# The Hash Function "Fugue 2.0"

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

April 2, 2012

## 1 Specification of Fugue 2.0

Fugue 2.0 is a modification of the original Fugue hash function. It retains the main design components of Fugue, e.g. SMIX, CMIX etc. In fact, it can be seen as a re-parametrization of the parametrized version of original Fugue. Essentially, Fugue2.224 and Fugue2.256 have one SMIX per input word as opposed to two SMIX per input word. Similarly, Fugue2.384 and Fugue2.512 have two and three SMIXes per input word resp. (cf. three and four SMIXes per input word in Fugue-384 and Fugue-512). The final rounds have been additionaly fortified. The reduction in number of rounds is based on analysis which proves that differential collision attacks of the most common kind (and all their known variants) do not work on even these reduced rounds. Such proofs are done in detail in the original paper, and in the full version of this document proofs for the new version Fugue 2.0 will also be given.

## 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 $4 \times 3$ 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 $a_7 a_6 ... a_0$ (or the polynomial $a_7 x^7 + a_6 x^6 + ... + a_0$) by the byte **02** (i.e. the polynomial $x$), we get the polynomial $a_6 x^7 + a_5 x^6 + a_4 x^5 + (a_3 \oplus a_7) x^4 + (a_2 \oplus a_7) x^3 + a_1 x^2 + (a_0 \oplus a_7) x + a_7$, 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 Fugue2.256

The main component in the hash function Fugue 2.0 (called "SMIX" below) is a mapping from 16 bytes to 16 bytes, which resembles the round functions of the block cipher AES [**?**] (this SMIX component is unchangeed from the previous (original) version of Fugue). As in AES, it is sometimes convenient to view the 16 bytes as a $4 \times 4$ 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 **??**.

### 4.2 Super-Mix

Similarly to the AES round function, the SMIX transformation in Fugue takes a $4 \times 4$ 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 $16 \times 16$ 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 $4 \times 4$ matrix (denoted $\mathbf{M}$). Like in AES, the matrix $\mathbf{M}$ will be a *circulant* matrix, but Fugue uses a different matrix than the $4 \times 4$ 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 $4 \times 4$ 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 $4 \times 4$ input matrix $\mathbf{U}$ we compute:

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

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 $4 \times 4$ 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 $16 \times 16$ matrix $\mathbf{N}$, when the $4 \times 4$ 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).

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

Figure 1: The matrix $N^{-1}$

$$\mathbf{N} = \left(\begin{array}{cccc@{\quad}cccc@{\quad}cccc@{\quad}cccc}
1 & 4 & 7 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 4 & 7 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 7 & 1 & 1 & 4 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 4 & 7 & 1 & 1 \\[1ex]
0 & 0 & 0 & 0 & 0 & 4 & 7 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 4 & 7 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 7 & 1 & 0 & 4 \\
4 & 7 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\[1ex]
0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 6 & 4 & 7 & 1 & 7 & 0 & 0 & 0 \\
0 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 1 & 6 & 4 & 7 \\
7 & 1 & 6 & 4 & 0 & 0 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 \\
0 & 0 & 0 & 7 & 4 & 7 & 1 & 6 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\[1ex]
0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 5 & 4 & 7 & 1 \\
1 & 5 & 4 & 7 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 4 & 0 & 0 \\
0 & 0 & 4 & 0 & 7 & 1 & 5 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 \\
0 & 0 & 0 & 4 & 0 & 0 & 0 & 4 & 4 & 7 & 1 & 5 & 0 & 0 & 0 & 0
\end{array}\right) \tag{3}$$

The Super-Mix transform is invertible, as verified by a computer program that yielded the inverse for **N** as described in Figure 1.

## 4.3 The Hash Function F2.256

In this section we specify the hash function **F**2.256 that underlies Fugue2.256. The function **F**2.256 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**2.256 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**2.256.

### 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 $4 \times 30$ matrix (e.g. the first column of the state being $\mathbf{S}_0$, etc.).

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

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

Note that this is half the number of steps in original **F**-256.

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

$$\begin{aligned}
&\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0;\\
&\mathbf{S}_0 = I \text{ (i.e., } \mathbf{S}_0^i = I^i \text{ for } i = 0,1,2,3);\\
&\mathbf{S}_{14} \mathrel{+}= \mathbf{S}_0;\\
&\mathbf{S}_{20} \mathrel{+}= \mathbf{S}_0;\\
&\mathbf{S}_8 \mathrel{+}= \mathbf{S}_1.
\end{aligned}$$

- 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 **TIX** above is slightly different from **TIX** in **F**-256.

- 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 **CMIX** step is same as in **F**-256.

- The step **SMIX**(which remains unchanged from **F**-256), just operates on the first four columns of the state, which can be viewed as a $4 \times 4$ 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 $\mathbf{W}^i_j$ by S-box[$\mathbf{W}^i_j$], 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**2.256 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 26 times
    {
    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.

The final round's first loop has 26 **SMIX**es compared to the 10 **SMIX**es in the original **F**-256. The second loop is unchanged.

### 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 $\text{IV}_0, \ldots, \text{IV}_7$, and for all $j = 0...7$ we set $\mathbf{S}_{22+j} = \text{IV}_j$. (In matrix notations, we set $\mathbf{S}_{0..21} = 0$ and $\mathbf{S}_{22..29} = \text{IV}$.)

### 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

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

then the first 16 bytes of the output would be

$$\mathbf{04\ 05\ 06\ 07\ 08\ 09}\ \dots\ \mathbf{12\ 13}$$

### 4.3.5 Complete Specification of the Hash Function F2.256

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 $\mathrm{IV}_0$, $\mathrm{IV}_1$,...,$\mathrm{IV}_7$), the hash function **F**2.256 operates as follows.

```
$for\ j = 0..21,\ \mathbf{S}_j = 0;$
$for\ j = 0..7,\ \text{Set } \mathbf{S}_{(22+j)} = \mathrm{IV}_j.$
$for\ i = 1..m$
{   $\mathbf{TIX}(P_i);$
    $repeat\ 1\ times\ \{\mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX};\}$
}
$repeat\ 26\ times\ \{\mathbf{ROR3}; \mathbf{CMIX}; \mathbf{SMIX};\}$
$repeat\ 13\ times$
{   $\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0; \mathbf{S}_{15} \mathrel{+}= \mathbf{S}_0; \mathbf{ROR15}; \mathbf{SMIX};$
    $\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0; \mathbf{S}_{16} \mathrel{+}= \mathbf{S}_0; \mathbf{ROR14}; \mathbf{SMIX};$
}
$\mathbf{S}_4 \mathrel{+}= \mathbf{S}_0; \mathbf{S}_{15} \mathrel{+}= \mathbf{S}_0;$

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

## 4.4 The Hash Function Fugue2.256

The hash function Fugue2.256 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**2.256 to the result, along with a fixed initial vector (IV). The fixed IV that is used by Fugue2.256 is defined (in matrix notations) as

$$\text{IV256} = \begin{pmatrix} \mathbf{e0} & \mathbf{c4} & \mathbf{e9} & \mathbf{a4} & \mathbf{a6} & \mathbf{3f} & \mathbf{41} & \mathbf{45} \\ \mathbf{1b} & \mathbf{87} & \mathbf{a9} & \mathbf{6b} & \mathbf{de} & \mathbf{74} & \mathbf{05} & \mathbf{80} \\ \mathbf{63} & \mathbf{07} & \mathbf{8e} & \mathbf{39} & \mathbf{57} & \mathbf{3c} & \mathbf{b3} & \mathbf{a1} \\ \mathbf{da} & \mathbf{e9} & \mathbf{ec} & \mathbf{15} & \mathbf{2c} & \mathbf{be} & \mathbf{17} & \mathbf{c6} \end{pmatrix}$$

That is, the first word in IV256 is $\text{IV256}_0$ = **e0 1b 63 da**, the second word is $\text{IV256}_1$ = **c4 87 07 e9**, and so on upto the last word $\text{IV256}_7$ = **45 80 a1 c6**. This fixed IV was obtained by running **F**2.256 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 Fugue2.256 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**2.256 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 Fugue2.256.

**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**2.256 is applied to $X''$.

[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.

### 4.5 Pseudo-Random Function PR-Fugue2.256

The function PR-Fugue2.256 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 Fugue2.256, 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**2.256 on the resulting encoded stream. The only difference between PR-Fugue2.256 and Fugue2.256 is that PR-Fugue2.256 calls the underlying function **F**2.256 using the 32-byte key as the IV value, instead of the fixed IV value IV256.

### 4.6 Compression Function C-Fugue2.256

We define the compression function C-Fugue2.256 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.

The function C-Fugue2.256 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-Fugue2.256 is just the output of **F**2.256 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

Fugue2.256 can be used as a drop-in replacement for SHA-256 in many other modes of operation, including HMAC [**?**] and randomized hashing [**?**], without resorting to the backward-compatibility mode C-Fugue2.256. For example, HMAC-Fugue2.256 takes an input $X$ and key $K$, and computes

$$\text{HMAC-Fugue2.256}(K, X) \;=\; \text{Fugue2.256}(K \oplus \mathsf{opad} \mid \text{Fugue2.256}(K \oplus \mathsf{ipad} \mid X)\;)$$

We note that when the key is 32-byte long, then PR-Fugue2.256 is a more efficient way of using Fugue to get a pseudo-random function.

## 5 Specification of Parameterized Fugue

The hash function Fugue2.256 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 Fugue2.224, Fugue2.256, Fugue2.384, and Fugue2.512, as required by NIST (as well as a weakened version of Fugue2.256 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 Fugue2.256 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 Fugue2.256 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 Fugue2.256 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 Fugue2.256 we have $r = 5$.

**Final rounds t:** the number of rounds in the second phase of the final transformation **G**. In Fugue2.256 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**2.256, 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 $\mathrm{IV}_0, \mathrm{IV}_1, \ldots, \mathrm{IV}_{n-1}$.

**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 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**2.256, and is independent of the parameters.

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

- **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 $\mathbf{S}$ is done modulo $s$.
- The step $\mathbf{TIX}[s,k](P)$ takes one four-bytes work of input $P$, and is defined as follows:

  $\mathbf{TIX}[36,2](P)$ is

$$\begin{aligned}&\mathbf{S}_7 += \mathbf{S}_0;\\ &\mathbf{S}_0 = P;\\ &\mathbf{S}_{10} += \mathbf{S}_0;\\ &\mathbf{S}_{14} += \mathbf{S}_0;\\ &\mathbf{S}_4 += \mathbf{S}_1\end{aligned}$$

  $\mathbf{TIX}[36,3](P)$ is

$$\begin{aligned}&\mathbf{S}_{10} += \mathbf{S}_0;\\ &\mathbf{S}_0 = P;\\ &\mathbf{S}_7 += \mathbf{S}_0;\\ &\mathbf{S}_{11} += \mathbf{S}_0;\\ &\mathbf{S}_4 += \mathbf{S}_1\\ &\mathbf{S}_{22} += \mathbf{S}_1\end{aligned}$$

- The definition of $\mathbf{CMIX}[s]$ is:

$$\begin{array}{ll}\mathbf{S}_0 += \mathbf{S}_4; & \mathbf{S}_{s/2} \;\; += S_4;\\ \mathbf{S}_1 += \mathbf{S}_5; & \mathbf{S}_{s/2+1} += \mathbf{S}_5;\\ \mathbf{S}_2 += \mathbf{S}_6; & \mathbf{S}_{s/2+2} += \mathbf{S}_6;\end{array}$$

**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:

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

- $\mathbf{G2[n, s, t]}$: 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; }
$\mathbf{S}_4$ += $\mathbf{S}_0$; $\mathbf{S}_p$ += $\mathbf{S}_0$; $\mathbf{S}_{2p}$ += $\mathbf{S}_0$;
```

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

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

**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 $n \le 4$ 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
```

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

### 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 $\mathrm{IV}_0$,$\mathrm{IV}_1$,...,$\mathrm{IV}_{n-1}$, 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**2.224 is $\mathbf{F}[n = 7,\ s = 30, k = 1, r = 15, t = 13]$.
- **F**2.256 is $\mathbf{F}[n = 8,\ s = 30, k = 1, r = 26, t = 13]$.
- **F**2.384 is $\mathbf{F}[n = 12, s = 36, k = 2, r = 14, t = 13]$.
- **F**2.512 is $\mathbf{F}[n = 16, s = 36, k = 3, r = 14, t = 13]$.

## 5.3 IVs

The fixed IV for Fugue2.224 is obtained by running **F**2.224 with an all-zero IV and the one-word input **00 00 00 e0** (representing the decimal number 224).

IV224 = **3a1d28af db9b0b75 66673079 ae45c71c 7efbd0e1 ad70e5be 85430488**

This fixed IV for Fugue2.384 is obtained by running **F**2.384 with an all-zero IV and the one-word input **00 00 01 80** (representing the decimal number 384).

IV384 =
**8e4f1231 837e3d2a ec427f83 925ac741 69fadae5 15398593 657d34f8 667eef64 3d06ff8b 1440123a 2d5101be 9d119f61**

This fixed IV for Fugue2.512 is obtained by running **F**2.512 with an all-zero IV and the one-word input **00 00 02 80** (representing the decimal number 512).

IV512 =
**9010bba7 a7a999bc e479d955 d50e2474 c0d1b8c6 3db445f3 6b00cb8a b1057fc7 a2ef9305 70c632f8 9834386c ac3c9940 b3c8ba4a ecafa8e5 ea32fa93 530a723b**