**2. For the full **F**-256 (with five TIX-less rounds and 13 final rounds) we obtain that every byte in the output of **F**-256 depends on all the 120 "initial bytes" (after the last **TIX**) via more than 25 **SMIX**-es (which is about the "level of non-linearity" in double-AES-256).

Finally, we consider the diffusion of the final transformation in the weak variant w**F**-256, as summarized in **Table 7**. That table lists two types of statistics, tracking both the "influenced-by" statistics (how many "initial bytes" influence each output byte) and the "influencing" statistics (how many output bytes are influenced by each initial byte). For each of these statistics, the bytes are listed column by column, and for each byte there are four numbers: First is the number bytes with "influence paths" that goes through at least one **SMIX**, then the number bytes with "influence paths" that goes through at least five **SMIX**-es, then ten **SMIX**-es and then fifteen **SMIX**-es.

As seen in this table, the input bytes from the last **TIX** step (in column [3] of the "influencing" table) effect each output byte via at least ten **SMIX**-es (which is the "level of non-linearity" in AES-128), and they effect all but two of the output bytes via at least fifteen **SMIX**-es. Moreover, even for this weak variant we have that most state bytes (all except columns 4-11) influence every output byte through at least five **SMIX**-es.

9 Properties of the S-Box

Recall from Section 4, that the S-Box used in Fugue is identical to that used in AES [16]. Moreover, the AES S-Box derives its non-linear properties from the multiplicative inverse function over GF(2^8), whose differential and linear characteristics were studied in [15].

Consider the mapping inv($\mathbf{x}$), from $(\mathrm{GF}2)^8$ to $(\mathrm{GF}2)^8$, by mapping 0 to 0, and otherwise first viewing $\mathbf{x}$ to be in GF(2^8) (as in Section 3), and then taking multiplicative inverse in GF(2^8), followed by the isomorphism back to $(\mathrm{GF}2)^8$. In [15], Nyberg shows that the mapping inv is differentially 4-uniform. In other words, for every pair of differences α, and β, $\alpha \neq 0$, the

Table 7: Diffusion statistics of the final transformation **G** in w**F**-256

```
      byte 0           byte 1           byte 2           byte 3
 col  >0  >4  >9 >14   >0  >4  >9 >14   >0  >4  >9 >14   >0  >4  >9 >14
 ---  -- --- --- ---   -- --- --- ---   -- --- --- ---   -- --- --- ---
[ 1] (88  88  88  28) (89  88  88  40) (89  88  88  40) (92  88  88  40)
[ 2] (88  88  88  28) (89  88  88  40) (92  88  88  40) (92  88  88  40)
[ 3] (88  88  88  28) (92  88  88  40) (89  88  88  40) (92  88  88  40)
[ 4] (92  88  88  40) (89  88  88  40) (89  88  88  40) (92  88  88  40)
...
[15] (92  88  88  40) (89  88  88  40) (92  88  88  40) (92  88  88  40)
[16] (89  88  76   0) (92  88  82  28) (89  88  88  28) (88  88  79  28)
[17] (92  88  76   0) (89  88  88  28) (89  88  82  28) (88  88  79  28)
[18] (89  88  88  28) (89  88  82  28) (89  88  82  28) (92  88  79  28)
```

Influenced-by statistics: Recorded for each output byte are the number of “initial bytes” that influence it through at least one, five, ten, and fifteen **SMIX**-es.

```
      byte 0           byte 1           byte 2           byte 3
 col  >0  >4  >9 >14   >0  >4  >9 >14   >0  >4  >9 >14   >0  >4  >9 >14
 ---  -- --- --- ---   -- --- --- ---   -- --- --- ---   -- --- --- ---
[ 0] (32  32  32  30) (32  32  32  30) (32  32  32  30) (32  32  32  30)
[ 1] (32  32  32  30) (32  32  32  30) (32  32  32  30) (32  32  32  30)
[ 2] (32  32  32  30) (32  32  32  30) (32  32  32  30) (32  32  32  30)
[ 3] (32  32  32  30) (32  32  32  30) (32  32  32  30) (32  32  32  30)
[ 4] (15   0   0   0) (19   0   0   0) (19   0   0   0) (13   0   0   0)
[ 5] ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0)
[ 6] ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0)
[ 7] ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0)
[ 8] ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0)
[ 9] ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0)
[10] ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0)
[11] ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0) ( 0   0   0   0)
[12] (32  32  23   0) (32  32  25   0) (32  32  25   0) (32  32  26   0)
[13] (32  32  25   0) (32  32  25   0) (32  32  27   0) (32  32  28   0)
[14] (32  32  25   0) (32  32  27   0) (32  32  25   0) (32  32  28   0)
[15] (32  32  32   0) (32  32  32   0) (32  32  32   0) (32  32  32   0)
[16] (32  32  32   0) (32  32  32   0) (32  32  32   0) (32  32  32   0)
[17] (32  32  32   0) (32  32  32   0) (32  32  32   0) (32  32  32   0)
[18] (32  32  32   0) (32  32  32   0) (32  32  32   0) (32  32  32   0)
[19] (32  32  32   0) (32  32  32   0) (32  32  32   0) (32  32  32   0)
[20] (32  32  32   0) (32  32  32   0) (32  32  32   0) (32  32  32   0)
[21] (32  32  32   0) (32  32  32   0) (32  32  32   0) (32  32  32   0)
[22] (32  32  32   0) (32  32  32   0) (32  32  32   0) (32  32  32   0)
[23] (32  32  32   0) (32  32  32   0) (32  32  32   0) (32  32  32   0)
[24] (32  32  32  17) (32  32  32  17) (32  32  32  17) (32  32  32  17)
[25] (32  32  32  17) (32  32  32  17) (32  32  32  17) (32  32  32  17)
[26] (32  32  32  17) (32  32  32  17) (32  32  32  17) (32  32  32  17)
[27] (32  32  32  30) (32  32  32  30) (32  32  32  30) (32  32  32  30)
[28] (32  32  32  30) (32  32  32  30) (32  32  32  30) (32  32  32  30)
[29] (32  32  32  30) (32  32  32  30) (32  32  32  30) (32  32  32  30)
```

Influencing statistics: Recorded for each “initial byte” are the number of output bytes that it influences through at least one, five, ten, and fifteen **SMIX**-es.

probability of obtaining the output difference β, on input difference α is at most $4/2^8$, i.e.

$$\Pr_{a,b \in (\mathrm{GF2})^8} [\, (a \oplus b = \alpha) \ \wedge \ (\mathrm{inv}(a) \oplus \mathrm{inv}(b) = \beta) \,] \leq 4 \cdot 2^{-8}$$

In fact, the equality is obtained exactly when α and β are multiplicative inverses of each other in $\mathrm{GF}(2^8)$. Otherwise, the probability is at most 2^{-7}.

Carlitz and Uchiyama [4] have earlier shown that the distance between any mapping which is a composition of the inv mapping above followed by any linear transformation over GF2, from any linear mapping over GF2 is at least $2^7 - 2^4$. It follows that, for all $\alpha, \beta \in (\mathrm{GF2})^8$, $\beta \neq 0$,

$$\Pr_{a \in (\mathrm{GF2})^8} [\, (\alpha \cdot a) \oplus (\beta \cdot \mathrm{inv}(a)) = 0 \,] < \frac{1}{2} + 2^{-4}$$

10 Differential Analysis of Fugue-256: Internal Collisions

One of the most appealing properties of Fugue is that it provably resists differential attacks. Moreover, under some very plausible assumptions it can even be proven to resist attacks that employ message-modification techniques and control/neutral byte analysis. In this section and the next we detail these attacks and our analysis that rule them out.

There are two types of collision-finding attacks that one can envision against Fugue, namely *internal collision* attacks and *external collision* attacks. Internal collisions refer to attacks that find two different messages $P \neq P'$ such that the internal state of Fugue after processing P is the same as the internal state after processing P'. External collisions refer to attacks where the internal states do not collide, but the outputs of the function do. In this section we analyze internal collisions, and in the next we tackle external collisions.

We begin in Section 10.1 by describing the "backward evolution" of the internal state of Fugue, starting from the end of a successful internal-collision attack. This analysis exhibits many conditions that must be satisfied in order for an internal-collision attack to be successful. These conditions are summarized in **Tables 8 and 9**, and they are used in later sections to lower-bound the complexity of any such attack.

10.1 Backward Evolution of the Internal State

Consider two different inputs $P \neq P'$, each consisting of an integral number of 4-byte words, that result in an internal collision of **F**-256. We can assume without loss of generality that the internal collision only occurs after processing the last words of P, P' (else we can just consider the prefixes that lead to the first internal collision and disregard the suffixes from there on).

Notations. Below we denote the words that comprise these two messages by $P[-m], \ldots, P[-1]$, $P[0]$ and $P'[-m'], \ldots, P'[-1], P'[0]$, respectively. Namely, the *last words* in these messages are denoted $P[0], P'[0]$, respectively, the words before that $P[-1], P'[-1]$, and so on. For simplicity, we assume below that $m, m' \geq 4$, i.e., both messages have at least five words in them.[5]

The round transformation that processes the last word ($P[0]$ or $P'[0]$) is called *round 0*, and in general the round during which the word $P[-i]$ (or $P'[-i]$) is processed is called *round* $-i$. We denote the 30-word states at the end of round 0 in the two executions by $\mathbf{S}(P)[0]$, and $\mathbf{S}(P')[0]$, respectively. Similarly, the states at the end of round $-i$ (i.e., just before the **TIX** step that takes the words $P[-i+1], P'[-i+1]$) are denoted $\mathbf{S}(P)[-i]$, and $\mathbf{S}(P')[-i]$, respectively.

Further, the xor-difference of the two messages is denoted ΔP and the xor-difference of the states is denoted $\Delta\mathbf{S}$, with indexes as before. For example the difference between the last two words is $\Delta P[0] = P[0] \oplus P'[0]$, and the difference of states after round -1 (i.e., before the last **TIX** step) is denoted $\Delta\mathbf{S}[-1] = \mathbf{S}(P')[-1] \oplus \mathbf{S}(P')[-1]$, etc. We use subscripts to denote specific columns of the state. For example, $\Delta\mathbf{S}[-1]_5$ denotes the difference in column number 5 of the state at the end of round -1. We also use the following notations for the states within a round. Consider for example the steps in round -1:

- The state *after the* **TIX** *step* of round -1 is denoted $\mathbf{S}(P)[-2+\text{tix}]$;
- The state after the first **CMIX** is $\mathbf{S}(P)[-2+c1]$;
- The state after the first **SMIX** is $\mathbf{S}(P)[-2+s1]$;
- The state after the second **CMIX** is $\mathbf{S}(P)[-2+c2]$;
- The state after the second **SMIX** is $\mathbf{S}(P)[-2+s2]$ (which is the same as $\mathbf{S}(P)[-1]$).

We use the same notations also for $\mathbf{S}(P')$ as well as $\Delta\mathbf{S}$. For example, the state difference after the last **TIX** step (in round 0) is denoted $\Delta\mathbf{S}[-1+\text{tix}]$.

10.1.1 Round Zero

Recall that we assume that the input words P, P' cause an internal collision at round zero. Clearly, this collision can only occurs at the **TIX** step (since the rest of the round is a permutation). In our notations, this means that $\Delta\mathbf{S}[-1+\text{tix}] = 0$ but $\Delta\mathbf{S}[-1] \neq 0$. Recall now the definition of the **TIX** step for **F**-256:

$$\mathbf{TIX}(P) : \{\ \mathbf{S}_{10} += \mathbf{S}_0; \quad \mathbf{S}_0 += P; \quad \mathbf{S}_8 += \mathbf{S}_0; \quad \mathbf{S}_1 += \mathbf{S}_{24}\ \}$$

---

[5] This assumption is only made so that we can talk about the state before processing $P[-4], P'[-4]$. We can eliminate the assumption by prepending the same sequence of arbitrary words to both these messages, and starting the computation from the state that one gets by evolving the initial state backward with that sequence of words.

Hence we have:

$$\begin{aligned}
\Delta\mathbf{S}[-1+\mathrm{tix}]_{10} &= \Delta\mathbf{S}[-1]_{10} + \Delta\mathbf{S}[-1]_0 \\
\Delta\mathbf{S}[-1+\mathrm{tix}]_0 &= \Delta P[0] \\
\Delta\mathbf{S}[-1+\mathrm{tix}]_8 &= \Delta\mathbf{S}[-1]_8 + \Delta P[0] \\
\Delta\mathbf{S}[-1+\mathrm{tix}]_1 &= \Delta\mathbf{S}[-1]_1 + \Delta\mathbf{S}[-1]_{24} \\
\Delta\mathbf{S}[-1+\mathrm{tix}]_i &= \Delta\mathbf{S}[-1]_i \quad \text{for all other columns}
\end{aligned}$$

Since $\Delta\mathbf{S}[-1+\mathrm{tix}] = 0$ but $\Delta\mathbf{S}[-1] \neq 0$, then for $\Delta\mathbf{S}[-1]$ we must have:

- $\Delta\mathbf{S}[-1]_{10} = \Delta\mathbf{S}[-1]_0 \neq 0$,
- $\Delta P[0] = 0$, and
- $\Delta\mathbf{S}[-1+\mathrm{tix}]_i = 0$ for all other columns.

Below we denote the non-zero difference in columns 0 and 10 by $X1 = \Delta\mathbf{S}[-1]_0 = \Delta\mathbf{S}[-1]_{10}$.

10.1.2 Introducing Tables 8 and 9

The argument from above about round 0 is illustrated in the top two lines of Table 8. That Table (and the subsequent Table 9) further describe the backward evolution of the difference in the state, with differences in individual words of the state named as follows:

- Blank entries imply zero difference. (We explicitly write a zero difference in places where we want to stress it.)
- Variables denoting differences that are necessarily non-zero are written in capital (such as $X1$ from above). Variables that can be either zero or non-zero are written in lowercase.
- Since each round transformation **R** in Fugue-256 has two sub-rounds, we use y variables for the difference $\Delta\mathbf{S}_{0..3}$ before the second **SMIX** in a round (or Y variables if they are necessarily non-zero).

  For example, the input differences to the second **SMIX** in round -1 are denoted $Y1_0$ through $Y1_3$. (We show below that all these differences must indeed be non-zero.)
- Similarly, we use z variables (or Z variables) to denote the difference $\Delta\mathbf{S}_{0..3}$ before the first **SMIX** in a round.

  Note that due to the **ROR3** step between the **TIX** and the first **SMIX**, then the input difference $\Delta P[-i]$ is always equal to zi_3 (or Zi_3 when it must be non-zero).
- The difference $\Delta\mathbf{S}_0[-i]$ is denoted by xi (or Xi when it must be non-zero).

Table 8: Evolution of Differential State for Internal Collision (Backwards)

	0			3			6			9			12			15			18			21			24			27		29
	0																													
TIX$_0$																														
$\Delta S[-1]$	$X1$	0	0	0							$X1$																			
SMIX																														
	$Y1_0$	$Y1_1$	$Y1_2$	$Y1_3$							$X1$																			
CMIX																														
	$Y1_0$	$Y1_1$	$Y1_2$	$Y1_3$							$X1$																			
ROR3																														
	$\boxed{Y1_3}$	0	0	0				$X1$																				$Y1_0$	$Y1_1$	$Y1_2$
SMIX																														
	$Z1_0$	$Z1_1$	$Z1_2$	$Z1_3$				$X1$																				$Y1_0$	$Y1_1$	$Y1_2$
CMIX																														
	$Z1_0$	$Z1_1$	$Z1_2$	$Z1_3$				$X1$																				$Y1_0$	$Y1_1$	$Y1_2$
ROR3																														
	$Z1_3$	0	0	0	$X1$																				$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$
TIX$_{-1}$																														
	0			3			6			9			12			15			18			21			24			27		29
$\Delta S[-2]$	$x2$	$Y1_0$	0	0	$X1$				$Z1_3$	0	$x2$														$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$
SMIX																														
	$y2_0$	$y2_1$	$y2_2$	$y2_3$	$X1$				$Z1_3$	0	$x2$														$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$
CMIX																														
	$y2'_0$	$y2_1$	$y2_2$	$y2_3$	$X1$				$Z1_3$	0	$x2$					$X1$									$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$
ROR3																														
	$\boxed{y2_3}$	$X1$	0	0	0	$Z1_3$	0	$x2$	0				$X1$									$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$
SMIX																														
	$z2_0$	$z2_1$	$z2_2$	$z2_3$	0	$Z1_3$	0	$x2$	0				$X1$									$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$
CMIX																														
	$z2_0$	$z2'_1$	$z2_2$	$z2_3$	0	$Z1_3$	0	$x2$	0				$X1$				$Z1_3$					$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$
ROR3																														
	$z2_3$	0	$Z1_3$	0	$x2$					$X1$			0	$Z1_3$					$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$
TIX††																														

[a] All *blank* cells are zero. Primed variables are defined in Section 10.1.4. The *shaded* cells are the ones affected in that step. The *boxed* variables are the ones that are *not determined* by variables from earier (lower) steps. Variables that are necessarily *non-zero* are in *capital*. Rounds are referred to by the subscript on the TIX step for that round. †† Continued on next page.

Table 9: Evolution of Differential State for internal Collision (contd.)

	0			3			6			9			12			15			18			21			24			27		29
	$z2_3$	0	$Z1_3$	0	$x2$					$X1$				$Z1_3$					$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$
TIX$_{-2}$																														
$\Delta S[-3]$	$x3$	$y2'_0$	$Z1_3$	0	$x2$				$z2_3{}^{\dagger}$	$X1$	$x3$			$Z1_3$					$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$
SMIX																														
	$y3_0$	$y3_1$	$y3_2$	$y3_3$	$x2$				$z2_3$	$X1$	$x3$			$Z1_3$					$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$
CMIX																														
	$y3'_0$	$y3_1$	$y3_2$	$y3_3$	$x2$				$z2_3$	$X1$	$x3$			$Z1_3$	0	$x2$			$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$
ROR3																														
	$\boxed{y3_3}$	$x2$	0	0		$z2_3$	$X1$	$x3$			$Z1_3$	0	$x2$			$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$
SMIX																														
	$z3_0$	$z3_1$	$z3_2$	$z3_3$		$z2_3$	$X1$	$x3$			$Z1_3$	0	$x2$			$Y1_0$	$Y1_1$	$Y1_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$
CMIX																														
	$z3_0$	$z3'_1$	$z3'_2$	$z3_3$		$z2_3$	$X1$	$x3$			$Z1_3$	0	$x2$			$Y1_0$	$y1'_1$	$y1'_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$
ROR3																														
	$z3_3$	0	$z2_3$	$X1$	$x3$	0	0	$Z1_3$	0	$x2$	0	0	$Y1_0$	$y1'_1$	$y1'_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$
TIX$_{-3}$																														
$\Delta S[-4]$	$x4$	$y3'_0$	$\boxed{z2_3 \mid X1}$		$x3$	0	0	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$
SMIX																														
	$y4_0$	$y4_1$	$y4_2$	$y4_3$	$x3$	0	0	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$Z1_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$
CMIX																														
	$y4'_0$	$y4_1$	$y4_2$	$y4_3$	$x3$	0	0	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$z1'_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$
ROR3																														
	$\boxed{y4_3 \mid x3}$		0	0	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$z1'_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$	$y4'_0$	$y4_1$	$y4_2$
SMIX																														
	$z4_0$	$z4_1$	$z4_2$	$z4_3$	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$z1'_0$	$Z1_1$	$Z1_2$	$y2'_0$	$y2_1$	$y2_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$	$y4'_0$	$y4_1$	$y4_2$
CMIX																														
	$z4'_0$	$z4'_1$	$z4'_2$	$z4_3$	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$z1'_0$	$Z1_1$	$Z1_2$	$y2''_0$	$y2'_1$	$y2'_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$	$y4'_0$	$y4_1$	$y4_2$
ROR3																														
	$\boxed{z4_3}$	$Z1_3$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$	$z1'_0$	$Z1_1$	$Z1_2$	$y2''_0$	$y2'_1$	$y2'_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$	$y4'_0$	$y4_1$	$y4_2$	$z4'_0$	$z4'_1$	$z4'_2$
TIX$_{-4}$																														
$\Delta S[-5]$	$x5$	$Z1_3$ $+$ $y4'_0$	$z3_3$	$x2$	$x4$	0	$Y1_0$	$y1'_1$	$y1'_2$ $+$ $z4_3$	$z1'_0$	$Z1_1$ $+$ $x5$	$Z1_2$	$y2''_0$	$y2'_1$	$y2'_2$	$z2_0$	$z2'_1$	$z2_2$	$y3'_0$	$y3_1$	$y3_2$	$z3_0$	$z3'_1$	$z3'_2$	$y4'_0$	$y4_1$	$y4_2$	$z4'_0$	$z4'_1$	$z4'_2$
	0			3			6			9			12			15			18			21			24			27		29

[a] Primed variables are defined in Sections 10.1.5 and 10.1.6.

10.1.3 Round -1

Recall that the difference in $\Delta\mathbf{S}[-1]$ is non-zero only in $\mathbf{S}_0$ and $\mathbf{S}_{10}$. Thus, the difference in columns 0 to 3 of $\mathbf{S}$ after the last **SMIX** in round -1 is (X_1,0,0,0), and these are the columns that are affected by this **SMIX**.

Recall now Example 1 from Section 7.3: Since the output difference of this **SMIX** has the last three columns zero, then the difference before the Super-Mix (but after the S-box transformation) must be in the code $\mathcal{C}_0$, which has minimum weight 13. This means that at most three bytes after the S-box have difference zero, and with the S-box being a permutation it also means that at most three bytes at the input to the **SMIX**have difference zero. We denote these four columns differences at the input to this **SMIX** by $Y1_0$ through $Y1_3$, and since there are at most 3 zero bytes among these 16 bytes then none of these column differences can be all-zero (and hence they are all written in uppercase).

Continuing backward, the **ROR3** steps implies that difference in columns 0 to 3 of $\mathbf{S}$ after the first **SMIX** of round -1 is $\Delta\mathbf{S}[-1+s1] = (Y1_3\ 0\ 0\ 0)$. Repeating the same argument as above, it again means that all four input columns to this **SMIX** step must have non-zero differences, and we denote them in uppercase $Z1_0$ through $Z1_3$.

Proceeding backward through the **TIX** step at the beginning of round -1, the difference $\Delta P[-1]$ must equal the variable $Z1_3$ (and hence must be non-zero). Also, since the difference in column 10 after the **TIX** step is zero, then again it must be the case that $\Delta\mathbf{S}[-2]_0 = \Delta\mathbf{S}[-2]_{10}$, and we denote that difference by $x2$. As opposed to round 0, however, here we cannot claim that this difference must be non-zero (thus we write it in lowercase).

However, since column 24 is XOR-ed into column 1 in the **TIX** step (and the difference in column 1 after the **TIX** step is zero), we must have $\Delta\mathbf{S}[-2]_1 = \Delta\mathbf{S}[-2]_{24} = Y1_0 \neq 0$, so we still know that $\Delta\mathbf{S}[-2]_{0..3} = (x2\ Y1_0\ 0\ 0)$ must be non-zero. Also, the non-zero difference $Z1_3$ is copied to column 8 due to the $\mathbf{S}_8 + = \mathbf{S}_0$ operation. (We will use this in Round -3 below.)

10.1.4 Round -2

Since $\Delta\mathbf{S}[-2]_{0..3} = (x2\ Y1_0\ 0\ 0)$ then the difference before the Super-Mix (and after the S-box) is a codeword in $\mathcal{C}_{0,1}$. However, the minimum weight of that code is only 6, so all we can say is that at least two of the four columns $y2_{0..3}$ must be non-zero.[6] In particular each of these column differences separately could be all-zero, hence we write them all in lowercase.

Going through the next **ROR3** step brings $X1$ to the first column, so we have $\Delta\mathbf{S}[-2+s1]_{0..3} = (y2_3\ X1\ 0\ 0)$ (which is non-zero). Again all we know is that we have a code word in $\mathcal{C}_{0,1}$, which is not enough to preclude any of the $z2_i$'s from being zero.

---

[6] However, if $x_2 = 0$ then we have a codeword in $\mathcal{C}_1$ with minimum weight 12, so at least three of the $y2_i$'s must be non-zero. In other words, at most two of the five column differences $x2, y2_{0..3}$ can be zero. The same argument implies that at most two of the fixed column differences $y2_3, z2_{0..3}$ can be zero.

Note that in this round the **CMIX** operations already add some non-zero differences. In particular, in the second **CMIX** we have $\Delta\mathbf{S}_0+ = \Delta\mathbf{S}_4(= X1)$ and in the first **CMIX** we have $\Delta\mathbf{S}_1+ = \Delta\mathbf{S}_5(= Z1_3)$. We denote the modified columns by primed variables, namely $y2'_0 = y2_0 + X1$ and $z2'_1 = z2_1 + Z1_3$.

Going back through the **TIX** step, we again have $\Delta\mathbf{S}[-3]_0 = \Delta\mathbf{S}[-3]_{10} = x3$, but again we have no guarantee that $x3$ is non-zero. (However, notice that the non-zero difference $Z1_3$ was moved in the two **ROR3** steps from column 8 to Column 2, where the **TIX** operation leaves it unchanged.)

10.1.5 Round −3

Now we have $\Delta\mathbf{S}[-3]_{0..3} = (x3\ y2'_0\ Z1_3\ 0)$, so we still know that the **SMIX**output differences (and therefore also input differences) must be non-zero, but each individual column could still be zero. For the first **SMIX** in this round we only have $\Delta\mathbf{S}[-4+s1]_{0..3} = (y3_3\ x2\ 0\ 0)$, so we don't even have a guarantee that the input/output differences are non-zero.

Here again, the **CMIX** steps add some potentially non-zero differences, so we need to introduce new primed variables, namely: $y3'_0 = y3_0 + x2$, $z3'_1 = z3_1 + z2_3$, $z3'_2 = z3_2 + X1$, $y1'_1 = Y1_1 + z2_3$, and $y1'_2 = Y1_2 + X1$.

The **ROR3** steps in this round bring back the non-zero $X1$ to Column 3, where the **TIX** step leaves it unchanged.

10.1.6 Round −4

Now we have $\Delta\mathbf{S}[-4]_{0..3} = (x4\ y3'_0\ z2_3\ X1)$, so we still know that the **SMIX**output differences (and therefore also input differences) must be non-zero, but each individual column could still be zero. For the first **SMIX** in this round we only have $\Delta\mathbf{S}[-4+s1]_{0..3} = (y4_3\ x3\ 0\ 0)$, so we don't even have a guarantee that the input/output differences are non-zero.

Again, we need to introduce new primed variables due to the **CMIX** steps, namely: $y4'_0 = y4_0 + x3$, $z1'_0 = Z1_0 + x3$, $z4'_{0,1,2} = z4'_{0,1,2} + (Z1_3\ z3_3\ x2)$, and $(y2''_0\ y2'_1\ y2'_2) = (y2'_0\ y2_1\ y2_2) + (Z1_3\ z3_3\ x2)$.

The **ROR3** steps in this round bring the non-zero $Z1_3$ to Column 1, but the **TIX** step adds to it the (possibly non-zero) difference $y4'_0$ from Column 24. The difference $\Delta\mathbf{S}[-5]$ is shown at the bottom of Table 9.

10.1.7 Summing it up

Evolving the state backward from a collision point, we obtained the state differences just before the **TIX** steps in rounds zero through −4. We now shift our point of view and consider the constraints that are imposed on $\Delta S[-4]$, and the constraints that $\Delta\mathbf{S}[-4]$ imposes on the later differences in rounds −3 through zero. We ignore round −4 for the moment.

First, observe that almost all the variables in Tables 8 and 9 are uniquely determined by variables from earlier (lower) steps. The only variables that are *not uniquely determined* by earlier values are listed below (and marked by a box in the tables):

vars	Determined by
$z4_3$	input difference $z4_3 = \Delta P[-4]$
$y4_3, x3$	output of first **SMIX** in round -4
$z2_3, X1$	output of second **SMIX** in round -4
$y3_3$	output of first **SMIX** in round -3
$y2_3$	output of first **SMIX** in round -2
$Y1_3$	output of first **SMIX** in round -1

This means that once the attack reaches the end of round -4, there is very limited amount of wiggle room left for the attacker to determine things further. For example, all the input differences from then on will be fixed, and moreover the state differences must all evolve to match the pattern from Tables 8 and 9. In the following section we will make heavy use of this limited choice of the attacker. Many of the constraints that we need are summarized in Lemma 10.1 below.

More notations. Before presenting the lemma, we need one more piece of notation. Recall that in Tables 8, 9 we use y and z variables to denote the differences in input to the **SMIX** steps. Below we will also use $\hat{y}$'s and $\hat{z}$'s to denote the differences after the byte-substitution and before the linear Super-Mix. Namely, if yi_j is the difference in the j'th input column to the second **SMIX** step in round i, then $\widehat{yi}_j$ will be the difference in the same column after the byte substitution of that **SMIX** step.

Lemma 10.1 (Difference Evolution) *With the notations in this section, the following must hold for every successful internal-collision attack:*

A. The state difference $\Delta\mathbf{S}[-4]$ *at the end of round* -4 *must satisfy the following conditions:*

$$\Delta S[-4]_{5,6,11} = 0$$
$$\Delta S[-4]_0 = \Delta S[-4]_{10}$$
$$\Delta S[-4]_1 = \Delta S[-4]_{24}$$
$$\Delta S[-4]_3, \Delta S[-4]_7, \Delta S[-4]_{12}, \Delta S[-4]_{15}, \Delta S[-4]_{16}, \Delta S[-4]_{17} \text{ are all non-zero.}$$

B. The input differences ΔP *in the last four rounds must satisfy the following conditions:*

$$\begin{aligned} \Delta P[-3] &= \Delta S[-4]_8 \\ \Delta P[-2] &= \Delta S[-4]_2 \\ \Delta P[-1] &= \Delta S[-4]_7 \\ \Delta P[\,0\,] &= 0 \end{aligned}$$

C. In addition, the following must hold for the intermediate differences $\hat{z}, \hat{y}$ *in the rounds* -3 *to* -1*.*

1. $\hat{z3}_{0..3} = N^{-1}(y3_3,\ \Delta S[-4]_9,\ 0, 0)^T$ *where* $y3_3$ *can be arbitrary.*
2. $\hat{y3}_{0..3} = N^{-1}(x3,\ y2'_0,\ \Delta P[-1]_3,\ 0)^T$*, where* $x3 = \Delta S[-4]_4$ *and* $y2'_0 = \Delta S[-4]_{18}$.
3. $\hat{z2}_{0..3} = N^{-1}(y2_3, X1, 0, 0)^T$ *where* $y2_3$ *can be arbitrary and* $X1 = \Delta S[-4]_3$
4. $\hat{y2}_{0..3} = N^{-1}(x2, Y1_0, 0, 0)^T$ *where* $x2 = \Delta S[-4]_9$ *and* $Y1_0 = \Delta S[-4]_{12}$.
5. $\hat{z1}_{0..3} = N^{-1}(Y1_3, 0, 0, 0)^T$ *where* $Y1_3$ *can be arbitrary but non-zero.*
6. $\hat{y1}_{0..3} = N^{-1}(X1, 0, 0, 0)^T$ *where* $X1 = \Delta S[-4]_3$.

(Recall that N *is the* 16×16 *matrix defining the linear Super-Mix operation.)*

10.2 Differential Attacks

Since any internal collision with both messages P and P' at least four words long, must satisfy the conditions in Lemma 10.1, the pair of messages and their states must satisfy the following

1. The states at the end of round -4 must have a difference $\mathbf{\Delta S}[-4]$ satisfying the conditions in Lemma 10.1.A
2. The last four words $P[-3..0]$ and $P'[-3..0]$ must be subject to the constraints in Lemma 10.1.B, and together with the states must satisfy Lemma 10.1.C.

Note that so far we have not restricted the attack in any manner, except for the length restriction. However, in the next few sections we focus on differential attacks, with increasing levels of generality. Later, in section 10.2.4, we will address the issue of short length messages.

10.2.1 Pure differential Attacks

We begin with an analysis of a "pure differential attack", where the attacker is assumed to control the difference $\mathbf{\Delta S}$, but the actual states and the messages are assumed to be random. Later we remove the unrealistic assumption about the plaintext being random. We model a "pure differential attack" by the following probabilistic game:

<u>Pure Differential Internal-Collision Attack:</u>

1. The attacker freely chooses a state difference $\mathbf{\Delta S}[-4]$.
2. The state $\mathbf{S}(P)[-4]$ and inputs $P[-3]$ through $P[0]$, as well as inputs $P'[-3]$ through $P'[0]$, are chosen at random. Then, the state $\mathbf{S}(P')[-4]$ is set to $\mathbf{S}(P)[-4] + \mathbf{\Delta S}[-4]$.
3. The attack is successful if the resulting states have an internal collision in round 0, namely $\mathbf{S}(P)[-1] \neq \mathbf{S}(P')[-1]$ but $\mathbf{S}(P)[0] = \mathbf{S}(P')[0]$.

Clearly, to have a non-zero probability of success the attacker in this game must choose differences $\mathbf{\Delta S}[-4]$ and ΔP that satisfy the constraints of Lemma 10.1.A-B. Since by Lemma 10.1.B,

Table 10: The inputs to the last six **SMIX** steps

SMIX Step	Input words ($\mathbf{S}$ denotes $\mathbf{S}[-4]$)	Output
$SMIX[-2.5]$	$\mathbf{S}_{27} + \mathbf{S}24 + \mathbf{S}_1,\ \mathbf{S}_{28} + P[-2],\ \mathbf{S}_{29} + M[-3]_3,\ P[-3]$	$M[-2.5]_{0..3}$
$SMIX[-2]$	$\mathbf{S}_{24} + M[-2.5]_1,\ \mathbf{S}_{25} + M[-2.5]_2,\ \mathbf{S}_{26} + M[-2.5]_3,\ M[-2.5]_0$	$M[-2]_{0..3}$
$SMIX[-1.5]$	$\mathbf{S}_{21} + \mathbf{S}_{18} + M[-2]_1,\ \mathbf{S}_{22} + M[-2]_2,\ \mathbf{S}_{23} + M[-2]_3,\ P[-2]$	$M[-1.5]_{0..3}$
$SMIX[-1]$	$\mathbf{S}_{18} + M[-1.5]_1,\ \ \mathbf{S}_{19} + M[-1.5]_2,\ \mathbf{S}_{20} + M[-1.5]_3,\ M[-1.5]_0$	$M[-1]_{0..3}$
$SMIX[-0.5]$	$\mathbf{S}_1 + \mathbf{S}24 + \mathbf{S}_{12} + \mathbf{S}_{15} + M[-1]_1,\ \mathbf{S}_{16} + M[-1]_2,\ \mathbf{S}_{17} + M[-1]_3,\ P[-1]$	$M[-0.5]_{0..3}$
$SMIX[0]$	$\mathbf{S}_1 + \mathbf{S}_{12} + \mathbf{S}_{24} + M[-0.5]_1,\ \mathbf{S}_2 + \mathbf{S}_{13} + M[-0.5]_2,\ \mathbf{S}_3 + \mathbf{S}_{14} + M[-0.5]_3,$ $M[-0.5]_0$	$M[0]_{0..3}$

$\Delta\mathbf{S}[-4]$ completely determines $\Delta P[-3]$ through $\Delta P[0]$, then once $P[-3]$ to $P[0]$ are chosen, $P'[-3]$ to $P'[0]$ are forced. Further, we argue, that for any choice of $\Delta\mathbf{S}[-4]$ that satisfies Lemma 10.1.A, the probability of satisfying Lemma 10.1.C is bounded from above by 2^{-246}.

The independence assumption. Table 10 lists the inputs to the **SMIX** steps in rounds -3 through -1, expressed in terms of the state $\mathbf{S}$ at the end of round -4 and the input $P[-3\ \ldots -1]$. The SMIX steps are numbered from -2.5 (which is the first **SMIX** in round -3) to 0 (which is the last **SMIX** step before round 0).

It can be seen from that table that when $\mathbf{S}[-4]$ and $P[-3\ \ldots 0]$ are chosen uniformly at random, then almost all the inputs to all the **SMIX** steps are also uniform and *independent*. In fact, the only ones that are not uniform are Column 3 in the input to steps SMIX$[-2]$, SMIX$[-1]$, and SMIX$[0]$. (These columns are just the first output column from the previous SMIX step.)

Below we assume, however, that the probability of satisfying the conditions in Lemma 10.1.C is no higher than if these inputs to the third column too are chosen at random (with the appropriate differentials). This heuristic assumption is very common in cryptanalysis, and below we refer to it as the *independence assumption*. We can now state our result for pure differential attacks:

Theorem 10.2 *Under the independence assumption, for every fixed non-zero D, it holds that*

$$\Pr[\textit{Internal collision at round 0} \mid \Delta\mathbf{S}[-4] = D] \leq 2^{-246}$$

where the probability is taken over a uniform choice of $\mathbf{S}(P)[-4]$, $\mathbf{S}(P')[-4]$, $P[-3\ \ldots 0]$, $P'[-3\ \ldots 0]$.

Proof: Since, we are bounding the probability conditioned on the state difference $\Delta\mathbf{S}[-4]$ being a fixed non-zero value D, it follows that if D (substituted for $\Delta\mathbf{S}[-4]$) does not satisfy the conditions in Lemma 10.1.A, then the probability above is zero. So, from now on we assume that D, and hence $\Delta\mathbf{S}[-4]$, do satisfy those conditions. Further by part B of that lemma, $\Delta P[-3\ \ldots 0]$ is fixed, once $\Delta\mathbf{S}[-4]$ is fixed to D. Thus in the above probability, it suffices to consider the case of choosing $\mathbf{S}(P)[-3\ \ldots 0]$ at random, and letting $P'[-3\ \ldots 0] = P[-3\ \ldots 0] + \Delta P[-3\ \ldots 0]$.

We analyze the **SMIX** steps one by one, obtaining a bound for each of them and then (using the independence assumption) multiplying all the bounds.

SMIX[−3.5]. The input differences to this **SMIX** steps are completely determined by $\Delta\mathbf{S}[-4]$ and ΔP. there are two cases to consider here: either these input differences (which are denoted $z4_{0..3}$) are all zero, or they are not. In the first case the output differences must also be zero (namely $y4_3 = x3 = 0$).

SMIX[−3]. For this step we use the trivial probability bound of 1, and just note that the condition C2 of the lemma implies that $X1 = M[3]_3 \neq 0$ (which we will use later).

SMIX[−2.5]. The input differences to this **SMIX** step are completely determined by $\Delta\mathbf{S}[-4]$ and ΔP. We have two cases: either these fixed input differences (denoted $z3_{0..3}$) are all zero, or they are not. In the first case the output differences must also be zero (namely $y3_3 = x3 = 0$). We now analyze the other case.

Recall that the differences after the byte substitution in this round are denoted $\hat{z3}_{0..3}$. To meet the condition C1 in the lemma we must have $N \cdot \hat{z3} = (y3_3, x2, 0, 0)^T$ for some values $y3_3, x2$, which means that $\hat{z3} \in \mathcal{C}_{0,1}$, and since we assume that the differences are not all-zero then $\hat{z3}$ is a non-zero codeword. Namely, with the input difference $z3$ being fixed by our choice of $\Delta\mathbf{S}[-4]$, the difference after the byte substitution must be a non-zero codeword in $\mathcal{C}_{0,1}$.

Now recall Example 2 form Section 7.3: with the input difference $z3$ being fixed, we can ask how many zero bytes are in the vector $z3$, and this must also be the number of zero-bytes in the codeword $\hat{z3}$. If we denote by $m = m_{2.5}$ the number of *non-zero bytes* in $z3$ (and therefore also $\hat{z3}$), then the byte-substitution in this step must "hit" a codeword of $\mathcal{C}_{0,1}$ with exactly that number of non-zero bytes, and by Lemma 7.3 there are at most $256^{m-\mathrm{minRank}_m(\mathcal{C}_{0,1})}$ such codewords. On the other hand, the probability of hitting any particular 16-byte difference with m non-zero bytes in the byte-substitution is at most 2^{-6m} (using the differential properties of the S-box). Consulting Table 3 that has the min-rank numbers for the various codes, we conclude that the probability of "hitting" any non-zero codeword in this step is at most

$$\max_m \left(2^{-6m} \cdot 2^{8(m-\mathrm{minRank}_m(\mathcal{C}_{0,1}))} \right) \;=\; 2^{-28}$$

(where the maximum is obtained at $m = 6$).

SMIX[−2]. The input differences here are determined by $\Delta\mathbf{S}[-4]$ and the output difference in the previous **SMIX** step. Condition C2 says that the output difference from this step is of the form $(x3\ y'_{20}\ \Delta P[-1]\ 0)$, which implies in particular that $M[-2]_2 = \Delta P[-1] \neq 0$. This means that the output difference (and hence also input difference) to this step is non-zero. On the other hand $M[-2]_3 = 0$, so the differences after byte substitution $\hat{y3}_{0..3}$ are a non-zero codeword in $\mathcal{C}_{0,1,2}$.

Observe that the required *output difference* of this step is also uniquely determined by $\Delta\mathbf{S}[-4]$ (see part C of Lemma 10.1). Hence after the byte-substitution we must "hit" one specific non-zero codeword in $\mathcal{C}_{0,1,2}$. This codeword must have at least two non-zero bytes (since the minimum-weight of $\mathcal{C}_{0,1,2}$ is 2), so the probability of "hitting" it is at most $2^{-6\times 2} = 2^{-12}$.

Moreover, if we have $x3 = 0$, then $\widehat{y3}_{0..3}$ is a non-zero codeword in $\mathcal{C}_{1,2}$. The min-weight of this code is six, so in this case we have a bound of $2^{-6\times 6} = 2^{-36}$.

We conclude that the probability of satisfying the condition C2 from the lemma is at most 2^{-12} when $x3 \neq 0$ and at most 2^{-36} when $x3 = 0$.

SMIX$[-1.5]$. This is similar to SMIX$[-\mathbf{2.5}]$. The input differences to this **SMIX** steps are completely determined by $\Delta\mathbf{S}[-4]$, ΔP, and by the output differences in the previous **SMIX** steps, but here we know that the output difference is non-zero. Hence the differences $\widehat{z1}$ after byte substitution must be a non-zero codeword in $\mathcal{C}_{0,1}$, and we get same analysis showing a bound of 2^{-28} on the probability of satisfying the condition C3 in the lemma.

SMIX$[-1]$. In this step the input difference is determined by $\Delta\mathbf{S}[-4]$, ΔP, and by the output differences in the previous **SMIX** steps. the output difference is completely determined by $\Delta\mathbf{S}[-4]$. Moreover, condition C4 says that the output difference is of the form $(x2\ Y1_0\ 0\ 0)$, so the differences after byte substitution $\widehat{y2}_{0..3}$ must be a specific non-zero codeword in $\mathcal{C}_{0,1}$. The min-weight of $\mathcal{C}_{0,1}$ is six, so the probability of hitting any specific codeword in it is at most $2^{-6\times 6} = 2^{-36}$.

Moreover, if we had an all-zero difference in step $[-\mathbf{2.5}]$ above, then we have $x2 = 0$ which means that $\widehat{y2}_{0..3}$ is a non-zero codeword in $\mathcal{C}_1$. The min-weight of that last code is twelve, so in this case we have a bound of $2^{-6\times 12} = 2^{-72}$.

We conclude that the probability of satisfying the condition C4 from the lemma is at most 2^{-36} when $x2 \neq 0$ and at most 2^{-72} when $x2 = 0$.

SMIX$[-0.5]$. The input differences here are determined by $\Delta\mathbf{S}[-4]$ and the output difference in the previous **SMIX** step, and Condition C5 says that the output difference from this step is of the form $(Y1_3\ 0\ 0\ 0)$ which is non-zero. Hence the difference after byte substitution $\widehat{Z1}_{0..3}$ is a non-zero codeword in $\mathcal{C}_0$ (and thus has at least 13 non-zero bytes).

Repeating the argument from Example 2 in Section 7.3, the fixed input difference $Z1_{0..3}$ to the **SMIX** step can have between 13 and 16 non-zero bytes. If we denote by $m = m_{0.5}$ the number of *non-zero bytes* in $Z1$ (and therefore also $\widehat{Z4}$), then the byte-substitution in this step must "hit" a codeword of $\mathcal{C}_0$ with exactly that number of non-zero byte. By Lemma 7.3 there are at most $256^{m-\mathrm{minRank}_m(\mathcal{C}_0)}$ such codewords, and the probability of hitting any specific one is at most 2^{-6m} (using the differential properties of the S-box). We again consult Table 3 and conclude that the probability of "hitting" a non-zero codeword in this step is at most

$$\max_m \left(2^{-6m} \cdot 2^{8(m-\mathrm{minRank}_m(\mathcal{C}_0))}\right) \;=\; 2^{-64}$$

(where the maximum is obtained at $m = 0$).

SMIX[0]. Here the input difference is determined by $\Delta\mathbf{S}[-4]$, ΔP, and by the output differences in the previous **SMIX** steps, while the output difference is completely determined by $\Delta\mathbf{S}[-4]$. Moreover, the output difference is of the form $(X1\ 0\ 0\ 0)$ some some fixed non-zero $X1$. Hence the difference after byte substitution $\hat{Y1}_{0..3}$ must be some fixed non-zero codeword in $\mathcal{C}_0$, and thus it has at least 13 non-zero bytes. The probability of hitting that codeword is therefore at most $2^{-6\times 13} = 2^{-78}$.

Concluding the proof. All that is left now is to multiply all the probability bounds. We have several cases to consider depending on whether or not the input difference to the step SMIX[−2.5] is all-zero.

For steps SMIX[−2.5] and SMIX[−1]: If the input differences to SMIX[−2.5] is all-zero then we have a bound of 1 for that step but a bound of 2^{-72} for SMIX[−1], and otherwise we have 2^{-28} for SMIX[−2.5] and 2^{-36} for SMIX[−1]. Either way, the product of the two is at most 2^{-64}. For step SMIX[−2] we have a bound of 2^{-12}. Factoring in also the bounds 2^{-28} for SMIX[−1.5], 2^{-64} for SMIX[−0.5], and 2^{-78} for SMIX[0] we obtain the total bound

$$\Pr[\text{Internal collision at round 0}] \le 2^{-64} \times 2^{-12} \times 2^{-28} \times 2^{-64} \times 2^{-78} = 2^{-246}$$

■

Some improvements and extensions. The analysis of Theorem 10.2 can be improved in several ways. First, the bound can be improved by a closer look at some of the constraints. In particular, a closer look at $\hat{Y1}$ and $X1$ reveals that any zero bytes in $\hat{Y1}$ imply some linear constraints on $X1$, which in turns implies some more linear constraints on the code word $\hat{z2}$, thus increasing its weight. This analysis is found in Section 10.3, where it is shown that this additional argument improves the bound in Theorem 10.2 from 2^{-246} to 2^{-264}.

A different approach for improving the bound takes advantage of the fact that most input/output differences of the S-box have probability either 2^{-7} or zero, and only very few of them have probability 2^{-6}. So far we did not check the extent to which we can apply this fact.

For the purists, a different extension allows us to get rid of the Independence assumption: Observe that since the only input columns that are not uniformly random are the third columns in the **SMIX**-es −2 and −1, we can eliminate the assumption at the price of discounting four non-zero bytes in each of these steps. It degrades the bounds for the two steps by 2^{-24} each. Hence, without making the Independence assumption we can still prove a bound of 2^{-198} (or 2^{-216} when using the improved argument from Section 10.3).

10.2.2 More Realistic Differential Attacks

The "pure differential attack" from above is quite weak, since the attacker is not allowed to control the input messages P, P'. However, this aspect is easy to fix, as we show now.

We relax the "pure differential" condition by allowing the attacker full control over the messages P, P', but still insisting that the "initial states" $\mathbf{S}, \mathbf{S}'$ are chosen at random (subject to some difference $\Delta\mathbf{S}[-4]$ that the attacker chooses). Namely, we consider the following probabilistic game:

Semi-Pure Differential Internal-Collision Attack:

1. The attacker freely chooses a state difference $\Delta\mathbf{S}[-4]$;
2. The state $\mathbf{S}(P)[-4]$ is chosen at random, and then the state $\mathbf{S}(P')[-4]$ is set to $\mathbf{S}(P)[-4] \oplus \Delta\mathbf{S}[-4]$.
3. The attacker is given $\mathbf{S}(P)[-4]$ and $\mathbf{S}(P')[-4]$. The attack is successful if there exist any two messages $P[-3\ \ldots 0]$, $P'[-3\ \ldots 0]$ that starting from these two states induce an internal collision in round 0 (namely $\mathbf{S}(P)[-1] \neq \mathbf{S}(P')[-1]$ but $\mathbf{S}(P)[0] = \mathbf{S}(P')[0]$).

We note that "semi-pure" differential attacks are actually quite realistic. Given the constraints from Lemma 10.1, a natural line of attack to consider is to choose at random very long messages P, P', and find pairs of intermediate internal states in the processing of these messages whose difference satisfies the conditions in Lemma 10.1. Then, given such a pair of states, the attacker can try to find four last words that can be added after each of these states to cause an internal collision. Since the internal states in this attacks were reached via a "random walk" from the initial state, it is reasonable to model them as random, with the freedom to choose their difference.

Theorem 10.3 *Under the Independence assumption, no "semi-pure" attacker as above can find internal collisions with probability better than* 2^{-150}*, where the probability is taken over a uniform choice of* $\mathbf{S}(P)[-4]$ *(and we set* $\mathbf{S}(P')[-4] = \mathbf{S}(P)[-4] + \Delta\mathbf{S}[-4]$*).*

Proof: The theorem follows from Theorem 10.2, and Lemma 10.1.B by a simple union bound. To elaborate, let the state difference that the adversary chooses be some non-zero D. Let π represent any 96-bit binary string, a potential choice for $P[-3\ \ldots -1]$. We will ignore $P[0]$, as its choice does not affect the internal collision. Now, in order to get an internal collision, Lemma 10.1.B forces $\Delta P[-3\ \ldots 0]$, once $\Delta\mathbf{S}[-4]$ is fixed to D. Thus, if $P[-3\ \ldots -1]$ is chosen to be π, then $P'[-3\ \ldots -1]$ is just π plus this forced $\Delta P[-3\ \ldots -1]$ (call it π'). Let $\Delta\mathbf{S}(\pi)[0]$ denote the difference in the states after round 0, with the first state starting from $\mathbf{S}(P)[-4]$ and evolving using input π, and the second state starting from $\mathbf{S}(P')[-4]$ and evolving using π'.

Now, starting with the quantity we bound in Theorem 10.2, we get

$$\begin{aligned}
&\Pr_{P[-3\ldots0],S(P)[-4]}[\Delta\mathbf{S}[0]=0 \mid \Delta S[-4]=D] \\
=\;& \sum_{\pi} \Pr_{P[-3\ldots0],S(P)[-4]}[P[-3\ \ldots 0]=\pi \ \wedge\ \Delta\mathbf{S}(\pi)[0]=0 \mid \Delta S[-4]=D] \\
=\;& \sum_{\pi} \Pr[P[-3\ \ldots 0]=\pi] \times \Pr_{S(P)[-4]}[\Delta\mathbf{S}(\pi)[0]=0 \mid \Delta S[-4]=D] \\
=\;& 2^{-96} \times \sum_{\pi} \Pr_{S(P)[-4]}[\Delta\mathbf{S}(\pi)[0]=0 \mid \Delta S[-4]=D] \\
\geq\;& 2^{-96} \times \Pr_{S(P)[-4]}[\exists\pi\ :\Delta\mathbf{S}(\pi)[0]=0 \mid \Delta S[-4]=D]
\end{aligned}$$

Thus,

$$\Pr_{S(P)[-4]}[\exists\pi\ :\Delta\mathbf{S}(\pi)[0]=0 \mid \Delta S[-4]=D]$$

is at most $2^{96} \times 2^{-246} = 2^{-150}$. ■

Clearly, all the extensions that are mentioned for the "pure" case hold here too, so we can improve the bound by at least a factor of 2^{-18} by using a more refined analysis, and/or eliminate the Independence assumption by paying a factor of 2^{48} in the bound.

10.2.3 Beyond Random Initial State

The analysis from above gave the adversary complete control over the initial state difference $\Delta S[-4]$, but required that the states themselves be random (subject to this difference). In reality, however, the state is not random, rather the attacker can choose the two inputs leading from the initial state to $\mathbf{S}[-4]$ such that not only would it satisfy the conditions in Part A of Lemma 10.1 but also have better than random probability to satisfy the other parts as well.

Specifically, one should be wary of message-modification techniques, where parts of the input message are held fixed to control some state bytes while others are chosen as needed for the differential attack. Such techniques are backed by control/neutral bytes analysis, taking advantage of the fact that an input bytes can be used to effect (control) some specific state byte without changing other bytes (neutral).

Below we argue informally that such techniques are not likely to apply to Fugue, due to its fast diffusion properties. First observe that in our context, we only need to argue about "message modification" being applied to message words $P[-4]$ or earlier, since the words $P[-3\ \ldots 0]$ are already included in the union-bound argument of Theorem 10.3.

Considering the diffusion Table 4 (on Page 42), we see that an input byte in $P[-4]$ or earlier influences every byte in every **SMIX** step in rounds -3 and on (except of course the new input bytes in these round), so we have no neutral bytes to consider here. Moreover, a byte in $P[-4]$ influences the bytes in the last four **SMIX** steps in a highly non-linear fashion, as its path of influence goes through four **SMIX**-es for the input to SMIX[−1.5] and via seven **SMIX**-es for the input to SMIX[0].

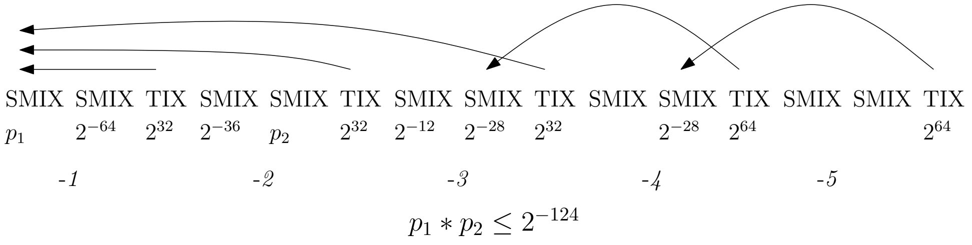

Figure 6: Free Message Modification upto 3 SMIXes

It seems reasonable to assume that an attacker cannot use "message modifications" through that many **SMIX**-es. Consider the improved bound 2^{-168} that one gets using the refined analysis in Section 10.3, and we show below how this bound degrades when we allow the attacker "free message modification" through three, four, or five **SMIX**-es.

- Allowing "free message modification" through three **SMIX**-es means that the attacker can use its input $P[-4]$ to get better than random probability upto SMIX$[-2.5]$ but not beyond that. This will leave us with the bounds on the last five SMIX-es which gives probability bound of 2^{-236} for random messages (with adversarial difference), and 2^{-140} after the union bound. This is depicted in Figure 6, where the bound on $p_1 * p_2$ is obtained from the tighter analysis in Section 10.3.
- Allowing "free message modification" through four **SMIX**-es degrades the bounds to 2^{-224} for random messages with adversarial difference, and 2^{-128} after the union bound.
- Allowing "free message modification" through five **SMIX**-es degrades the bounds to 2^{-196} for random messages with adversarial difference, and 2^{-100} after the union bound.

To appreciate the last bullet, note that even if we allow the attacker full control over the state difference, and moreover let it "magically" use input words from five **SMIX**-es ago (which are at least as non-linear as half-AES) to avoid having to pay for fourteen active S-boxes, we still have a *proof* that the probability that *any extension exists* that would lead to internal collision is below 2^{-100}. Note that even a modest cost of 2^{28} to actually find these extensions would already push it beyond the complexity needed for a simple birthday attack.

10.2.4 The Length-Padding in Fugue-256

We conclude this section by noting the effect of the length padding in Fugue-256 on differential internal-collision attacks. For one thing, it implies that messages whose bit-lengths are not the same modulo 2^{32} cannot have an internal collision (since the last input word will be different in this case). This means that one can only have internal collision either for messages of the same length, or for messages whose lengths differ by a multiple of 2^{29} bytes. If one of the messages (w.l.o.g. P) is less than four words long (say m words long) in the analysis of the previous sections,

Table 11: Min-Rank$_m$ values for Codes $\mathcal{C}_{0,1}^I$

m	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\lvert I\rvert = 3$	-	-	-	-	-	-	-	-	9	9	10	11	11	11	11	11
$\lvert I\rvert = 2$	-	-	-	-	-	-	7	7	8	9	9	10	10	10	10	10
$\lvert I\rvert = 1$	-	-	-	-	-	-	7	7	8	8	9	9	9	9	9	9

then its state $\mathbf{S}[-m+1]$ is the initial state. If the other message is of equal length, then there can be no collision, by Lemma 10.1. Otherwise, the other message is extremely long, and must achieve a state $\mathbf{S}(P')[-m+1]$ as required by Lemma 10.1, since the other $\mathbf{S}(P)[-m+1]$ is fixed. Note that the analysis of the previous section was focused on achieving zero differences in the state. Now, the adversary must find an exact match in the state for those columns. We assume that this is a harder problem than finding the difference to be zero.

10.3 A Tighter Analysis for Theorem 10.2

Recall that the output difference in SMIX[0] is (X_1 0 0 0) and the input difference is $Y1_{0..3}$. We used that to conclude that $\hat{Y1} \in \mathcal{C}_0$ and therefore it can have at most three zero bytes. We now observe that if $\hat{Y1}$ has *any zero bytes* then this implies additional linear constraints on $X1$. Namely, every zero byte in $\hat{Y1}$ implies a constraint on $X1$ of the form $(N^{-1})^i_{0..3} \cdot X1 = 0$ (where $(N^{-1})^i_{0..3}$ is a 1×4 sub-matrix of N^{-1}).

Now, we focus our attention on step SMIX[−1.5], where the output difference is ($y2_3$ $X1$ 0 0), and the differences after the byte substitution are denoted $\hat{z2}_{0..3}$. The additional constraints on $X1$ imply more parity-check equations on $\hat{z2}$ beyond the ones that define $\mathcal{C}_{0,1}$. Namely, a subset I of zero bytes in $Y1$ implies that $\hat{z2}$ belongs to a sub-code $\mathcal{C}^I_{0,1} \subseteq \mathcal{C}_{0,1}$, and that sub-code may have higher minimum weight.

We used a computer program to compute the min-rank of the codes $\mathcal{C}^I_{0,1}$, for all possible subsets I (of three or less indices). The values are listed in Table 11, with each row representing the minimum over all I of a particular size. We can now repeat the analysis of the step SMIX[−1.5] using each of these codes rather than the code $\mathcal{C}_{0,1}$, while at the same time using $16 - |I|$ (rather than 13) as the number of non-zero bytes of $Y1$ in SMIX[0]. This gives us the following bounds:

$\lvert I\rvert$	SMIX[0] bound	SMIX[−1.5] bound	Total bound
0 :	$2^{-6\times16}$	2^{-28}	2^{-124}
1 :	$2^{-6\times15}$	2^{-40}	2^{-130}
2 :	$2^{-6\times14}$	2^{-40}	2^{-124}
3 :	$2^{-6\times13}$	2^{-52}	2^{-130}

We conclude that the steps SMIX[0] and SMIX[−1.5] together contribute at least 2^{-124} to the bound (as compared to 2^{-106} in the analysis of Theorem 10.2).

11 Differential Analysis of Fugue-256: External Collisions

The final round **G** of **F**-256 consists of two phases. The first phase consists of five TIX-less rounds, which are the same as the input round transformation **F** with the **TIX** step removed. The purpose of this first phase is to provide quick non-linear mixing before the second phase, which has been designed with the sole purpose of providing a provable upper bound on the probability of an external collision (given that there was no internal collision).

Below we provide a differential analysis of the second phase of the the final round, proving that the probability of an external collision is bounded below 2^{-129}. As in Section 10, here too we consider an attacker that can freely chose the difference between two states at the beginning of the second phase, but we assume that the states themselves are chosen at random (subject to this difference). The reason for the first (TIX-less) phase is to make this random-state assumption more plausible: Recall from the Diffusion section (Section 8) that after five TIX-less rounds (consisting of ten **SMIX**-es), every byte of the state depends on every byte of the input in a highly non-linear fashion (via at least five **SMIX**-es). Moreover, columns 1-4 and 15-17 after the TIX-less phase depend non-linearly on every byte in the state before that phase.

For the analysis below it will be easier to view the second phase as implemented in a "sliding window" fashion, as described in Figure 7. Namely, we can write the second phase as follows:

```
$p = 0$;
For $i = 0$ to 25
{
    $\mathbf{S}_{p+4} \mathrel{+}= \mathbf{S}_p$;
    If $i$ is even Then
        $\mathbf{S}_{p+15} \mathrel{+}= \mathbf{S}_p$;
        $p = p + 15 \bmod 30$;
    Else
        $\mathbf{S}_{p+16} \mathrel{+}= \mathbf{S}_p$;
        $p = p + 16 \bmod 30$;
    $\mathbf{S}_{p..p+3} = \text{Super-Mix}(\text{S-Box}[\mathbf{S}_{p..p+3}])$;
}
$\mathbf{S}_{p+4} \mathrel{+}= \mathbf{S}_p; \mathbf{S}_{p+15} \mathrel{+}= \mathbf{S}_p$;
Output $\mathbf{S}_{p+1}, \mathbf{S}_{p+2}, \mathbf{S}_{p+3}, \mathbf{S}_{p+4}, \mathbf{S}_{p+15}, \mathbf{S}_{p+16}, \mathbf{S}_{p+17}, \mathbf{S}_{p+18}$
```

where the additions in the subscripts of **S** are done modulo 30. One can check that at the end of the loop we have $p = 13$, so the output is $\mathbf{S}_{14}$, $\mathbf{S}_{15}$, $\mathbf{S}_{16}$, $\mathbf{S}_{17}$, and $\mathbf{S}_{28}$, $\mathbf{S}_{29}$, $\mathbf{S}_0$, $\mathbf{S}_1$, as also seen from Figure 7.

We call each iteration of the above loop a *round*, and index the rounds by $i = 0, 1, \ldots, 25$.

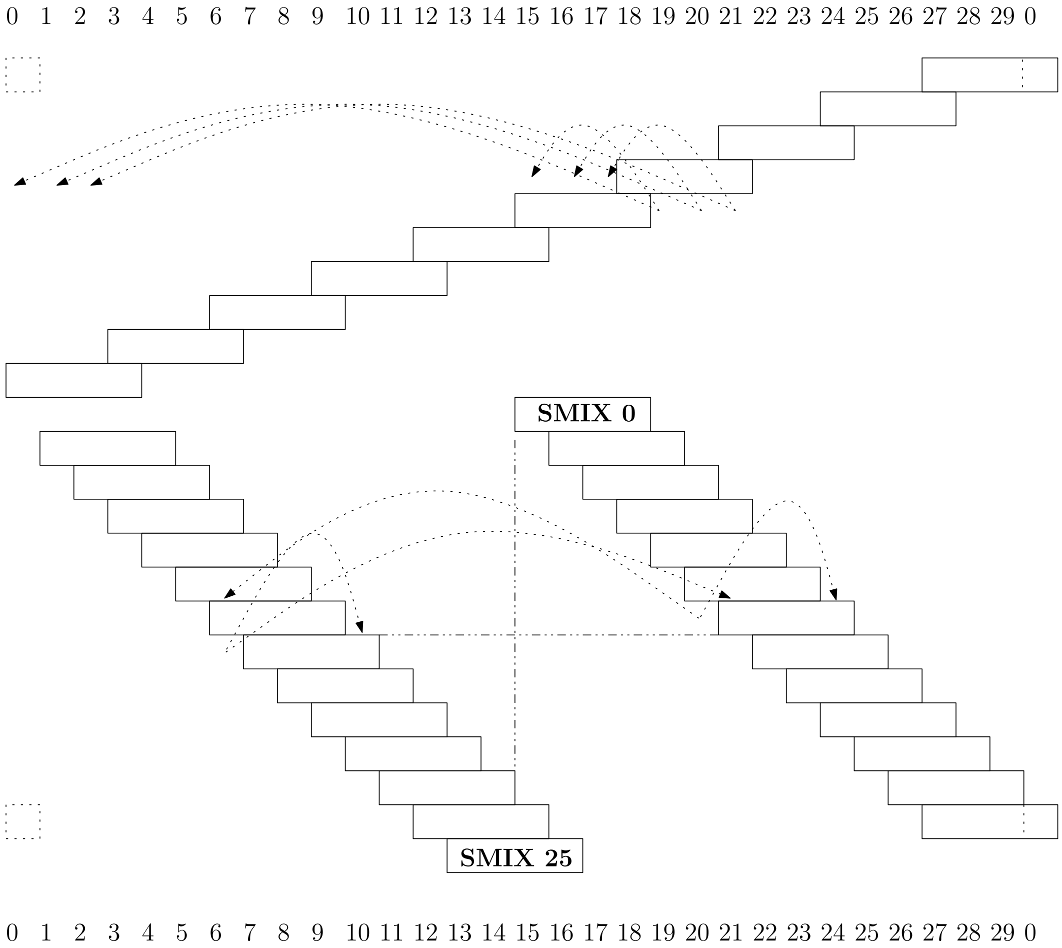

Figure 7: The Final Round **G** (in sliding window format)

The **SMIX** step in round i (that performs the **SMIX** transformation on the columns $\mathbf{S}_{p..p+3}$) is denoted SMIX$[i]$. The *output difference* in this **SMIX** step is denotes outSMIX$(i)_{0..3}$ (or just outSMIX(i)) and the input difference is inSMIX$(i)_{0..3}$ (or just inSMIX(i)).

The Independence Assumption for G2. Just as we did in the analysis of internal collisions, here too we will analyze each **SMIX** separately as if it is applied on an adversarially-chosen differential but a random states (subject to that differential). We will then multiply all the probabilities that we get, and assume that the combined differential behaves like a product of the individual ones. Formally this is much less justified here than it is for the internal collisions, but such an Independence assumption is nonetheless very common in cryptanalysis.

Theorem 11.1 *Under the Independence assumption, the probability of obtaining an external collision is at most* 2^{-129}*, where the probability is over choosing the state (for one input) at random at the end of the TIX-less rounds, and conditioned on any non-zero constant difference in the state at the end of the TIX-less rounds.*

Proof: Let the non-zero constant difference in the state at the end of the TIX-less rounds be D. Thus, e.g., the input difference to SMIX[0] is $D_{15..18}$ (as can be readily checked from Figure 7). Observe that since $D \neq 0$, then there must be some **SMIX** steps with non-zero input (and output) difference.

Let j^*-1 be the index of the last **SMIX** step that has a non-zero input (and output) difference. That is, outSMIX$(j^*-1) \neq 0$ but outSMIX$(k) = 0$ for all $k \geq j^*$. We observe that the last non-zero **SMIX** step cannot be SMIX[0] (so we much have $j^* \geq 2$). The reason is that if outSMIX$(0) \neq 0$ then either the first output column or at least one of the last three output columns must have a non-zero difference. If the last three column differences are not all zero then the input difference to SMIX[1] is non-zero. And if outSMIX$(0)_0 \neq 0$ (and all the steps SMIX[1] through SMIX[22] all have zero difference), then SMIX[23] that revisits the same column ($\mathbf{S}_{15}$) must have non-zero input difference.

We now have two cases: either $j^* \leq 24$ or $j^* > 24$, and we begin by considering the first case. Now, since inSMIX$(j^*)_{1,2}$ = outSMIX$(j^*-2)_{2,3}$, we have

$$\text{outSMIX}(j^*-2)_2 = \text{outSMIX}(j^*-2)_3 = 0 \tag{6}$$

Also, inSMIX$(j^*)_3$ = outSMIX$(j^*-2)_0+c$, where c is either determined by the starting differential D, or by the output of outSMIX(0) and/or outSMIX(2) (see Figure 7). Thus,

$$\text{outSMIX}(j^*-2)_0 = c \tag{7}$$

Further, inSMIX$(j^*)_0$ = outSMIX$(j^*-2)_1$ + outSMIX$(j^*-1)_0$. Thus,

$$\text{outSMIX}(j^*-2)_1 = \text{outSMIX}(j^*-1)_0 \tag{8}$$

Now, if outSMIX$(j^*-1)_0$ is zero, then all of outSMIX(j^*-1) is zero, as inSMIX$(j^*+1)_0$ = outSMIX$(j^*)_0$ + outSMIX$(j^*-1)_1$, and since the first summand is zero, and the sum itself is zero, then the second summand must be zero. Also, $\mathrm{inSMIX}(j^*+1)_{1,2} = \mathrm{outSMIX}(j^*-1)_{2,3}$, and hence $\mathrm{outSMIX}(j^*-1)_{2,3}$ are zero as well. But, $\mathrm{outSMIX}(j^*-1)$ is non-zero, by the choice of j^*, and hence $\mathrm{outSMIX}(j^*-1)_0$ must be non-zero. Thus, the input differences, $\mathrm{inSMIX}(j^*-1)$, after the S-Box substitutions form a codeword of $\mathcal{C}_0$. Further, $\mathrm{outSMIX}(j^*-1)_0$ equals a value c', which is either determined by D itself, or by the output of outSMIX(0) and/or outSMIX(2), because $\mathrm{inSMIX}(j^*+1)_3 = \mathrm{outSMIX}(j^*-1)_0 + c' = 0$.

Now, by equation (8) it follows that $\mathrm{outSMIX}(j^*-2)_1 = \mathrm{outSMIX}(j^*-1)_0 = c' \neq 0$. From, equations (6) and (7), it follows that the input difference $\mathrm{inSMIX}(j^*-2)$, after the S-Box substitutions, forms a codeword of $\mathcal{C}^I_{0,1}$ (as in Section 10.3).

Now, the values c and c' above were either determined by D itself or by outSMIX(0) and/or outSMIX(2). (The latter case arises only if $j^* > 21$.) In that case, w.l.o.g. we can take D to be the differential state after SMIX[2]. Thus, c and c' are completely determined by D. Hence, there is exactly one codeword choice for both $\mathrm{inSMIX}(j^*-1)$ and $\mathrm{inSMIX}(j^*-2)$ (after S-Box substitutions), and the two being codewords of $\mathcal{C}_0$ and $\mathcal{C}^I_{0,1}$ respectively, where I is the indexes in which $\mathrm{inSMIX}(j^*-1)$ is zero.

Thus, if there are k zero bytes in the input difference $\mathrm{inSMIX}(j^*-1)$, then by tables 11 and 3, the probability of obtaining the above conditions together is at most

$$\begin{aligned}
k=0 &\quad:\quad 2^{-96} \times 2^{-36} = 2^{-132}\\
k=1 &\quad:\quad 2^{-90} \times 2^{-48} = 2^{-138}\\
k=2 &\quad:\quad 2^{-84} \times 2^{-48} = 2^{-132}\\
k=3 &\quad:\quad 2^{-78} \times 2^{-60} = 2^{-138}
\end{aligned}$$

Hence the probability is upper bounded by 2^{-132}.

Now, we consider the case where $j^* > 24$. There are only two possibilities: either $j^* = 25$ (outSMIX(25) is zero) or $j^* = 26$ (outSMIX(25) is non-zero).

Before we proceed, note that the output of the hash function is the following

$$\begin{aligned}
&\mathrm{SMIX}(25)_1, \mathrm{SMIX}(25)_2, \mathrm{SMIX}(25)_3, \mathrm{SMIX}(25)_0 + \mathrm{SMIX}(4)_0,\\
&\mathrm{SMIX}(24)_1 + \mathrm{SMIX}(25)_0, \mathrm{SMIX}(24)_2 + \mathrm{SMIX}(24)_3, \mathrm{SMIX}(24)_0 + \mathrm{SMIX}(1)_0
\end{aligned}$$

If we have $j^* = 26$ and outSMIX(25) is non-zero, then it must be the case that $\mathrm{outSMIX}(25)_0 \neq 0$ and $\mathrm{outSMIX}(25)_{1,2,3} = 0$ (since we have an external collision). Further, $\mathrm{outSMIX}(25)_0 = \mathrm{SMIX}(4)_0$. Thus, inSMIX(25) after the S-Box substitutions form a non-zero codeword of $\mathcal{C}_0$, ad as before, w.l.o.g.,there is only a unique choice of this codeword. outSMIX(24) cannot be zero either, as $\mathrm{outSMIX}(25)_0$ is non-zero, and that will cause a difference in the output. Hence, as before the inSMIX(24) after the S-Box substitution forms a codeword of $\mathcal{C}^I_{0,1}$, and the combined probability is at most 2^{-132}.

If $j^* = 25$ then outSMIX(25) is zero but outSMIX(24) is non-zero. Now, $\mathrm{outSMIX}(24)_{1,2,3}$ must be zero, and a similar argument shows an upper bound of 2^{-132} on the combined probability.

Now to complete the proof, note that j^* itself is a random variable, but we show that given the initial difference D it can only assume one of five different values. First note that, for all $k \geq j^* + 2$ $\text{inSMIX}(k)_3 = 0 = \text{SMIX}(k-2)_0 + c(k)$, where $c(k)$ is a column in D as before, or $\text{outSMIX}(0)_0$ or $\text{outSMIX}(2)_0$. Since, $\text{SMIX}(k-2)_0 = 0$, by definition of j^*, we have $c(k) = 0$. On the other hand, $\text{inSMIX}(j^*+1)_3 = 0 = \text{SMIX}(j^*-1)_0 + c'$, where c' is as above, and in the notation $c(k)$ can be written as $c(j^*+1)$, and hence $c' = c(j^*+1) \neq 0$. Now, unless inSMIX(0) or inSMIX(2) is non-zero, j^* is fixed by D. Similarly, unless inSMIX(1) or inSMIX(4) is non-zero, j^* exists. Thus, a careful case analysis shows that there can be at most five cases when input differences to outSMIX(0) to outSMIX(4) are non-zero. Thus, a union bound leads to a maximum probability of 2^{-129}. ■

12 Various other Properties of Fugue-256

12.1 Pre-Image Resistance of Fugue-256

The Pre-image attack consists of finding a message which hashes to a given hash value. We expect that there is no method substantially better than brute force search to find the pre-image. We also expect that each possible output value, i.e. a 256 bit string, is almost equally likely to be output as hash value on a uniformly chosen message of large enough length. Thus, brute force search will require about 2^{256} random messages, and computation of Fugue-256 on those messages, before a hash value can be obtained which matches the given hash value.

12.2 Second Pre-Image Resistance of Fugue-256

Given a hash output H on an input P, the problem of finding another input P' which is different from P, such that the hash output on P' is also H, is called the problem of finding a second pre-image.

One possibility is to ignore the given input P, on which H was computed, and directly trying to find a pre-image of H. However, the possibility exists that the input P and the evolution of the internal state on the input P, can be used as a guide to compute a second pre-image. In fact, if at all this is possible, then it is better for the adversary to start with a same internal state $\mathbf{S}$ (i.e. before the final round $\mathbf{G}$ starts) for the two inputs P and P', and work backwards towards the same initial state, diverging somewhere in the middle (so as to make P' different different from P), and then converging back to the same state.

Thus, this can be seen as a differential attack, where one of the inputs and hence the whole state development is already fixed, and we start with a zero difference in the full 30 column state, and evolve the state backwards to a non-zero difference in the state, and then back again to a zero difference in the state. We focus on this latter part of the evolution of the state difference.

So, suppose that the rounds are numbered 0,1,2,... starting from the initial state. Further, suppose that the adversary has decided to work backwards from a common state at the end of round j. Thus, the difference in state **S** at the end of round j is zero. While working backwards, suppose the adversary introduces a difference in the state for the first time in round $j1$ $(0 < j1 < j)$, and then manages to get back to a zero difference in state in round $j2$ $(0 \leq j2 < j1)$.

We now re-number the rounds, and call round $j2$ itself as round 0, and rename round $j1$ to be round $j1 - j2$, and round j as round $j - j2$. Thus, at the end of round 1, there is a non-zero difference in state **S**. This, implies that the input difference in round 1 must be non-zero. The situation is depicted in Table 14, where we start with this input difference of a_1 in round 1, and calculate what must have been the state difference at the start of round 2 to get back a state difference of zero at the start of round 1. This evolution of the state difference is continued on to round 2, 3, etc. The adversary now tries to choose the value in column 0 which is truncated, and not the value that is input into column 0. Thus, from Table 14, the adversary tries to choose a value according to difference z_{10} for round 1.

Now there are two different ways the adversary can try to achieve this zero difference in state at the start of round 1.

1. If the adversary tries to fix the differential in state at the start of round 2 (remember, the adversary is working backwards, i.e. from state 2 to 1 to 0), then, since one of the inputs (and hence its corresponding state) is already fixed, the other input and its state also gets fixed. Then, there is no choice for the adversary to choose the (second) value in column 0 which is truncated in this round. In particular, z_{10} is already fixed, as it is required to be same as column 10 difference at the start of round 2. Thus, in this case the adversary is forced to already require all the variables a_2, z_{11}, z_{12} etc. to be such that they satisfy the constraints of the **SMIX** steps in round 1.

   However, the situation is similar in round 2 as well, as there again, the choice z_{20} is already forced by the difference of column 10 and column 21, at the start of round 3. In other words, the difference in state at the start of round 3 is y'_{11} which is same as $y_{11} + z_{20}$, and y_{11} is also required to be the difference in column 21. A similar situation holds in round 3 as well.

   It is only in round 4, that the difference z_{40} is not pre-determined, and there is a choice for z_{40}, even if the difference at the start of round 5 is fixed. However, the choice of z_{20} is constrained by a complicated set of non-linear equations of high polynomial degree over GF2.

   To elaborate, any choice of z_{40} must satisfy the following. Once, z_{40} is chosen, it forces y_{31}, as $y'_{31} = y_{31} + z_{40}$. Then, y_{31} along with all other differential values in columns 1 to 29 at the start of round 5, can be used to calculate $\hat{z}_{11}$, $\hat{z}_{12}$, $\hat{a}_{11}$, and hence to evaluate a value back for z_{40}, using the specification of **SMIX**. This value must equal the value already chosen. Thus, choosing z_{40} strategically so that the above constraints are satisfied seems to be an intractable problem, especially given that the problem gets more difficult in the next round, as there are many more similar constraints on the other variables in that round.

2. The adversary may not require the differentials to be fixed to a specific value, and try to satisfy constraints in each round by *dynamically* choosing the value truncated in column 0 in each round. However, we already saw in the previous case, that for the first 4 rounds,

the adversary has no choice for even choosing this value, as it is already determined by the difference in state at the start of the next round. Now, e.g., the probability of

$$\mathbf{SMIX}(\langle 0, 0, 0, a_1 \rangle) = \langle y_{10}, y_{11}, y_{12}, y_{13} \rangle$$

where the probability is over random a_1, y_{10}, y_{11}, y_{12}, is $(2^8)^4/(2^8)^{5\times 4} = 2^{-128}$, as **SMIX** is invertible. If we consider the two **SMIX**es of round 1, we get a probability of 2^{-256}.

Thus, it seems no easier to obtain a second pre-image of Fugue-256(P) than just inverting Fugue-256 directly without using P.

12.3 Strength of MD-Mode usage of C-Fugue-256

Note that the Compression function C-Fugue-256 always has inputs of length 16 words. However, since it accepts an IV as a parameter, the possibility exists of obtaining an internal collision with a chosen IV. In fact, we will allow two different IVs, one each for the two input messages. Collisions obtained with different inputs, as well as different IVs are called **pseudo-collisions**. Pseudo-collision resistance is important as otherwise multi-block collision attacks, in the Merkle-Damgard [14, 6] mode of using the compression function, could become a possibility.

We now show that obtaining internal pseudo-collisions for the Compression Function C-Fugue-256 is no easier than obtaining internal collisions in Fugue-256. For pseudo-collisions, the situation is similar to as analyzed in Section 10, except that an adversary can now start at the beginning of *round -j*, with j as small as 1, and with a chosen difference in $\Delta\mathbf{S}[-(j+1)]_{22..29}$ (the columns 22 to 29 are where the IV is placed). However, if the adversary starts the compression function at the beginning of *round -j*, then the other differences $\Delta\mathbf{S}[-(j+1)]_{0..21}$ must be zero, as the initial state is identical in columns 0 to 21, regardless of the IV.

However, from Table 8, it is clear that the adversary cannot start from the beginning of *round -1*, as the required difference in state at that point has columns 1 and 4 non-zero. Similarly, the adversary cannot start from the beginning of *round -2* as the difference in state at that point in column 9 is required to be non-zero. Similarly, for *rounds -3* and *rounds -4*. But, if the adversary is forced to start at the beginning of *round -5* or earlier, then the analysis is no different from that of Section 10, as there we allow conditioning on an arbitrary state difference at the beginning of *round -5* or earlier.

12.4 Analysis of PR-Fugue-256 as a PRF

Recall that PR-Fugue-256 uses the 32-byte secret key as the initial IV and then applies the underlying **F**-256 to the message (after padding and length-encoding). In this section we provide some analysis of the strength of this construction, when viewed as a pseudo-random function.

12.4.1 Linear Cryptanalysis

As was noted in Section 8.3, each of the output bytes of **F**-256 depends on every one of the 256 bytes of the state after the last **TIX** step in a "very non-linear" fashion. In fact, between every pair of state bytes and output byte there is a path of influence that goes through at least 25 **SMIX**-es. This is more than twice the number of round functions in AES-128 (and in addition Fugue has better diffusion than AES, as was seen in Section 8), which means that any linear trail in Fugue would have hundreds of active S-boxes. This seems to make linear cryptanalysis of Fugue-256 far out of reach. We did not explore this attack direction any further.

12.4.2 Differential Cryptanalysis

We will focus on resistance to differential attacks on PR-Fugue-256 in this section. One key difference in analysis of a PRF construction from that of a block cipher construction is that in the former there are no chosen ciphertext attacks. The second difference which is specific to our style of construction is that there are no sub-round keys, as the key is mixed with the input data right from the first round onwards. Thus, the usual concept of peeling away rounds from the final rounds of a block cipher, by guessing the sub-round keys of those final rounds, does not apply to differential attacks here.

Comparing with the differential cryptanalysis of obtaining collisions in Fugue-256, there are both positives and negatives for an adversary. The downside for the adversary is that it can not do any message modification, or early stopping, and in fact is assumed to consider the state at each point as random. Notice, that in PR-Fugue-256, the keys are placed in the back columns of the state, i.e. columns 22 to 29, and these are the columns which are operated on first by **SMIX** steps. On the positive side (from an adversary's perspective), the adversary now can look for arbitrary differentials, and not just ones which have output difference zero. In fact, since the AES S-box has almost perfect differential properties, except for zero differentials, the best bet for the adversary is to look for partial collisions.

Thus, for a partial output differential, which has difference zero in k bytes, in a random function, one expects that differential to hold with probability 2^{-8k}. If there is an input differential which leads to such an output differential with probability $(2^{-8k} + \epsilon)$, then by Chernoff bound, the adversary needs about $\epsilon^2 \times 2^{8k}$ random samples to distinguish the two functions with high probability.

Proving such bounds, especially for small k is a challenging task. Here we just focus on showing that it is difficult to launch distinguishing attacks with probability better than 2^{-128}.

The adversary's strategy to obtaining partial collisions may consist of starting with a zero differential in state **S**, say at initial state, or even later, and then introducing input differences, and maintaining a controlled differential in the state, whatever the probability of accomplishing that may be. We have already shown in section 10, that if the input is randomly chosen, then the probability of obtaining a full internal collision is at most 2^{-250}, and the probability of obtaining an external collision, without first obtaining an internal collision is at most 2^{-129}.

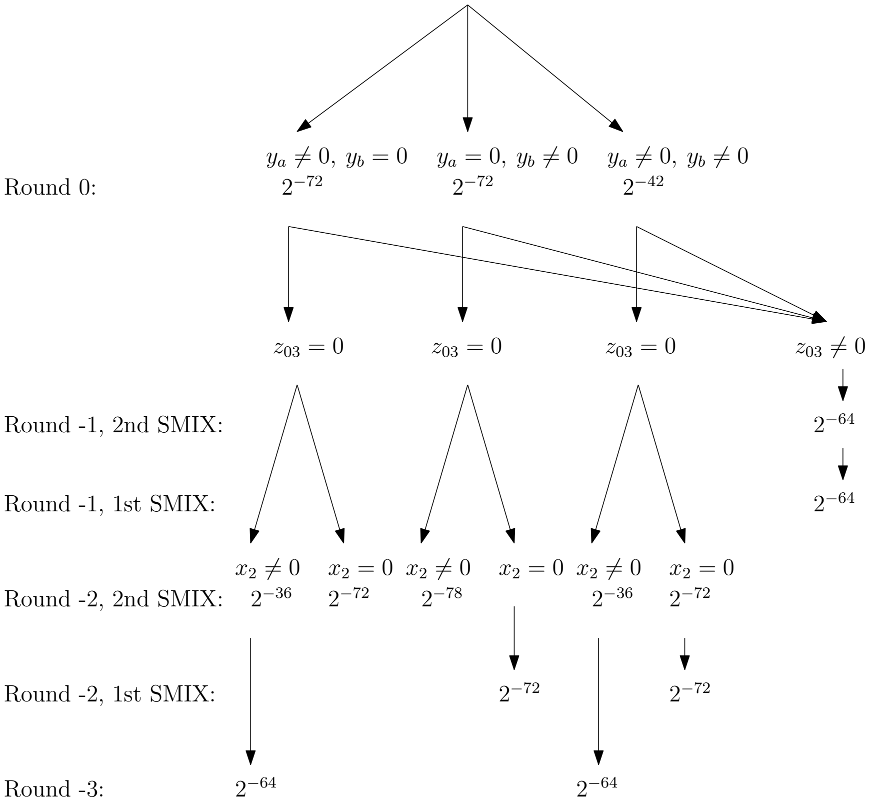

Figure 8: The Case Analysis for one kind of Partial Collision

Thus, we can ignore this possibility, and focus on the adversary obtaining a strictly partial collision. So, suppose an internal collision has not been achieved after the end of all the round transformations **R**. Then, since the final round is invertible, at every stage the state differential is non-zero. At this point, it is better to visualize the state evolution as depicted in Figure 7. Instead of rotating the state **S** by three or fifteen, here we use a sliding window on the same array of state columns, and the sliding window indicates the columns on which the **SMIX** step is performed. Thus, there are 10 **SMIX** steps in the first loop of the final transformation **G**, and 26 **SMIX** steps in the second loop of **G**.

As discussed in section 10, the adversary's success probability is increased by decreasing the total number of active bytes, i.e. bytes which are fed into an **SMIX** step, and which have a non-zero difference corresponding to two inputs. Thus, at first glance, one strategy that may be beneficial for the adversary is to have non-zero differential *only* in the last few columns of the state as depicted in Figure 7, after the first one or two **SMIX** steps of the final round (i.e. the top right most **SMIX** steps). If this can be accomplished, then there are no active bytes in the next eight **SMIX** steps of the first loop of **G**, and no active bytes in most of the **SMIX** steps of the second loop of **G**.

However, obtaining such a differential state after the first one or two **SMIX** steps of **G** is an extremely low probability event, as explained next.

We will only consider here the case of obtaining a differential state with zero everywhere except for the columns one and two output by the first **SMIX** step (i.e. columns 28, 29), at the start of the second **SMIX** step of **G**. This way, there will be *no more active bytes* till the final few **SMIX** steps of **G**. The other cases are similar, and just extend the analysis backwards, though we do not claim a proof yet.

Let us call these differences y_a and y_b for columns 28 and 29, resp. Now, we do an analysis similar to that done in Section 10 using table 8. In particular, we rename columns 28 and 29 of figure 7 to be columns 1 and 2, so that the first **SMIX** step of the final round actually operates on columns 0 to 3. The backwards evolution of the state differential is now shown in Tables 16 and 17. The first **SMIX** step of **G** will be called *round 0*, and the last input round will be called *round -1*, and so forth backwards.

Now, already in *round 0*'s **SMIX** (i.e. the first **SMIX** of **G**), we have seven active bytes, as the differences z_{00} to z_{03} after the S-Box substitutions must form a codeword of $\mathcal{C}_{1,2}$. This code has minimum distance 7, by Table 13, and since we assumed no internal collision, then one of y_a or y_b is non-zero. If one of them is zero, then at this stage itself we have 12 active bytes, as then the code is either $\mathcal{C}_1$, or $\mathcal{C}_2$, both of which have minimum distance 12.

Now, z_{03} taking a value zero is not ruled out, and indeed, that is the best option for the adversary, as otherwise there will be two consecutive $\mathcal{C}_0$ codeword requirements in *round -1*, which would lead to 26 active bytes in *round -1*.

Now, in *round -2*, if "y_a zero" option was taken before, then it is better for the adversary that x_2 be zero, for otherwise we have 13 active bytes in the second **SMIX** of this round. But, in this case y_{23} is zero, and hence in the first **SMIX** of this round there are at least 12 active bytes (assuming y_a was zero). If instead, y_a is non-zero, then it is better for the adversary to have x_2 non-zero, otherwise there are 12 active bytes in the second **SMIX** of this round. However, if x_2 is zero, then in *round -3*, for the first **SMIX** we have 12 active bytes.

Thus, in all cases, we have at least 24 active bytes, already by *round -3*. The case analysis is shown in Figure 8, from which it follows that the probability of obtaining this partial collision is at most 2^{-142}. The probabilities have been calculated as in Section 10. For example, the probability in the case $z_{03} \neq 0$, of obtaining the requisite post-condition of the second **SMIX** step of *round -1* is at most 2^{-64}, because we sum over all possibilities for z_{03}. And, using lemma 7.3, and Table 3, the maximum probability is obtained when there are no zero difference bytes in the input to this **SMIX** step, and hence the probability is at most $2^{4\times 8} * 2^{-16\times 6} = 2^{-64}$.

12.5 PR-Fugue-256 as a Universal Hash Function

Even though the property of being a universal hash function can be thought of as a weaker requirement than collision resistance, it can potentially be more difficult to prove as it is a *combinatorial* requirement in contrast to resistance to computationally bounded adversaries in usual properties like collision resistance and PRF. However, since the state of the art does not help us prove non-trivial computational lower bounds, it is reasonable to expect that proving universal hash properties is much easier.

Indeed, since in Section 10 we do prove combinatorial bounds on differential collision attacks, we now show that the same analysis leads to provable universal hash property for Fugue-256 with even weaker assumptions.

Since the universal hash function property is defined using a randomly chosen key, one can use PR-Fugue-256 to define a universal hash function. The precise definition of universal hash property is as follows. A function $f : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^m$ is called an ϵ-universal (hash) function if for all $a, b \in \{0,1\}^*$, $a \neq b$,

$$\Pr_{K \in \{0,1\}^k}[\, f(K,a) = f(K,b)] \;\leq\; \epsilon.$$

We will first focus on internal collisions. We will use notation as in Section 10.1. The main difference from the model in Section 10.2.1 is that now we do not need to assume that the state $\mathbf{S}(P)[-4]$ is chosen randomly. That the initial state is random (at least 8 words of it), instead, is now a stipulation of the universal hash property. However, we will make an *assumption* here that the whole of the initial state is random (instead of just the 8 words). If one wants to be more rigorous, a new mode can be defined where a 30 word random key is used. Or, one can assume that Fugue works as a good pseudo-random number generator, and the 8 words of initial randomness can be treated as 30 words of randomness at $\mathbf{S}(P)[-4]$. In fact, we first formally prove that no matter what the plaintext input is (i.e. a in the above definition), the initial (min-) entropy of 8 words is maintained, and hence Fugue is definitely not losing entropy as input rounds progress.

The min-entropy $\mathcal{H}(X)$ of a random variable X is defined as

$$\min \{\, \log 1/\Pr[\, X = c\,]\,\}$$

where the minimum is over all c in the support of X.

Lemma 12.1 *Consider a plaintext message* $P = P[-(m-1)], ..., P[1], P[0]$ *of length* m *words. Then,* $\mathcal{H}(\mathbf{S}(P)[0]) = \mathcal{H}(\mathbf{T}_{0..7})$*, where* $\mathbf{T}$ *is the initial state*

Proof: Recall, that by our notation the initial state $\mathbf{T}$ is $\mathbf{S}(P)[-m]$, i.e. the state before $P[-(m-1)]$ is input. Also, the state after the **TIX** step with input $P[-(m-1)]$ is denoted $\mathbf{S}[-m+\text{tix}]$. The rounds are numbered $-(m-1)$ to 0, with round 0 corresponding to input $P[0]$.

Note that the first step of a round is the **TIX** step, and the first sub-step of the **TIX** step is to add $\mathbf{S}_0$ to $\mathbf{S}_{10}$. It will be convenient to view the end of a round to include this first sub-step of the next round. So, for the purpose of this lemma we will use $\mathbf{S}(P)[-(m-i)]$ to denote the state after round $-(m-i)$ plus the sub-step of next round.

Thus, since in the initial state $\mathbf{T}_{8..29}$ is set to zero, this implies that the min-entropy of $\mathbf{S}[-m]_{1..29}$ is $\mathcal{H}(\mathbf{T}_{0..7})$ which is $8 * 32$.

Hence, we will actually show something stronger, which is that the min-entropy in words 1 to 29 of the state is $8 * 32$ (instead of words 0 to 29). We prove this by induction over $0 \leq i \leq m$, with the following induction hypothesis: for every 29 word constant c,

$$\Pr[\, \mathbf{S}(P)[-(m-i)]_{1..29} \;=\; c\,] \leq 2^{-8*32}$$

The base case ($i = 0$) holds by the previous paragraph.

So, now suppose the hypothesis holds for i, and we will prove it for $i + 1$. We show that for every possible value of $\mathbf{S}(P)[-(m-i-1)]_{1..29}$, there is at most one value of $\mathbf{S}(P)[-(m-i)]_{1..29}$ which along with $P[-(m-i-1)]$ leads to the former, from which the induction step follows.

Suppose to the contrary, there are two values $S1$ and $S2$ for $\mathbf{S}(P)[-(m-i)]_{1..29}$ which along with the same $P[-(m-i-1)]$ lead to the same value T for $\mathbf{S}(P)[-(m-i-1)]_{1..29}$. If we look at the differentials, this implies that the difference in words 1, 2 and 3 of $\mathbf{S}(P)[-(m-i-1)]_{1..29}$ is zero. Then, as in section 10 it follows that all four words in $\mathbf{S}(P)[-(m-i)+\text{tix}]_{1..29}$ must have non-zero difference. a contradiction. ■

A similar argument shows that if the full initial state $\mathbf{T}$ is chosen randomly, then the state after every round has min-entropy $8 * 29$. In the following, we will assume that the adversary chooses two messages of the same length. In the case that the adversary chooses messages of different lengths, say $m_1 > m_2$, then the above analysis shows that one can just start after incorporating the first $m_1 - m_2$ blocks of input of the longer message into the random state.

Universal Internal-Collision Attack:

1. The attacker chooses two messages P and P', each of equal length m.
2. The initial state $\mathbf{T}$ is chosen at random.
3. The attack is successful if starting from the initial state the two messages induce an internal collision in round 0 (namely $\mathbf{S}(P)[-1] \neq \mathbf{S}(P')[-1]$ but $\mathbf{S}(P)[0] = \mathbf{S}(P')[0]$).

Theorem 12.2 *The success probability of an attacker in the Universal Internal-Collision Attack is at most* 2^{-130}.

Proof: The proof will use the analysis of both the pure differential attack (Theorem 10.2), and the semi-pure differential attack (Theorem 10.3). Further, we will use the improved analysis of Section 10.3.

As opposed to these earlier theorems, we will only consider rounds -1 and -2, and hence only $\mathbf{SMIX[0]}$, $\mathbf{SMIX[-0.5]}$, $\mathbf{SMIX[-1]}$, and $\mathbf{SMIX[-1.5]}$. Further, as the plaintext is fixed before the random initial state is chosen, instead of charging 2^{32} per input round as in Theorem 10.3, we can charge 2^{6*4}, essentially giving the 4 possible S-Box differentials for free.

Thus, the probability of finding a collision is $2^{-124} \times 2^{-36} \times 2^{-64}$ times $2^{24} \times 2^{24} \times 2^{24}$, where 2^{-124} is the probability for $\mathbf{SMIX[0]}$, and $\mathbf{SMIX[-1.5]}$ (see section 10.3), 2^{-36} is the probability for **SMIX**$[-\mathbf{1}]$, and 2^{-64} is the probability for **SMIX**$[-\mathbf{0.5}]$. The factor $2^{24} \times 2^{24} \times 2^{24}$ comes from the two input rounds, and the price for eliminating the independence assumption used in **SMIX**$[-\mathbf{1}]$. ■

The theorem for universal external collision is same as Theorem 11.1, and gives a bound of 2^{-129}. In this case we are not able to get rid of the independence assumption of Section 11, but we can consider the probability under choosing the initial state at random (as opposed to at the beginning of the TIX-less rounds in Theorem 11.1).

13 Other Security Considerations

13.1 A Meet-in-the-Middle Attack on Fugue

As we mentioned in the introduction, Fugue is not subject to many of the "generic attacks" that are known against Merkle-Damgard functions. The only generic attack that we are aware of for Fugue is the meet-in-the-middle attack that is described below.

Since the round functions of Fugue are invertible, there is an easy pre-image attack with time- and space-complexity roughly $2^{m/2}$ where m is the state size (in bits).[7] Below we describe the attack for the underlying **F**-256, but similar attacks apply to all version of Fugue (even with padding an length encoding).

Given the output Y from **F**-256 (which was taken from columns 1-4 and 15-18 of the state) we set up the following two processes:

- The first process just choose many different messages (say, of length more than 30 words) and records each message with the state that it induces after incorporating the whole message (but before the final round).
- The second process fills the state with the output Y in columns 1-4 and 15-18 and with random bytes elsewhere, and then evolves that random state backward to just before the beginning of the final round.

After roughly $2^{m/2}$ trials for each process, we expect to find a collision. That is, a message M that leads to a state $\mathbf{S}$ just before the final round, such that when going through the final round we end up with exactly Y in the right columns of the final state.

As we said, this attack has complexity (both time and space) of about $2^{m/2}$. Namely, we have complexity 2^{480} for Fugue-256 and Fugue-224 and complexity 2^{576} for Fugue-384 and Fugue-512.

---

[7] This attack is the reason for the requirement $s \geq 2n$ in the parametrized version of Fugue.

13.2 Side-channel Cryptanalysis Attacks

Since Fugue has very similar implementations to AES, we expect it to exhibit the same characteristics as AES with respect to size-channel attacks. On one hand, this means that Fugue is likely to be vulnerable to similar cache attacks as AES [1, 19]. On the flip side, it is likely that the same counter-measures that are employed to protect AES from side-channel attacks will also be applicable to Fugue.

We also mention that side-channel attacks are only applicable to hash functions when they are used with a secret key, and have no bearing on "integrity properties" such as collision resistance or second-pre-image resistance.

14 Cryptanalysis of wFugue-256

In this section we focus on the possibilities of internal collisions in w**F**-256. As for **F**-256, the backwards evolution of the differential state is shown in tables 18 and 19.

However, to get a tighter bound, we need to capitalize on the fact that there are even more constraints on the differential state variables than analyzed in section 10. We will use notation similar to that in Section 10. From Table 18, it is clear that the (differential) variables input to the **SMIX** in *round -2* after the S-Box substitutions form a codeword of $\mathcal{C}^I_{0,1}$, as shown in Section 10.3. This allowed us to prove that if there were $k > 0$ zeroes in the input to the **SMIX** in *round -1*, then there were more than 6 non-zeroes in the input to the **SMIX** in *round -2*. The precise relation was given by table 11.

However, in this section, we do a similar analysis for two other similar situations. The first is simpler to describe, and is essentially the same as described in the previous paragraph, except we consider the case where x_1 is zero. Then, the code for the input variables for the Super-Mix in *round -2* is called $\mathcal{C}^I_1$, and its minimum maxmin-rank, over all I, is given as follows (as checked by a computer program):

$$\begin{aligned} |I| = 0 &: 11 \\ |I| = 1 &: 12 \\ |I| = 2 &: 12 \\ |I| = 3 &: 14 \end{aligned}$$

In other words, if there are three zeros in the input to **SMIX** which produces $\langle X_0, 0, 0, 0\rangle$, then the input to an **SMIX** that produces $\langle 0, X_0, 0, 0\rangle$ has minimum weight 14+1.

The second situation is more difficult to analyze, though the code description is simple. If an **SMIX** which produces an output of $\langle a, b, 0, 0\rangle$ has k zeroes in its input, say, in indices I, then the inputs to the linear transformation Super-Mix that produce $\langle c, a, 0, 0\rangle$ as output for any c, are called codewords of code $\mathcal{C}^{I*}_{0,1}$. The minimum maxmin-rank, over all I, for such codes is given as follows (and as checked by a computer program as described below):

$$\begin{aligned}|I| = 7 &: 8\\ |I| = 8 &: 8\\ |I| = 9 &: 9\\ |I| = 10 &: 10\end{aligned}$$

The code $\mathcal{C}_{0,1}^{I*}$ has parity check equations which can be described as follows. To start with, say,

$$\mathbf{N} \cdot \langle \hat{Y}_{10},\ \hat{Y}_{11},\ \hat{Y}_{12},\ \hat{Y}_{13} \rangle^{\mathrm{T}} = \langle x_1,\ x_0,\ 0,\ 0 \rangle^{\mathrm{T}}$$

Since $\mathbf{N}$ is invertible, this implies,

$$\langle \hat{Y}_{10},\ \hat{Y}_{11},\ \hat{Y}_{12},\ \hat{Y}_{13} \rangle^{\mathrm{T}} = \mathbf{N}^{-1} \cdot \langle x_1,\ x_0,\ 0,\ 0 \rangle^{\mathrm{T}}$$

Now, if the $\hat{Y}$ byte variables have k zeroes, say in byte indices I, then it imposes that many constraints on x_1, x_0, via the first eight columns, and the k rows I of $\mathbf{N}^{-1}$. Let this $k \times 8$ submatrix of $\mathbf{N}^{-1}$ be called N-inv$_I$. Thus, N-inv$_I \cdot \langle x_1, x_0 \rangle = 0$. However, from these equations, by Gaussian elimination, we can obtain constraints just on x_1, though how many such linearly independent constraints can be obtained is only checked by a computer (for each I). so suppose we obtain k_I constraints on x_1, and let these constraints be given by $k_1 \times 4$ matrix N-inv-reduced$_I$.

Thus, the code $\mathcal{C}_{0,1}^{I*}$ has $8 + k_I$ parity check equations given by $\mathbf{N}^{8..15}$ and N-inv-reduced$_I\mathbf{N}^{4..7}$. A computer program can then compute the maxmin-rank of each such code.

Equipped with these additional bounds on the ranks of various codes, and with the help of table 18, we first upper bound the probability of finding a collision under the assumption that the inputs are chosen randomly and independently (as one in Section 10.2.1).

In this section we will also assume that the probability of any non-zero differential of the S-Box is 2^{-8}, as indeed that is the average probability over all differentials. This is a reasonable assumption, as it is difficult for the adversary to fix non-zero parts of the state differentials to his choice, without incurring additional cost. Moreover, for all differentials (i.e. differential characteristics for an **SMIX**) that we consider the output difference is determined by the full state difference before the start of the **SMIX** step, and hence there is only one choice for a codeword that leads to that output difference. Thus, we can just count the number of active bytes.

In each *round -j*, let the number of active bytes be denoted by $16 - k_j$. Thus, in *round -1*, let there be $16 - k_1$ active bytes. Indeed $k_1 \leq 3$, as the code corresponding to the **SMIX**in this round is $\mathcal{C}_0$. In *round -2*, if x_1 is zero, then $k_2 \leq 4$, as the code corresponding to this **SMIX** is then $\mathcal{C}_1$. If x_1 is non-zero, then the code is $\mathcal{C}_0^I$, where I is the indices in *round -1*— which correspond to zero input byte differences ($|I| = k_1$).

In *round -3*, if x_1 is zero, the best option for the adversary is to take x_2 to be zero. If, on the other hand x_1, then it is best for the adversary to have x_2 non-zero, in which case the code for this round is $\mathcal{C}_{0,1}^{J*}$, where J is the indices in *round-2* which correspond to zero input byte differences ($|J| = k_2$).

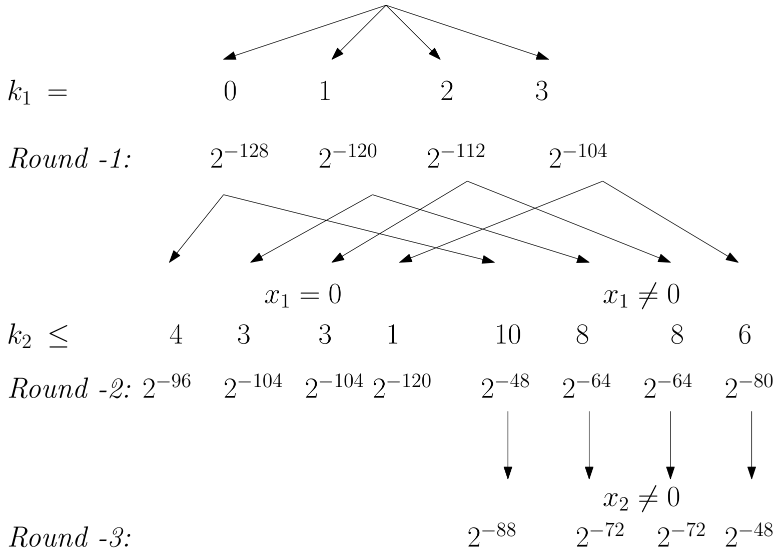

Figure 9: Case Analysis for w**F**-256

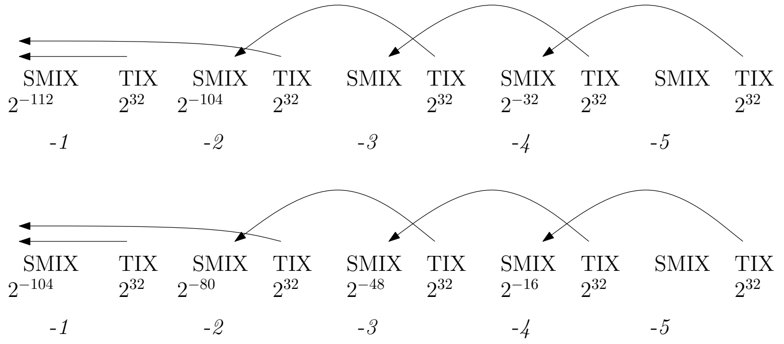

Figure 10: Internal Collision Probability for w**F**-256 assuming Free Message Modification for two **SMIX**es

In *round -4*, if x_2 (and x_1) were zero, then since Y_{03} is necessarily non-zero, it is best for the adversary to have x_3 non-zero, in which the code for this round is $\mathcal{C}_{0,2}$.

The case analysis is depicted in Figure 9. The situation under the weak adversarial advantage assumption of free message modification upto two **SMIX**es is shown in Figure 10. Note that at each round shown, there are only 2^{32} possibilities for the input pairs, as the differentials of the inputs are already determined by the state differential before the **TIX** step. Thus, for example, the input pair difference for the **TIX** in *round -1* is Y_{03}, but that is required to be same as $\Delta\mathbf{S}[-2]_8$. Similar situation holds for all **TIX** steps till *round -5*.

In Figure 10, two situations are shown, corresponding to $x_1 = 0$ and $x_1 \neq 0$ resp. Now, first we consider the weak adversarial advantage assumption of free message modification upto only **one SMIX**. This is not an unreasonable assumption, as we could not build a small enough pre-computed table (i.e. one which is practical in the foreseeable future) which allows us to do free message modification beyond one **SMIX** step. Under this assumption, the probability of internal collision is at most 2^{-152} in the top case, and at most 2^{-136} in the bottom case.

If on the other hand, we assume that upto two **SMIX**es free message modification is feasible, and beyond that is infeasible, then we get (naively) an upper bound of 2^{-120} in the top case, and an upper bound of 2^{-104}. However, this assumes, e.g. in the bottom case, that input in *round -4* can satisfy all the constraints in the **SMIX** of *round -3*, and the inputs of *round -3* remain neutral w.r.t. satisfaction of these constraints, and can be used as control bytes for constraints on the **SMIX** in *round -2*. If the input bytes of *round -3* do not remain neutral (and we could not see a situation where they could be), then the probability of collision is upper bounded by 2^{-120}.

Thus, it is safe to claim that the workload of finding an internal collision is at least 2^{96} w**F**-256 computations on messages of 64 bytes, even with pre-computed tables of size 2^{96} bits.

15 Strength of Fugue-224, Fugue-384 and Fugue-512

15.1 Collision Resistance of F-224

Since, Fugue-224 is identical in structure to Fugue-256, except that only a subset of seven columns of the output of Fugue-256 is output, the probability of obtaining an internal collision is upper bounded exactly as for Fugue-256. The analysis for probability of obtaining an external collision in Fugue-224 is not as clean as that for Fugue-256, and the reason is similar to that described in the section 12.4.2 on differential attacks on PR-Fugue-256. However, as shown there, it is difficult to obtain any useful partial collision before the start of the final round, and hence we do not expect the probability of obtaining an external collision in Fugue-224 to be more than 2^{-112}.

15.2 Collision Resistance of F-384

In this document, we focus solely on the possibilities of internal collisions in Fugue-384, and in particular in **F**-384. The analysis is similar to that for **F**-256 in Section 10, and the (backwards)

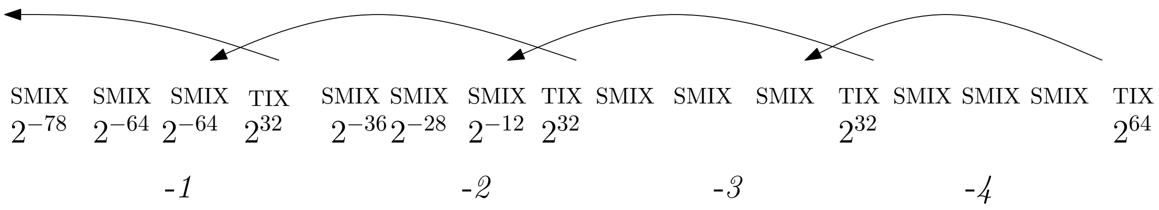

Figure 11: Internal Collisions in **F**-384 with Free Message Modification upto 4 SMIXes

evolution of the differential state required for an internal collision is shown in Tables 20 and 21. As before, all differences which are necessarily non-zero are shown in capital letters.

Since, the input word difference in *round -1* is W_{03}, and since the specification of **F**-384 requires the input word to be XOR-ed into column 8, we have that $W_{03} + \Delta\mathbf{S}[-2]_8 = 0$. Hence, the input difference for this round is already fixed, if we condition on $\Delta\mathbf{S}[-2]$ being a fixed value.

Similarly, the input pair difference for *rounds -2* and *-3* are fixed. Thus, in these rounds there are only 2^{32} possibilities for the pair of input words. This is shown in Figure 11, where the probabilities of satisfying the various post-conditions of the different **SMIX** steps are also shown (analyzed in a similar fashion as done in Section 10).

From Figure 11, it follows that under the multiplicative differential assumption, the probability of obtaining an internal collision is at most 2^{-282}, where the probability is over a random state at the end of *round -j* ($j > 5$), and random inputs, and conditioned on any fixed state differential at the end of *round -j*.

On the other hand, since the adversary may choose its inputs strategically, we consider the situation where we allow it to do *free message modification upto four* **SMIX**es. In that case, the probability is still upper bounded by $2^{-282+32+32+12} = 2^{-206}$.

15.3 Collision Resistance of F-512

In this document, we focus solely on the possibilities of internal collisions in Fugue-512, and in particular in **F**-512. The analysis is similar to that for **F**-256 in Section 10, and the (backwards) evolution of the differential state required for an internal collision is shown in Tables 22 and 23. As before, all differences which are necessarily non-zero are shown in capital letters.

Since, the input word difference in *round -1* is U_{03}, and since the specification of **F**-512 requires the input word to be XOR-ed into column 8, we have that $U_{03} + \Delta\mathbf{S}[-2]_8 = 0$. Hence, the input difference for this round is already fixed, if we condition on $\Delta\mathbf{S}[-2]$ being a fixed value.

Similarly, the input pair difference for *round -2* is fixed. Thus, in these rounds there are only 2^{32} possibilities for the pair of input words. This is shown in Figure 12, where the probabilities of satisfying the various post-conditions of the different **SMIX** steps are also shown (analyzed in a similar fashion as done in Section 10).

From Figure 12, it follows that under the multiplicative differential assumption, the probability of obtaining an internal collision is at most 2^{-374}, where the probability is over a random state

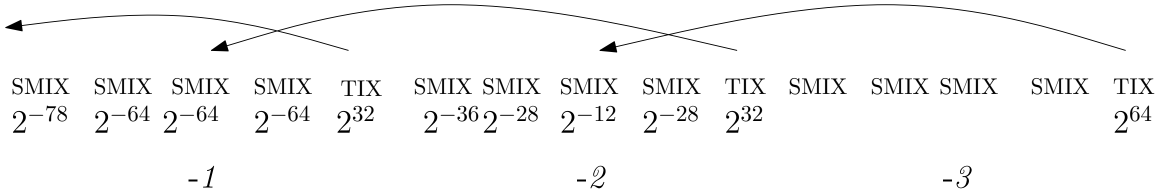

Figure 12: Internal Collisions in **F**-512 with Free Message Modification upto 6 SMIXes

at the end of *round -j* ($j > 3$), and random inputs, and conditioned on any fixed state differential at the end of *round -j*.

On the other hand, since the adversary may choose its inputs strategically, we consider the situation where we allow it to do *free message modification upto six* **SMIX**es. In that case, the probability is still upper bounded by $2^{-374+32+32+28+12} = 2^{-270}$. Note that it already gives an additional advantage to the adversary that the input bits of *round -2* are neutral for the constraints set in the first and second **SMIX** steps of *round -2* using control bits from inputs in *round -3*.

16 On the Choice of the Matrix M

The 4×4 matrix **M** of Section 4.2 has been chosen as follows. Note that its first column can be written as **1174**, i.e. a four digit hexadecimal number. We considered all circulant matrices, and hence determined by their first columns, with each of the four GF(2^8) values in the column, i.e. M_0^0, M_0^1, M_0^2, and M_0^3, being equivalent to a single digit hexadecimal number.

Next, the smallest such four digit number (with M_0^0 being the most significant digit) such that the resulting matrix **M**, and the derived matrix **N**, satisfy the following constraints was chosen.

- The matrix **N** is invertible,
- the matrix **M** itself is MDS, i.e. all its square sub-matrices are non-singular,
- the code $\mathcal{C}_0$ corresponding to **N** is MDS,
- the maxmin-rank of codes $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ are at least 11,
- the maxmin-rank of code $\mathcal{C}_{0,1}$ is at least 5,
- the minimum over all I, $|I| = 3$, of the maxmin-rank of all codes $\mathcal{C}_{0,1}^{I}$ is at least 9,
- the minimum over all I, $|I| = 1, 2$, of the maxmin-rank of all codes $\mathcal{C}_{0,1}^{I}$ is at least 7.

References

[1] Daniel J. Bernstein. Cache timing attacks on AES. Available on-line from `http://cr.yp.to/papers.html#cachetiming`. 2005.

[2] Brian Gladman. `http://fp.gladman.plus.com/cryptography_technology/index.htm`

[3] Eli Biham, Rafi Chen. Near-Collisions of SHA-0. *Advances in Cryptology - CRYPTO '04*, Lecture Notes in Computer Science Vol. 3152, Springer, 2004, pp. 290-305.

[4] L. Carlitz, S. Uchiyama. Bounds for Exponential Sums. *Duke Math. J.*, 24, 1957, 37-41.

[5] J. Daeman, V. Rijmen, "The Design of Rijndael: AES - The Advanced Encryption Standard." Springer-Verlag, 2002.

[6] Ivan Damgard. A Design Principle for Hash Functions. *Advances in Cryptology - CRYPTO '89*, Lecture Notes in Computer Science Vol. 435, Springer-Verlag, 1989, pp. 416-427.

[7] Yedidya Hilewitz, Yiqun Lisa Yin, Ruby B. Lee. Accelerating the Whirlpool Hash Function Using Parallel Table Lookup and Fast Cyclical Permutation. *Fast Software Encryption - FSE'08*, Lecture Notes in Computer Science Vol. 5086, Springer, 2008, pp. 173-188.

[8] Antoine Joux. Multi-collisions in Iterated Hash Functions. Application to Cascaded Constructions. *Advances in Cryptology - CRYPTO '04*, Lecture Notes in Computer Science Vol. 3152, Springer, 2004, pp. 306-316.

[9] Antoine Joux, Thomas Peyrin. Hash Functions and the (Amplified) Boomerang Attack *Advances in Cryptology - CRYPTO'07*, Lecture Notes in Computer Science Vol. 4622, Springer, 2004, pp. 244-263.

[10] Charanjit S. Jutla, Anindya C. Patthak. Provably Good Codes for Hash Function Design. *Selected Areas in Cryptography - SAC '06*, Lecture Notes in Computer Science Vol. 4356, Springer, 2006, pp. 376-393.

[11] Geoffrey Keating. Performance Analysis of AES candidates on the 6805 CPU core Available on-line from `http://csrc.nist.gov/archive/aes/round1/pubcmnts.htm`

[12] John Kelsey, Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. *Advances in Cryptology - EUROCRYPT '06*, Lecture Notes in Computer Science Vol. 4004, Springer, 2006, pp. 183-200.

[13] Lars R. Knudsen, Christian Rechberger, Soren S. Thomsen. The Grindahl Hash Functions. *Fast Software Encryption - FSE '07*, Lecture Notes in Computer Science Vol. 4593, Springer, 2007, pp. 39-57.

[14] Ralph C. Merkle. A Certified Digital Signature. *Advances in Cryptology - CRYPTO '89*, Lecture Notes in Computer Science Vol. 435, Springer-Verlag, 1989, pp. 218-238.

[15] Kaisa Nyberg. Differentially Uniform Mappings for Cryptography. *Advances in Cryptology - Eurocrypt 1993*.

[16] NIST. Advanced Encryption Standard. FIPS 197, November 2001.

[17] NIST. The Keyed-Hash Message Authentication Code (HMAC). FIPS 198-1, July 2008.

[18] NIST. DRAFT Randomized Hashing Digital Signatures SP 800-106, Jul 31, 2008

[19] Dag Arne Osvik, Adi Shamir, Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. *The Cryptographers' Track at the RSA Conference - CT-RSA '06*, Lecture Notes in Computer Science Vol. 3860, Springer, 2006, pp. 1-20

[20] Thomas Peyrin. Cryptanalysis of Grindahl. *Advances in Cryptology - ASIACRYPT '07*, Lecture Notes in Computer Science Vol. 4833, Springer, 2007, pp. 551-567.

[21] Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, Pankaj Rohatgi. Efficient Rijndael Encryption Implementation with Composite Field Arithmetic. *Workshop on Cryptographic Hardware and Embedded Systems - CHES'01*, Lecture Notes in Computer Science Vol. 2162, Springer, 2001, pp. 171-184.

[22] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. *Advances in Cryptology - EUROCRYPT '05*, Lecture Notes in Computer Science Vol. 3494, Springer, 2007, pp. 1-18.

[23] Xiaoyun Wang, Hongbo Yu. How to Break MD5 and Other Hash Functions. *Advances in Cryptology - EUROCRYPT '05*, Lecture Notes in Computer Science Vol. 3494, Springer, 2007, pp. 19-35.

Table 12: The Fugue S-Box[ ]. The Fugue S-box is identical to the AES S-box, and is specified below as a table of 256 bytes. Note that the table is given in *row order.* that is the first few bytes in the table are S-box[**00**] =**63**, S-box[**01**]=**7c**, S-box[**02**] =**77**,etc.

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

Table 13: Min-Rank$_m$ values for Various Linear Codes (Expanded)

m	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\mathcal{C}_\phi$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	16
$\mathcal{C}_0$	-	-	-	-	-	-	-	-	-	-	-	12	12	12	12	12
$\mathcal{C}_1$	-	-	-	-	-	-	-	-	-	-	11	11	12	12	12	12
$\mathcal{C}_2$	-	-	-	-	-	-	-	-	-	-	11	11	12	12	12	12
$\mathcal{C}_3$	-	-	-	-	-	-	-	-	-	-	11	11	12	12	12	12
$\mathcal{C}_{0,1}$	-	-	-	-	5	5	6	6	7	7	8	8	8	8	8	8
$\mathcal{C}_{0,2}$	-	-	3	3	4	4	5	6	7	7	7	8	8	8	8	8
$\mathcal{C}_{0,3}$	-	-	-	-	5	5	6	6	7	7	7	8	8	8	8	8
$\mathcal{C}_{1,2}$	-	-	-	-	-	6	6	7	7	7	8	8	8	8	8	8
$\mathcal{C}_{1,3}$	-	-	-	-	5	5	6	6	7	7	8	8	8	8	8	8
$\mathcal{C}_{2,3}$	-	-	3	3	4	5	6	6	7	7	7	8	8	8	8	8
$\mathcal{C}_{0,1,2}$	1	1	2	2	2	2	3	3	3	3	4	4	4	4	4	4

[†] The shaded entries are the maxmin-rank.

Table 14: Evolution of Differential State for Second Pre-Image(Forward)

	0			3			6			9			12			15			18			21			24			27		29
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TIX$_1$																														
	a_1								a_1																					
ROR3																														
				a_1								a_1																		
CMIX																														
				a_1								a_1																		
SMIX																														
	y_{10}	y_{11}	y_{12}	y_{13}								a_1																		
ROR3																														
				y_{10}	y_{11}	y_{12}	y_{13}								a_1															
CMIX																														
	y_{11}	y_{12}	y_{13}	y_{10}	y_{11}	y_{12}	y_{13}								a_1	y_{11}	y_{12}	y_{13}												
SMIX																														
	z_{10}	z_{11}	z_{12}	z_{13}	y_{11}	y_{12}	y_{13}								a_1	y_{11}	y_{12}	y_{13}												
TIX$_2$																														
	a_2	z_{11}	z_{12}	z_{13}	y_{11}	y_{12}	y_{13}		a_2		z_{10}				a_1	y_{11}	y_{12}	y_{13}												
ROR3																														
				a_2	z_{11}	z_{12}	z_{13}	y_{11}	y_{12}	y_{13}		a_2		z_{10}				a_1	y_{11}	y_{12}	y_{13}									
CMIX																														
	z_{11}	z_{12}	z_{13}	a_2	z_{11}	z_{12}	z_{13}	y_{11}	y_{12}	y_{13}		a_2		z_{10}		z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}									
SMIX																														
	y_{20}	y_{21}	y_{22}	y_{23}	z_{11}	z_{12}	z_{13}	y_{11}	y_{12}	y_{13}		a_2		z_{10}		z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}									
ROR3																														
				y_{20}	y_{21}	y_{22}	y_{23}	z_{11}	z_{12}	z_{13}	y_{11}	y_{12}	y_{13}		a_2		z_{10}		z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}						
CMIX																														
	y_{21}	y_{22}	y_{23}	y_{20}	y_{21}	y_{22}	y_{23}	z_{11}	z_{12}	z_{13}	y_{11}	y_{12}	y_{13}		a_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}						
SMIX																														
	z_{20}	z_{21}	z_{22}	z_{23}	y_{21}	y_{22}	y_{23}	z_{11}	z_{12}	z_{13}	y_{11}	y_{12}	y_{13}		a_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}						
TIX$_3$																														

Table 15: Evolution of Differential State for Second Pre-Image(Contd.)

	0			3			6			9			12			15			18			21			24			27		29
	z_{20}	z_{21}	z_{22}	z_{23}	y_{21}	y_{22}	y_{23}	z_{11}	z_{12}	z_{13}	y_{11}	y_{12}	y_{13}		a_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}						
TIX$_3$																														
	a_3	z_{21}	z_{22}	z_{23}	y_{21}	y_{22}	y_{23}	z_{11}	z'_{12}	z_{13}	y'_{11}	y_{12}	y_{13}		a_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}						
ROR3																														
				a_3	z_{21}	z_{22}	z_{23}	y_{21}	y_{22}	y_{23}	z_{11}	z'_{12}	z_{13}	y'_{11}	y_{12}	y_{13}		a_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}			
CMIX																														
	z_{21}	z_{22}	z_{23}	a_3	z_{21}	z_{22}	z_{23}	y_{21}	y_{22}	y_{23}	z_{11}	z'_{12}	z_{13}	y'_{11}	y_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}			
SMIX																														
	y_{30}	y_{31}	y_{32}	y_{33}	z_{21}	z_{22}	z_{23}	y_{21}	y_{22}	y_{23}	z_{11}	z'_{12}	z_{13}	y'_{11}	y_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}			
ROR3																														
				y_{30}	y_{31}	y_{32}	y_{33}	z_{21}	z_{22}	z_{23}	y_{21}	y_{22}	y_{23}	z_{11}	z'_{12}	z_{13}	y'_{11}	y_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}
CMIX																														
	y_{31}	y_{32}	y_{33}	y_{30}	y_{31}	y_{32}	y_{33}	z_{21}	z_{22}	z_{23}	y_{21}	y_{22}	y_{23}	z_{11}	z'_{12}	z'_{13}	y''_{11}	y'_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}
SMIX																														
	z_{30}	z_{31}	z_{32}	z_{33}	y_{31}	y_{32}	y_{33}	z_{21}	z_{22}	z_{23}	y_{21}	y_{22}	y_{23}	z_{11}	z'_{12}	z'_{13}	y''_{11}	y'_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}
TIX$_4$																														
	a_4	z'_{31}	z_{32}	z_{33}	y_{31}	y_{32}	y_{33}	z_{21}	z'_{22}	z_{23}	y'_{21}	y_{22}	y_{23}	z_{11}	z'_{12}	z'_{13}	y''_{11}	y'_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1	y_{11}	y_{12}	y_{13}
ROR3																														
	y_{11}	y_{12}	y_{13}	a_4	z'_{31}	z_{32}	z_{33}	y_{31}	y_{32}	y_{33}	z_{21}	z'_{22}	z_{23}	y'_{21}	y_{22}	y_{23}	z_{11}	z'_{12}	z'_{13}	y''_{11}	y'_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1
CMIX																														
	$\hat{y}_{11}$	$\hat{y}_{12}$	$\hat{y}_{13}$	a_4	z'_{31}	z_{32}	z_{33}	y_{31}	y_{32}	y_{33}	z_{21}	z'_{22}	z_{23}	y'_{21}	y_{22}	y''_{23}	z'_{11}	z''_{12}	z'_{13}	y''_{11}	y'_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1
SMIX																														
	y_{40}	y_{41}	y_{42}	y_{43}	z'_{31}	z_{32}	z_{33}	y_{31}	y_{32}	y_{33}	z_{21}	z'_{22}	z_{23}	y'_{21}	y_{22}	y''_{23}	z'_{11}	z''_{12}	z'_{13}	y''_{11}	y'_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}	z_{11}	z_{12}	a'_1
ROR3																														
	z_{11}	z_{12}	a'_1	y_{40}	y_{41}	y_{42}	y_{43}	z'_{31}	z_{32}	z_{33}	y_{31}	y_{32}	y_{33}	z_{21}	z'_{22}	z_{23}	y'_{21}	y_{22}	y''_{23}	z'_{11}	z''_{12}	z'_{13}	y''_{11}	y'_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}
CMIX																														
	$\hat{z}_{11}$	$\hat{z}_{12}$	$\hat{a}_1$	y_{40}	y_{41}	y_{42}	y_{43}	z'_{31}	z_{32}	z_{33}	y_{31}	y_{32}	y_{33}	z_{21}	z'_{22}	z'_{23}	y''_{21}	y'_{22}	y''_{23}	z'_{11}	z''_{12}	z'_{13}	y''_{11}	y'_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}
SMIX																														
	z_{40}	z_{41}	z_{42}	z_{43}	y_{41}	y_{42}	y_{43}	z'_{31}	z_{32}	z_{33}	y_{31}	y_{32}	y_{33}	z_{21}	z'_{22}	z'_{23}	y''_{21}	y'_{22}	y''_{23}	z'_{11}	z''_{12}	z'_{13}	y''_{11}	y'_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}
TIX$_5$																														
	a_5	z''_{41}	z_{42}	z_{43}	y_{41}	y_{42}	y_{43}	z'_{31}	z'_{32}	z_{33}	y'_{31}	y_{32}	y_{33}	z_{21}	z'_{22}	z'_{23}	y''_{21}	y'_{22}	y''_{23}	z'_{11}	z''_{12}	z'_{13}	y''_{11}	y'_{12}	y'_{13}	z_{22}	a'_2	y_{21}	z'_{10}	y_{23}
ROR3																														

[a] Primed and hatted variables should be easy to deduce. In general, the number of primes indicate the number of other summands.

Table 16: Evolution of Differential State for one type of Partial Collision (Backwards)

	0			3			6			9			12			15			18			21			24			27		29
	0	0	0	0	y_a	y_b																								
CMIX																														
	y_a	y_b	0	0	y_a	y_b										y_a	y_b													
ROR3																														
	0	y_a	y_b	0									y_a	y_b														y_a	y_b	0
SMIX																														
	z_{00}	z_{01}	z_{02}	z_{03}									y_a	y_b														y_a	y_b	0
CMIX																														
	z_{00}	z_{01}	z_{02}	z_{03}									y_a	y_b														y_a	y_b	0
ROR3																														
G starts																														
	z_{03}	0	0	0						y_a	y_b														y_a	y_b	0	z_{00}	z_{01}	z_{02}
SMIX																														
	y_{10}	y_{11}	y_{12}	y_{13}						y_a	y_b														y_a	y_b	0	z_{00}	z_{01}	z_{02}
CMIX																														
	y_{10}	y_{11}	y_{12}	y_{13}						y_a	y_b														y_a	y_b	0	z_{00}	z_{01}	z_{02}
ROR3																														
	y_{13}	0	0	0			y_a	y_b														y_a	y_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}
SMIX																														
	z_{10}	z_{11}	z_{12}	z_{13}			y_a	y_b														y_a	y_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}
CMIX																														
	z_{10}	z_{11}	z'_{12}	z_{13}			y_a	y_b										y_a				y_a	y_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}
ROR3																														
	z_{13}			y_a	y_b										y_a				y_a	y_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}
TIX$_{-1}$																														
	x_2	0	0	y_a	y_b				z_{13}		x_2				y_a				y_a	y_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}
SMIX																														
	y_{20}	y_{21}	y_{22}	y_{23}	y_b				z_{13}		x_2				y_a				y_a	y_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}
CMIX																														
	y'_{20}	y_{21}	y_{22}	y_{23}	y_b				z_{13}		x_2				y_a	y_b			y_a	y_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}
ROR3																														
	y_{23}	y_b	0	0		z_{13}		x_2				y_a	y_b			y_a	y_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}	y'_{20}	y_{21}	y_{22}
SMIX																														
	z_{20}	z_{21}	z_{22}	z_{23}		z_{13}		x_2				y_a	y_b			y_a	y_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}	y'_{20}	y_{21}	y_{22}
CMIX																														
	z_{20}	z'_{21}	z_{22}	z_{23}		z_{13}		x_2				y_a	y_b			y_a	y'_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}	y'_{20}	y_{21}	y_{22}
ROR3																														
	z_{23}		z_{13}		x_2				y_a	y_b			y_a	y'_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}	y'_{20}	y_{21}	y_{22}	z_{20}	z'_{21}	z_{22}
TIX$_{-2}^{\dagger}$																														
	x_3		z_{13}		x_2				y'_a	y_b	z_3		y_a	y'_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}	y'_{20}	y_{21}	y_{22}	z_{20}	z'_{21}	z_{22}

† continued on next page.

Table 17: Evolution of Differential State for one type of Partial Collision (Continued)

	0			3			6			9			12			15			18			21			24			27		29
	x_3	0	z_{13}	0	x_2	0	0	0	y'_a	y_b	z_3		y_a	y'_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}	y'_{20}	y_{21}	y_{22}	z_{20}	z'_{21}	z_{22}
SMIX																														
	y_{30}	y_{31}	y_{32}	y_{33}	x_2	0	0	0	y'_a	y_b	z_3		y_a	y'_b	0	z_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}	y'_{20}	y_{21}	y_{22}	z_{20}	z'_{21}	z_{22}
CMIX																														
	y'_{30}	y_{31}	y_{32}	y_{33}	x_2	0	0	0	y'_a	y_b	z_3		y_a	y'_b	0	z'_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}	y'_{20}	y_{21}	y_{22}	z_{20}	z'_{21}	z_{22}
ROR3																														
	x_2	0	0	0	y'_a	y_b	z_3		y_a	y'_b	0	z'_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}	y'_{20}	y_{21}	y_{22}	z_{20}	z'_{21}	z_{22}	y'_{30}	y_{31}	y_{32}	y_{33}
SMIX																														
	z_{30}	z_{31}	z_{32}	z_{33}	y'_a	y_b	z_3		y_a	y'_b	0	z'_{00}	z_{01}	z_{02}	y_{10}	y_{11}	y_{12}	z_{10}	z_{11}	z'_{12}	y'_{20}	y_{21}	y_{22}	z_{20}	z'_{21}	z_{22}	y'_{30}	y_{31}	y_{32}	y_{33}

Table 18: Evolution of Differential State for weak-Fugue-256 Internal Collision (Backwards)

	0			3			6			9			12			15			18			21			24			27		29
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TIX$_0$																														
	X_0				X_0																									
SMIX																														
	Y_{00}	Y_{01}	Y_{02}	Y_{03}	X_0																									
CMIX																														
	y'_{00}	Y_{01}	Y_{02}	Y_{03}	X_0											X_0														
ROR3																														
	Y_{03}	X_0											X_0															y'_{00}	Y_{01}	Y_{02}
TIX$_{-1}$																														
	x_1	X_0			x_1				Y_{03}				X_0															y'_{00}	Y_{01}	Y_{02}
SMIX																														
	y_{10}	y_{11}	y_{12}	y_{13}	x_1				Y_{03}				X_0															y'_{00}	Y_{01}	Y_{02}
CMIX																														
	y'_{10}	y_{11}	y_{12}	y_{13}	x_1				Y_{03}				X_0			x_1												y'_{00}	Y_{01}	Y_{02}
ROR3																														
	y_{13}	x_1				Y_{03}				X_0			x_1												y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}
TIX$_{-2}$																														
	x_2	x_1			x_2	Y_{03}			y_{13}	X_0			x_1												y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}
SMIX																														
	y_{20}	y_{21}	y_{22}	y_{23}	x_2	Y_{03}			y_{13}	X_0			x_1												y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}
CMIX																														
	y'_{20}	y'_{21}	y_{22}	y_{23}	x_2	Y_{03}			y_{13}	X_0			x_1			x_2	Y_{03}								y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}
ROR3																														
	y_{23}	x_2	Y_{03}			y_{13}	X_0			x_1			x_2	Y_{03}								y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}
TIX$_{-3}$																														
	x_3	x_2	Y_{03}		x_3	y_{13}	X_0		y_{23}	x_1			x_2	Y_{03}								y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}
SMIX																														
	y_{30}	y_{31}	y_{32}	y_{33}	x_3	y_{13}	X_0		y_{23}	x_1			x_2	Y_{03}								y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}
CMIX																														
	y'_{30}	y'_{31}	y'_{32}	y_{33}	x_3	y_{13}	X_0		y_{23}	x_1			x_2	Y_{03}		x_3	y_{13}	X_0				y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}
ROR3																														
	y_{33}	x_3	y_{13}	X_0		y_{23}	x_1			x_2	Y_{03}		x_3	y_{13}	X_0				y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}	y'_{30}	y'_{31}	y'_{32}
TIX$_{-4}$																														

Table 19: Evolution of Differential State for weak-Fugue-256 Internal Collision (Contd.)

	0			3			6			9			12			15			18			21			24			27		29
	y_{33}	x_3	y_{13}	X_0		y_{23}	x_1			x_2	Y_{03}		x_3	y_{13}	X_0				y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}	y'_{30}	y'_{31}	y'_{32}
TIX$_{-4}$	x_4	x_3	y_{13}	X_0	x_4	y_{23}	x_1		y_{33}	x_2	Y_{03}		x_3	y_{13}	X_0				y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}	y'_{30}	y'_{31}	y'_{32}
SMIX	y_{40}	y_{41}	y_{42}	y_{43}	x_4	y_{23}	x_1		y_{33}	x_2	Y_{03}		x_3	y_{13}	X_0				y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}	y'_{30}	y'_{31}	y'_{32}
CMIX	y'_{40}	y'_{41}	y'_{42}	y_{43}	x_4	y_{23}	x_1		y_{33}	x_2	Y_{03}		x_3	y_{13}	X_0	x_4	y_{23}	x_1	y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}	y'_{30}	y'_{31}	y'_{32}
ROR3	y_{43}	x_4	y_{23}	x_1		y_{33}	x_2	Y_{03}		x_3	y_{13}	X_0	x_4	y_{23}	x_1	y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}	y'_{30}	y'_{31}	y'_{32}	y'_{40}	y'_{41}	y'_{42}
TIX$_{-5}$	x_5	x_4	y_{23}	x_1	x_5	y_{33}	x_2	Y_{03}	y_{43}	x_3	y_{13}	X_0	x_4	y_{23}	x_1	y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}	y'_{30}	y'_{31}	y'_{32}	y'_{40}	y'_{41}	y'_{42}
SMIX	y_{50}	y_{51}	y_{52}	y_{53}	x_5	y_{33}	x_2	Y_{03}	y_{43}	x_3	y_{13}	X_0	x_4	y_{23}	x_1	y'_{00}	Y_{01}	Y_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}	y'_{30}	y'_{31}	y'_{32}	y'_{40}	y'_{41}	y'_{42}
CMIX	y'_{50}	y'_{51}	y'_{52}	y_{53}	x_5	y_{33}	x_2	Y_{03}	y_{43}	x_3	y_{13}	X_0	x_4	y_{23}	x_1	y''_{00}	y'_{01}	y'_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}	y'_{30}	y'_{31}	y'_{32}	y'_{40}	y'_{41}	y'_{42}
ROR3	y_{53}	x_5	y_{33}	x_2	Y_{03}	y_{43}	x_3	y_{13}	X_0	x_4	y_{23}	x_1	y''_{00}	y'_{01}	y'_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}	y'_{30}	y'_{31}	y'_{32}	y'_{40}	y'_{41}	y'_{42}	y'_{50}	y'_{51}	y'_{52}
TIX$_{-6}$	x_6	x_5	y_{33}	x_2	y'_{03}	y_{43}	x_3	y_{13}	x'_0	x_4	y_{23}	x_1	y''_{00}	y'_{01}	y'_{02}	y'_{10}	y_{11}	y_{12}	y'_{20}	y'_{21}	y_{22}	y'_{30}	y'_{31}	y'_{32}	y'_{40}	y'_{41}	y'_{42}	y'_{50}	y'_{51}	y'_{52}

Table 20: Evolution of Differential State for Fugue-384 Internal Collision (Backwards)

	0			3			6			9			12			15			18			21			24			27			30...
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TIX$_0$																															
	X_0																X_0														
SMIX																															
	Y_{00}	Y_{01}	Y_{02}	Y_{03}													X_0														
CMIX																															
	Y_{00}	Y_{01}	Y_{02}	Y_{03}													X_0														
ROR3																															
	Y_{03}													X_0																	
SMIX																															
	Z_{00}	Z_{01}	Z_{02}	Z_{03}										X_0																	
CMIX																															
	Z_{00}	Z_{01}	Z_{02}	Z_{03}										X_0																	
ROR3																															
	Z_{03}										X_0																				Y_{00}
SMIX																															
	W_{00}	W_{01}	W_{02}	W_{03}							X_0																				Y_{00}
CMIX																															
	W_{00}	W_{01}	W_{02}	W_{03}							X_0																				Y_{00}
ROR3																															
	W_{03}							X_0																				Y_{00}	Y_{01}	Y_{02}	Z_{00}
TIX$_1$																															
	x_1	Y_{00}			Z_{00}			X_0	W_{03}								x_1											Y_{00}	Y_{01}	Y_{02}	Z_{00}
SMIX																															
	y_{10}	y_{11}	y_{12}	y_{13}	Z_{00}			X_0	W_{03}								x_1											Y_{00}	Y_{01}	Y_{02}	Z_{00}
CMIX																															
	y'_{10}	y_{11}	y_{12}	y_{13}	Z_{00}			X_0	W_{03}								x_1		Z_{00}									Y_{00}	Y_{01}	Y_{02}	Z_{00}
ROR3																															
	y_{13}	Z_{00}			X_0	W_{03}								x_1		Z_{00}									Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}
SMIX																															
	z_{10}	z_{11}	z_{12}	z_{13}	X_0	W_{03}								x_1		Z_{00}									Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}
CMIX																															
	z'_{10}	z'_{11}	z_{12}	z_{13}	X_0	W_{03}								x_1		Z_{00}			X_0	W_{03}					Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}
ROR3																															
	z_{13}	X_0	W_{03}								x_1		Z_{00}			X_0	W_{03}					Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	y'_{10}
SMIX																															
	w_{10}	w_{11}	w_{12}	w_{13}							x_1		Z_{00}			X_0	W_{03}					Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	y'_{10}
CMIX																															
	w_{10}	w_{11}	w_{12}	w_{13}							x_1		Z_{00}			X_0	W_{03}					Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	y'_{10}
ROR3																															
	w_{13}							x_1		Z_{00}			X_0	W_{03}					Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}
TIX$_2$																															

Table 21: Evolution of Differential State for Fugue-384 Internal Collision (Contd.)

	0			3			6			9			12			15			18			21			24			27			30...
	w_{13}							x_1		Z_{00}			X_0	W_{03}					Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}
TIX$_2$	x_2	y'_{10}			z'_{10}			x_1	w_{13}	Z_{00}			X_0	W_{03}			x_2		Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}
SMIX	y_{20}	y_{21}	y_{22}	y_{23}	z'_{10}			x_1	w_{13}	Z_{00}			X_0	W_{03}			x_2		Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}
CMIX	y''_{20}	y_{21}	y_{22}	y_{23}	z'_{10}			x_1	w_{13}	Z_{00}			X_0	W_{03}			x_2		Y''_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}
ROR3	y_{23}	z'_{10}			x_1	w_{13}	Z_{00}			X_0	W_{03}			x_2		Y''_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w_{10}
SMIX	z_{20}	z_{21}	z_{22}	z_{23}	x_1	w_{13}	Z_{00}			X_0	W_{03}			x_2		Y''_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w_{10}
CMIX	z'_{20}	z'_{21}	z'_{22}	z_{23}	x_1	w_{13}	Z_{00}			X_0	W_{03}			x_2		Y''_{00}	Y_{01}	Y_{02}	Z'_{00}	Z'_{01}	Z'_{02}	W_{00}	W_{01}	W_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w_{10}
ROR3	z_{23}	x_1	w_{13}	Z_{00}			X_0	W_{03}			x_2		Y''_{00}	Y_{01}	Y_{02}	Z'_{00}	Z'_{01}	Z'_{02}	W_{00}	W_{01}	W_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w_{10}	w_{11}	w_{12}	y''_{20}
SMIX	w_{20}	w_{21}	w_{22}	w_{23}			X_0	W_{03}			x_2		Y''_{00}	Y_{01}	Y_{02}	Z'_{00}	Z'_{01}	Z'_{02}	W_{00}	W_{01}	W_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w_{10}	w_{11}	w_{12}	y''_{20}
CMIX	w_{20}	w_{21}	w'_{22}	w_{23}			X_0	W_{03}			x_2		Y''_{00}	Y_{01}	Y_{02}	Z'_{00}	Z'_{01}	Z'_{02}	W_{00}	W_{01}	W'_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w_{10}	w_{11}	w_{12}	y''_{20}
ROR3	w_{23}			X_0	W_{03}			x_2		Y''_{00}	Y_{01}	Y_{02}	Z'_{00}	Z'_{01}	Z'_{02}	W_{00}	W_{01}	W'_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w_{10}	w_{11}	w_{12}	y''_{20}	y_{21}	y_{22}	z'_{20}
TIX$_3$	x_3	y''_{20}		X_0	W''_{03}			x_2	w_{23}	Y''_{00}	Y_{01}	Y_{02}	Z'_{00}	Z'_{01}	Z'_{02}	W_{00}	W'_{01}	W'_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w_{10}	w_{11}	w_{12}	y''_{20}	y_{21}	y_{22}	z'_{20}

Table 22: Evolution of Differential State for Fugue-512 Internal Collision (Backwards)

	0			3			6			9			12			15			18			21			24			27			30...
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TIX$_0$	X_0																						X_0								
SMIX	Y_{00}	Y_{01}	Y_{02}	Y_{03}																			X_0								
CMIX	Y_{00}	Y_{01}	Y_{02}	Y_{03}																			X_0								
ROR3	Y_{03}																			X_0											
SMIX	Z_{00}	Z_{01}	Z_{02}	Z_{03}																X_0											
CMIX	Z_{00}	Z_{01}	Z_{02}	Z_{03}																X_0											
ROR3	Z_{03}																X_0														Y_{00}
SMIX	W_{00}	W_{01}	W_{02}	W_{03}													X_0														Y_{00}
CMIX	W_{00}	W_{01}	W_{02}	W_{03}													X_0														Y_{00}
ROR3	W_{03}													X_0														Y_{00}	Y_{01}	Y_{02}	Z_{00}
SMIX	U_{00}	U_{01}	U_{02}	U_{03}										X_0														Y_{00}	Y_{01}	Y_{02}	Z_{00}
CMIX	U_{00}	U_{01}	U_{02}	U_{03}										X_0														Y_{00}	Y_{01}	Y_{02}	Z_{00}
ROR3	U_{03}										X_0														Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}
TIX$_1$	x_1	Y_{00}			Z_{00}			W_{00}	U_{03}		X_0												x_1		Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}
SMIX	y_{10}	y_{11}	y_{12}	y_{13}	Z_{00}			W_{00}	U_{03}		X_0												x_1		Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}
CMIX	y'_{10}	y_{11}	y_{12}	y_{13}	Z_{00}			W_{00}	U_{03}		X_0								Z_{00}				x_1		Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}
ROR3	y_{13}	Z_{00}			W_{00}	U_{03}		X_0								Z_{00}				x_1		Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}
SMIX	z_{10}	z_{11}	z_{12}	z_{13}	W_{00}	U_{03}		X_0								Z_{00}				x_1		Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}
CMIX	z'_{10}	z'_{11}	z_{12}	z_{13}	W_{00}	U_{03}		X_0								Z_{00}			W_{00}	x'_1		Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}
ROR3	z_{13}	W_{00}	U_{03}		X_0								Z_{00}			W_{00}	x'_1		Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}	U_{01}	U_{02}	y'_{10}
SMIX	w_{10}	w_{11}	w_{12}	w_{13}	X_0								Z_{00}			W_{00}	x'_1		Y_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}	U_{01}	U_{02}	y'_{10}
CMIX	w'_{10}	w_{11}	w_{12}	w_{13}	X_0								Z_{00}			W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}	U_{01}	U_{02}	y'_{10}
ROR3	w_{13}	X_0								Z_{00}			W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}	U_{01}	U_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}
SMIX	u_{10}	u_{11}	u_{12}	u_{13}						Z_{00}			W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}	U_{01}	U_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}
CMIX	u_{10}	u_{11}	u_{12}	u_{13}						Z_{00}			W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}	U_{01}	U_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}
ROR3	u_{13}						Z_{00}			W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}	U_{01}	U_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}
TIX$_2$																															

Table 23: Evolution of Differential State for Fugue-512 Internal Collision (Contd.))

	0			3			6			9			12			15			18			21			24			27			30...
	u_{13}						Z_{00}			W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}	U_{01}	U_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}
TIX$_2$	x_2	y'_{10}			z'_{10}		Z_{00}	w'_{10}	u_{13}	W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}	U'_{01}	U_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}
SMIX	y_{20}	y_{21}	y_{22}	y_{23}	z'_{10}		Z_{00}	w'_{10}	u_{13}	W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W_{00}	W_{01}	W_{02}	U_{00}	U'_{01}	U_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}
CMIX	y''_{20}	y_{21}	y'_{22}	y_{23}	z'_{10}		Z_{00}	w'_{10}	u_{13}	W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W''_{00}	W_{01}	W'_{02}	U_{00}	U'_{01}	U_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}
ROR3	y_{23}	z'_{10}		Z_{00}	w'_{10}	u_{13}	W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W''_{00}	W_{01}	W'_{02}	U_{00}	U'_{01}	U_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}	w_{11}	w_{12}	u_{10}
SMIX	z_{20}	z_{21}	z_{22}	z_{23}	w'_{10}	u_{13}	W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W''_{00}	W_{01}	W'_{02}	U_{00}	U'_{01}	U_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}	w_{11}	w_{12}	u_{10}
CMIX	z''_{20}	z'_{21}	z'_{22}	z_{23}	w'_{10}	u_{13}	W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W''_{00}	W_{01}	W'_{02}	U''_{00}	U''_{01}	U'_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}	w_{11}	w_{12}	u_{10}
ROR3	z_{23}	w'_{10}	u_{13}	W_{00}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W''_{00}	W_{01}	W'_{02}	U''_{00}	U''_{01}	U'_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}	w_{11}	w_{12}	u_{10}	u_{11}	u_{12}	y''_{20}
SMIX	w_{20}	w_{21}	w_{22}	w_{23}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W''_{00}	W_{01}	W'_{02}	U''_{00}	U''_{01}	U'_{02}	y'_{10}	y_{11}	y_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}	w_{11}	w_{12}	u_{10}	u_{11}	u_{12}	y''_{20}
CMIX	w''_{20}	w_{21}	w''_{22}	w_{23}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W''_{00}	W_{01}	W'_{02}	U''_{00}	U''_{01}	U'_{02}	y'''_{10}	y_{11}	y''_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}	w_{11}	w_{12}	u_{10}	u_{11}	u_{12}	y''_{20}
ROR3	w_{23}	x'_1		Y'_{00}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W''_{00}	W_{01}	W'_{02}	U''_{00}	U''_{01}	U'_{02}	y'''_{10}	y_{11}	y''_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}	w_{11}	w_{12}	u_{10}	u_{11}	u_{12}	y''_{20}	y_{21}	y'_{22}	z''_{20}
SMIX	u_{20}	u_{21}	u_{22}	u_{23}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W''_{00}	W_{01}	W'_{02}	U''_{00}	U''_{01}	U'_{02}	y'''_{10}	y_{11}	y''_{12}	z'_{10}	z'_{11}	z_{12}	w'_{10}	w_{11}	w_{12}	u_{10}	u_{11}	u_{12}	y''_{20}	y_{21}	y'_{22}	z''_{20}
CMIX	u'_{20}	u'_{21}	u'_{22}	u_{23}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W''_{00}	W_{01}	W'_{02}	U''_{00}	U''_{01}	U'_{02}	y'''_{10}	y_{11}	y''_{12}	z''_{10}	z''_{11}	z'_{12}	w'_{10}	w_{11}	w_{12}	u_{10}	u_{11}	u_{12}	y''_{20}	y_{21}	y'_{22}	z''_{20}
ROR3	u_{23}	Y_{01}	Y_{02}	Z_{00}	Z_{01}	Z_{02}	W''_{00}	W_{01}	W'_{02}	U''_{00}	U''_{01}	U'_{02}	y'''_{10}	y_{11}	y''_{12}	z''_{10}	z''_{11}	z'_{12}	w'_{10}	w_{11}	w_{12}	u_{10}	u_{11}	u_{12}	y''_{20}	y_{21}	y'_{22}	z''_{20}	z'_{21}	z'_{22}	w''_{20}
TIX$_3$	x_3	Y'''_{01}	Y_{02}	Z_{00}	Z'''_{01}	Z_{02}	W''_{00}	W'''_{01}	W''_{02}	U''_{00}	U''_{01}	U'_{02}	y'''_{10}	y_{11}	y''_{12}	z''_{10}	z''_{11}	z'_{12}	w'_{10}	w_{11}	w_{12}	u_{10}	u'_{11}	u_{12}	y''_{20}	y_{21}	y'_{22}	z''_{20}	z'_{21}	z'_{22}	w''_{20}
ROR3																															