

WAMM: A CANDIDATE ALGORITHM FOR THE SHA-3 COMPETITION

JOHN WASHBURN

ABSTRACT. The WaMM algorithm is a candidate algorithm submitted to the US National Institute of Standards and Technology (NIST) as part of the SHA-3 Competition. The SHA-3 Competition seeks to find a replacement for the current Secure Hash Algorithm (include hyper-link to FIPS). WaMM is the author's submission to this contest.

The algorithm uses matrix multiplication as the mixing operator. The algebraic structure consists of two binary operations, $\{\oplus, +\}$ which operate on bytes and the integer values they represent. Byte values are integers in the range, $[0 \cdots 255]$. The addition operator, $+$ is addition modulo 256 and forms an Abelian group over byte values. The only property possessed by the multiplication operator, \oplus , is closure on byte values.

The lack of higher algebraic structure provides the algorithm the necessary properties of irreversibility, non-linearity, and collision resistance.

Date: November 1, 2008.

Key words and phrases. Cryptography, Secure Hash Algorithm, Hashing, SHA-3, WaMM, Matrix Multiplication .

CONTENTS

1. Introduction	3
2. Hashing 101	3
3. Definitions	4
3.1. Bit	5
3.2. Byte	5
3.3. Byte values	5
3.4. Vector	5
3.5. Matrix	6
3.6. Addition modulo 256	7
3.7. XOR of Bits	7
3.8. XOR of Bytes	7
3.9. XOR of Vectors	8
3.10. WaMM Multiplication	8
3.11. Squaring with Matrix Multiplication	11
3.12. Addition of Vectors	11
4. Input and Output	12
5. The WaMM Algorithm	14
5.1. Internal State of WaMM Algorithm	15
5.2. Initializing the Internal State	16
5.3. Read Data	16
5.4. Updating the State Matrix	17
5.5. The Count Vector, V_C	17
5.6. Post Processing	17
5.7. Tapping the State Matrix	17
6. Design Considerations	20
7. Alternate Designs	23
8. Pros and Cons of WaMM	24
9. Known Attacks	25
10. Performance	25

1. INTRODUCTION

This hashing algorithm is a proposed replacement for the Secure Hash Algorithm. The working name is name is WaMM, which derives from (Wa)shburn: (M)atrix (M)ultiplication. Matrix multiplication using a non-invertible and non-linear binary operator is the mechanism used to resist collisions and to create a one-way function.

The algorithm limits itself to operations that are particularly fast on current implementations of silicon: Table lookup, Addition modulo 256, Addition modulo 1024, Addition modulo 2^{256} and bit-wise exclusive-OR (XOR).

2. HASHING 101

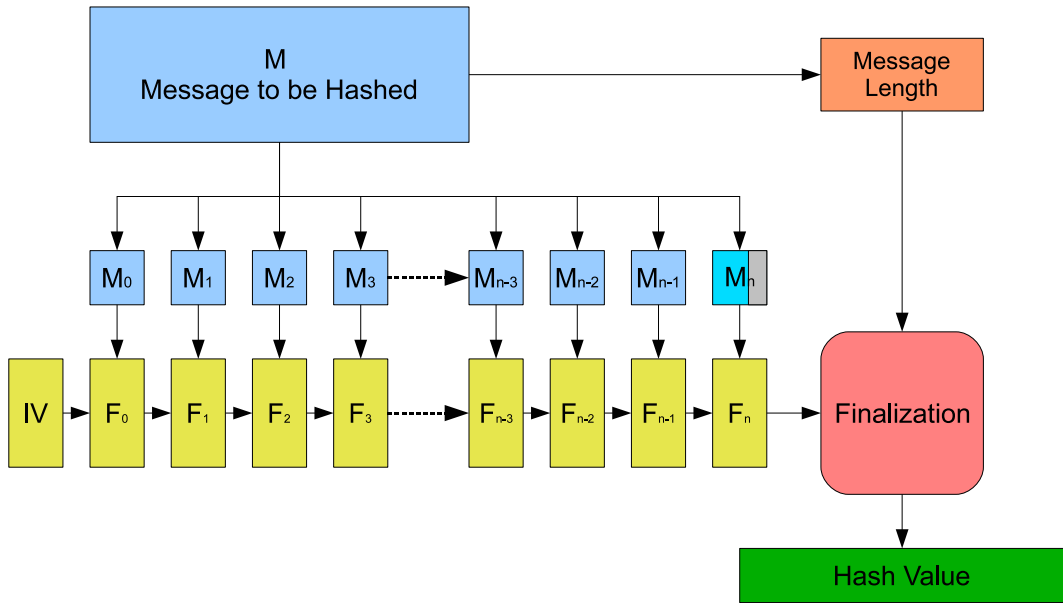


FIGURE 1. Merkle-Damgard Construction

Figure 1 illustrates the Merkle-Damgard construction of hashes. Nearly all modern hashes use this construction in order to process a message of arbitrary length into a fixed-sized hash value.

The message, M , is broken down into a sequence of fixed length blocks,

$$M_0, M_1, M_2, \dots, M_{n-3}, M_{n-2}, M_{n-1}.$$

The number of message blocks, n , is given by $n = \lfloor \frac{\text{Message Length}}{\text{Message Block size}} \rfloor$; where $\lfloor \rfloor$ is the floor function. The floor function of a real number x ,

$\lfloor x \rfloor$, is the largest integer less than or equal to x . In this case n is the largest integer less than or equal to $\frac{\text{Message Length}}{\text{Message Block size}}$. In Figure 1 the complete message and the message blocks are in blue.

It is an usual message where $n = \frac{\text{Message Length}}{\text{Message Block size}}$. For the more common situation where $n \neq \frac{\text{Message Length}}{\text{Message Block size}}$ there will a short message block, M_n , which must be padded in some fashion to the full length of a message block. In Figure 1 the short message block is in cyan and the message padding is in gray.

Each of the message blocks is operated on by a compressor function. The compressor function takes two inputs; the message block and some form of state information. The compressor function creates one output; the state information to transfer to the next iteration of the message processing. The output of the final iteration of the compressor function is used as an input to the post processing and ultimately the hash value generated. In Figure 1 the compressor functions are colored yellow.

Consider two messages which are identical except for the contents of the short message block and the resulting message padding. What if the extra bits of the longer message exactly mimic the padding scheme? If this happens, you have the situation where two different messages result in the same hash value. Because of the necessity of padding, there must be some form of finalization in order to distinguish between the pathological situation where the few extra bits of the longer message mimic the padding. The most common forms of finalization used involve the length of the message.

Once the finalization is complete, the hash value is generated from the state information available.

The Merkle-Damgard construction is so common because it can be proven that if the compressor function (yellow boxes) is collision resistant, then the resulting hash value (green box) is also collision resistant. The WaMM algorithm is also a Merkle-Damgard construction, with an internal state of 8192 bits. The compressor function used in the WaMM hashing algorithm is more similar to the round function of a block cipher than it is to the compressor functions of other Merkle-Damgard hashes (e.g. SHA-1, SHA-2, MD5, etc.). The internal state of the algorithm which passed from one iteration of the compressor function to the next is a 32x32 matrix, SM, called the State Matrix. Both the message padding and the processing of the message length are slightly different than other Merkle-Damgard constructions as well.

3. DEFINITIONS

The data structures used by the WaMM algorithm are:

- Bits,
- Bytes,
- Byte values,
- Vectors, and
- Matrices

The operations used by the WaMM algorithm are:

- Addition modulo 256,
- XOR bits,
- XOR of bytes,
- XOR of vectors,
- WaMM multiplication,
- Squaring with matrix multiplication,
- Addition of Vectors (addition modulo 2^{256})

3.1. **Bit.** Bit retains its common meaning as a single unit of binary information. A bit is an integer and a member of the set: $\{0, 1\}$.

3.2. **Byte.** A byte retains its common meaning as a collection of eight bits. A byte consists of eight bits arranged from the most significant bit to the least significant bit. As per the requirements of the NIST, the eight bits of byte, B , are designated

$$B = \{b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7\}$$

where b_0 is the most significant bit of the byte and b_7 is the least significant bit of the byte.

A byte also represents an integer value, N , in the range: $[0 \dots 255]$. The conversion of a byte from a collection of ordered bits to an integer is given by the formula:

$$(1) \quad N_B = \sum_{k=0}^7 b_k 2^{7-k}$$

3.3. **Byte values.** Byte values are integers in the range $[0 \dots 255]$.

3.4. **Vector.** A vector consists of 32 bytes arranged from the least significant byte to the most significant byte. The 32 bytes of vector, V , are designated:

$$V = \{B_0, B_1, B_2, \dots, B_{29}, B_{30}, B_{31}\}$$

The byte, B_0 , is the first and most significant byte of the buffer. The byte, B_{31} , is the last and least significant byte of the buffer.

A vector also represents an integer value, N , in the range: $[0 \dots 2^{256} - 1]$. The conversion of a vector from a collection of ordered bytes to an integer is given by the formula:

$$(2) \quad N_V = \sum_{k=0}^{31} N_{B_k} 2^{8(31-k)}$$

where N_{B_k} is the integer value of byte B_k as defined by (1).

A vector can be represented as string of 64 hexadecimal digits; two digits per byte. In order to improve readability, the string of hexadecimal digits is written in one of three forms; 32 sub-strings of two digits each, or 16 sub-strings of 4 digits each, or as 8 sub-strings of 8 digits each. The sub-strings are separated by a space and the prefix is only for the whole string of 64 digits. The first hexadecimal digit represents the four (4) most significant bits, $\{b_0, b_1, b_2, b_3\}$. The second hexadecimal digit represents the four (4) least significant bits, $\{b_4, b_5, b_6, b_7\}$. Below is the a vector represented as a string of hexadecimal digits in a variety of ways.

```
0x48 AF 34 34 98 B2 A9 23 E6 23 F2 73 8B E2 AA 72 77 92 73 2E 13 23 11 DD 1C EE 17 01 00 99 3B 23
0x48AF 3434 98B2 A923 E623 F273 8BE2 AA72 7792 732E 1323 11DD 1CEE 1701 0099 3B23
0x48AF3434 98B2A923 E623F273 8BE2AA72 7792732E 132311DD 1CEE1701 00993B23
0x48AF343498B2A923E623F2738BE2AA727792732E132311DD1CEE170100993B23
```

The most significant byte of the vector, B_0 , is 0x48. The least significant byte of the vector, B_{31} , is 0x23. The 16 most significant bits of the vector,

$[b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10}, b_{11}, b_{12}, b_{13}, b_{14}, b_{15}]$,

are:

$[0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1]$.

The 16 least significant bits of the vector,

$[b_{241}, b_{242}, b_{243}, b_{244}, b_{245}, b_{246}, b_{247}, b_{248}, b_{248}, b_{249}, b_{250}, b_{251}, b_{252}, b_{253}, b_{254}, b_{255}]$,

are:

$[0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1]$.

3.5. Matrix. A matrix is a collection of bytes arranged as 32 rows, each with 32 bytes. The 1024 bytes of matrix, M , are designated:

$$M = \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & \cdots & B_{0,29} & B_{0,30} & B_{0,31} \\ B_{1,0} & B_{1,1} & B_{1,2} & \cdots & B_{1,29} & B_{1,30} & B_{1,31} \\ B_{2,0} & B_{2,1} & B_{2,2} & \cdots & B_{2,29} & B_{2,30} & B_{2,31} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ B_{29,0} & B_{29,1} & B_{29,2} & \cdots & B_{29,29} & B_{29,30} & B_{29,31} \\ B_{30,0} & B_{30,1} & B_{30,2} & \cdots & B_{30,29} & B_{30,30} & B_{30,31} \\ B_{31,0} & B_{31,1} & B_{31,2} & \cdots & B_{31,29} & B_{31,30} & B_{31,31} \end{bmatrix}$$

The byte, $B_{j,k}$, is byte found in column k of row j of the matrix.

A matrix is also a collection of 32 vectors arranged as rows of the matrix. The 32 vectors of matrix, M , are designated:

$$M = \begin{bmatrix} V_0 \\ V_1 \\ V_2 \\ \vdots \\ V_{29} \\ V_{30} \\ V_{31} \end{bmatrix}$$

3.6. Addition modulo 256. Addition modulo 256, designated by the symbol $+$, is a binary operation between two bytes. Given two bytes, B_j and B_k , the result, $B_m = B_j + B_k$, is defined as the byte, B_m , such that

$$(3) \quad N_{B_m} = (N_{B_j} + N_{B_k}) \bmod 256$$

where $0 \leq N_{B_m} < 256$ and N_{B_m} , N_{B_j} , and N_{B_k} are the integer values given by (1). Addition modulo 256 retains its customary meaning when two bytes are added together and the carry is discarded.

3.7. XOR of Bits. XOR, designated by the symbol \oplus , is a binary operation between two bits. XOR has the following truth table.

$A \oplus B$		A	
		0	1
B	0	0	1
	1	1	0

3.8. XOR of Bytes. Bitwise XOR of two bytes is also designated by the symbol \oplus . Given two bytes, X and Y , the bits of the 2 bytes are designated as:

$$X = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$$

$$Y = \{y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7\}$$

The bits of the result, Z , of the operation $Z = X \oplus Y$ are designated as

$$Z = \{z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7\}$$

and the bits of $Z = X \oplus Y$ are defined as follows:

$$Z_k = X_k \oplus Y_k$$

where \oplus is the XOR operation described in 3.7. Bit-wise XOR of two bytes retains its customary meaning.

3.9. XOR of Vectors. Bitwise XOR of two vectors is also designated by the symbol \oplus . Given two vectors, X and Y , the bytes of the 2 vectors are designated as:

$$X = [X_0, X_1, X_2, \dots, X_{29}, X_{30}, X_{31}]$$

$$Y = [Y_0, Y_1, Y_2, \dots, Y_{29}, Y_{30}, Y_{31}]$$

The bytes of the result, Z , of the operation $Z = X \oplus Y$ are designated as

$$Z = [Z_0, Z_1, Z_2, \dots, Z_{29}, Z_{30}, Z_{31}]$$

and the bytes of $Z = X \oplus Y$ are defined as follows:

$$Z_k = X_k \oplus Y_k$$

where \oplus is the XOR operation described in 3.8. Bit-wise XOR of the 32 bytes of two vectors retains its customary meaning.

3.10. WaMM Multiplication. WaMM Multiplication, designated by \otimes , is a binary operator operating on two bytes. The operator, \otimes , is closed. The operator, \otimes , is not associative. In general $((x \otimes y) \otimes z \neq x \otimes (y \otimes z))$. The operator, \otimes , is not commutative. In general $(x \otimes y \neq y \otimes x)$. The operator, \otimes , has no identity element and, thus, no inverse.

The operation is defined by a truth table containing 65,536 byte values arranged as a matrix of 256 rows, each with 256 byte values. The truth table for the \otimes operator is contained in Appendix A. The left hand operand determines the row of the truth table to use for the current operation. The right hand operand determines the column of the truth table to use for the current operation.

Appendix A contains the truth table. The table is arranged as 256 separate printed grids of byte values, one grid for each row of the truth table.

Appendix A contains the truth table. The true table is as 256 separate printed grids of byte values, one grid for each row of the truth table. A given row is identified by the label (e.g. $0xA4 \otimes B$) in the upper left corner cell of its grid. Each grid contains 16 rows and 16 columns of byte values. The 256 values in a single row of the truth table are arranged in the corresponding grid as 16 rows and 16 columns of byte values. The four most significant bits of the right hand operand determine which row of the grid to use for the current operation. The four least significant bits of the right hand operand determine which column of the grid to use for the current operation.

Demonstrating that the operator is not associative will illustrate the arrangement of the truth table values in the grids printed found in Appendix A.

$$\begin{array}{l|l}
 (0xA6 \otimes 0x07) \otimes 0x83 = & 0xA6 \otimes (0x07 \otimes 0x83) = \\
 0x71 \otimes 0x83 = 0x42 & 0xA6 \otimes 0x01 = 0xA5
 \end{array}$$

$0xA6 \otimes B$		Four Least Significant Bits of B									
		0x00	0x01	...	0x06	0x07	0x08	0x09	...	0x0E	0x0F
Bits of B	0x00	0xDC	0xA5	...	0xDE	0x71	0xFD	0x87	...	0xC7	0xC3
	0x10	0x16	0x05	...	0x43	0x09	0xF9	0x06	...	0x51	0xFC
	0x20	0xFB	0x8A	...	0x49	0x98	0x42	0x66	...	0x80	0xB1
	0x30	0x20	0x10	...	0xCA	0xAA	0x67	0xBC	...	0x5D	0xB3
	0x40	0xF0	0xD4	...	0x7F	0x02	0x69	0x01	...	0x2D	0x45
	0x50	0xEC	0x8E	...	0xBD	0x1D	0xB4	0xAF	...	0xFA	0xA9
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

TABLE 1. $(0xA6 \otimes 0x07)$

$0x07 \otimes B$		Four Least Significant Bits of B									
		0x00	0x01	0x02	0x03	0x04	...	0x0C	0x0D	0x0E	0x0F
Bits of B	0x00	0xA9	0x72	0xD4	0x8B	0x60	...	0x6E	0x5D	0x94	0x90
	0x10	0xE3	0xD2	0xA8	0x59	0x91	...	0xB5	0x11	0x1E	0xC9
	0x20	0xC8	0x57	0xF1	0x93	0x3C	...	0x7A	0x0E	0x4D	0x7E
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	0x70	0x43	0x0D	0x2E	0xC2	0xD7	...	0xD8	0x96	0x58	0xA7
	0x80	0x13	0xD0	0x66	0x01	0xA5	...	0x4A	0x15	0x82	0x2D
	0x90	0x6D	0x44	0xF0	0x27	0x9C	...	0xF8	0x06	0x35	0x75
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	

TABLE 2. $(0x07 \otimes 0x83)$

The value of $(0xA6 \otimes 0x07)$ was determined by finding the grid in Appendix A labeled: $0xA6 \otimes B$. The 4 most significant bits of $0x07$ are $0x00$. This value determines the row of the grid to use for the operation. The 4 least significant bits of $0x07$ are $0x07$. This value determines the column of the grid to use for the operation. The byte value in this cell of the printed grid is $0x71$.

The value of $(0x07 \otimes 0x83)$ was determined by finding the table in Appendix A labeled: $0x07 \otimes B$. The 4 most significant bits of $0x83$ are $0x80$. This value determines the row of the grid to use for the operation. The 4 least significant bits of $0x83$ are $0x03$. This value

$0x71 \otimes B$		Four Least Significant Bits of B									
		0x00	0x01	0x02	0x03	0x04	...	0x0C	0x0D	0x0E	0x0F
Bits of B	0x00	0xEA	0xB3	0x15	0xCC	0xA1	...	0xAF	0x9E	0xD5	0xD1
	0x10	0x24	0x13	0xE9	0x9A	0xD2	...	0xF6	0x52	0x5F	0x0A
	0x20	0x09	0x98	0x32	0xD4	0x7D	...	0xBB	0x4F	0x8E	0xBF
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	0x70	0x84	0x4E	0x6F	0x03	0x18	...	0x19	0xD7	0x99	0xE8
	0x80	0x54	0x11	0xA7	0x42	0xE6	...	0x8B	0x56	0xC3	0x6E
	0x90	0xAE	0x85	0x31	0x68	0xDD	...	0x39	0x47	0x76	0xB6
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

TABLE 3. $(0x71 \otimes 0x83)$

$0xA6 \otimes B$		Four Least Significant Bits of B									
		0x00	0x01	...	0x06	0x07	0x08	0x09	...	0x0E	0x0F
Bits of B	0x00	0xDC	0xA5	...	0xDE	0x71	0xFD	0x87	...	0xC7	0xC3
	0x10	0x16	0x05	...	0x43	0x09	0xF9	0x06	...	0x51	0xFC
	0x20	0xFB	0x8A	...	0x49	0x98	0x42	0x66	...	0x80	0xB1
	0x30	0x20	0x10	...	0xCA	0xAA	0x67	0xBC	...	0x5D	0xB3
	0x40	0xF0	0xD4	...	0x7F	0x02	0x69	0x01	...	0x2D	0x45
	0x50	0xEC	0x8E	...	0xBD	0x1D	0xB4	0xAF	...	0xFA	0xA9
	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

TABLE 4. $(0xA6 \otimes 0x01)$

determines the column of the grid to use for the operation. The byte value in this cell of the printed grid is 0x01.

The same process is used to look up the values of $(0x71 \otimes 0x83)$ and $(0xA6 \otimes 0x01)$.

Using a truth table as the definition of a binary operator is cumbersome when laid out on paper as is done in Appendix A. The practical matter is the truth table would be implemented in 64 kilobytes of memory as a lookup table arranged as a two dimensional array, T . The lookup into the truth table, T , for the result of the operation $Z = (X \otimes Y)$ is given by: $Z = T[X][Y]$.

The whole truth table would conveniently fit on a single chip such as the M27C512-12F1. The 8 bits of the right operand and the 8 bits of the left operand are used to form the signals to the 16 address lines of the memory chip. Depending on the exact chip set used, the result of the operation is found 70-200 nanoseconds later on the 8 output lines

of the memory chip. If this algorithm is selected as the standard, then the truth table could be a mass-produced IC with pin outs and data sheets similar to a 64K x 8 (512 Kbit) memory chip currently on the market.

3.11. Squaring with Matrix Multiplication. Squaring with Matrix Multiplication is a unitary operator operating on a matrix to produce a second matrix. Given a matrix, X , the resultant matrix, Y is

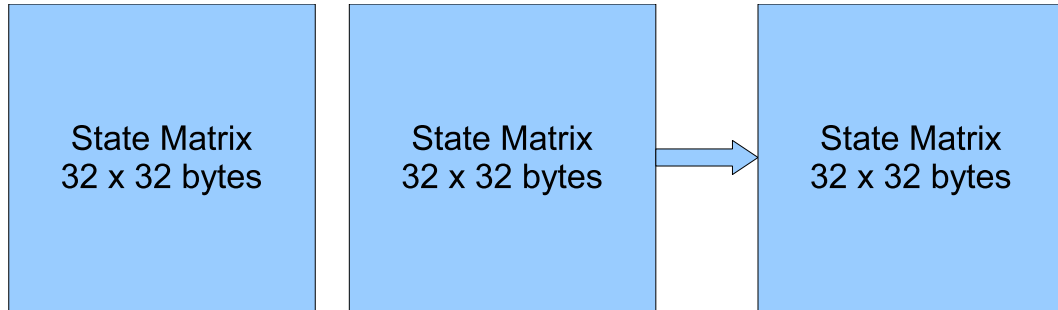


FIGURE 2. Squaring with Multiplication

given by:

$$\left[\begin{array}{c} X \\ \end{array} \right] \left[\begin{array}{c} X \\ \end{array} \right] = \left[\begin{array}{c} Y \\ \end{array} \right]$$

where:

$$(4) \quad Y_{ij} = \sum_{k=0}^{31} X_{ik} \otimes X_{kj}$$

The summation is addition modulo 256 and the multiplication is the \otimes operator described in 3.10.

3.12. Addition of Vectors. Addition of 2 vectors is designated by the symbol $+$ and is a binary operation between two vectors. Using (2) it is possible to consider a vector as a 256-bit integer. As integer values, two vectors can then be added modulo 2^{256} . Given two vectors, X and Y , the result $Z = X + Y$, is defined as the vector, Z , such that

$$(5) \quad N_Z = (N_X + N_Y) \text{ mod } 2^{256}$$

where $0 \leq N_Z < 2^{256}$ and N_X , N_Y , and N_Z are the integer values of the vectors: X , Y , and Z as given by (2).

4. INPUT AND OUTPUT

A message to be hashed comes in two basic forms; bit streams and byte streams. Bit streams are message that have a natural frame of a single bit. Bit streams are message that admit a frame of a byte. Along with the statement, "*The next 256-bit block of the message is read.*", there must be a precise definition of how bits from the message are transferred into the internal state of the hash algorithm.

For messages with bit framing, the bits are sequenced in the following manner:

$$b_0, b_1, b_2, b_3, \dots, b_{n-3}, b_{n-2}, b_{n-1}$$

where n is the length of the message in bits and

Bit b_0 is the first bit of the message. Bit b_1 is the second bit of the message. Bit b_2 is the third bit of the message. Bit b_{n-1} is the last bit of the message.

For messages with simple byte framing, the bytes are sequenced in the following manner:

$$B_0, B_1, B_2, B_3, \dots, B_{n-2}, B_{n-1}$$

where B_0 is the first byte of the message. Byte B_1 is the second byte of the message. Byte B_2 is the third byte of the message. Byte B_{n-1} is byte n of the message. Byte B_n is byte $n + 1$ of the message. where n is the length of the message in bytes. Byte b_0 is the first byte of the message. Byte b_1 is the second byte of the message. Byte b_2 is the third byte of the message. Byte b_{n-1} is the last byte of the message. Within each byte there is a least significant bit and a most significant bit. As per the NIST requirements, the most significant bit of a byte is b_0 and the least significant bit is b_7 .

Messages can also have word framing where the number of bits in the word more than 8. If so, the message admits byte framing. For messages with a word framing, the words of the message are sequenced in the following manner:

$$W_0, W_1, W_2, W_3, \dots, W_{n-2}, W_{n-1}$$

where W_0 is the first word of the message. Word W_1 is the second word of the message. Word W_2 is the third word of the message. Word W_{n-1} is the last word of the message. Within each word there is a least significant bit and a most significant bit. As per the NIST requirements the most significant bit of a word is w_0 and the least significant bit is w_{r-1} for a word length of r bits.

Vectors are the data structure used to accept bits from the message. The 256 bits of a vector, $v_0, v_1, v_2, \dots, v_{253}, v_{254}, v_{255}$, are arranged as 32

bytes, $B_0 - B_{31}$. The bit n of the vector, v_n , is located in bit k of byte j such that:

$$(6a) \quad n = 8j + k$$

$$(6b) \quad j = \left\lfloor \frac{n}{8} \right\rfloor$$

$$(6c) \quad k = n \bmod 8$$

where $\left\lfloor \frac{n}{8} \right\rfloor$ is the largest integer less than or equal to $\frac{n}{8}$; i.e. $\left\lfloor \frac{n}{8} \right\rfloor \leq \frac{n}{8} < \left\lfloor \frac{n}{8} \right\rfloor + 1$.

For bit framed messages, the WaMM algorithm processes the message in blocks of 256-bits. If the message length is not a multiple of 256 bits, then message is implicitly padded with zeros. This implicit padding will be discussed in more detail in 5. Bit n of the message block is transferred to bit n of the vector using (6).

For byte framed messages, the WaMM algorithm processes the message in blocks of 32 bytes. If the message length is not a multiple of 32 bytes, then message is implicitly padded with zeros. This implicit padding will be discussed in more detail in 5. Byte n of the message block is transferred to byte n of the vector.

For word framed messages, the WaMM algorithm processes the message in blocks of T words where T is a function of the word size. If the word size evenly divides 256, then the mapping from words to vectors is simple. But, if the word size does not evenly divide 256, the bits, bytes, and words of the message can still be transferred to a set of vectors in a precise manner. This is because, regardless of word size, there exists within each word a least significant bit and a most significant bit.

Assume the word size is R bits. Find, C , the least common multiple of R and 256. There will be $T = \frac{C}{R}$ words which map to $N = \frac{C}{256}$ vectors. Bit k of word t maps to bit i of byte j in vector n vector by the mapping formula:

$$(7a) \quad Rt + k = 256n + 8j + i$$

$$(7b) \quad n = \left\lfloor \frac{(Rt + k)}{256} \right\rfloor$$

$$(7c) \quad j = \left\lfloor \frac{(Rt + k) - 256n}{8} \right\rfloor$$

$$(7d) \quad i = (Rt + k) \bmod 8$$

where $0 \leq m < T$ and $0 \leq n < N$.

An example will clarify this. For a word size of 48 bits the least common multiple of 48 and 256 is 768. Thus, the bits of 16, words map to the bits of 3 vectors using (7). From (7), m is in the range

$0 \leq m \leq 15$ and n is in the range $0 \leq n \leq 2$. The WaMM algorithm will process the message in blocks of 16-words each. The 3 vectors created from the 16 words will be processed by the WaMM algorithm in this order. Vector V_0 is processed first. Vector V_1 is processed second. Vector V_2 is processed third. Then the next block (16 words) of the message is processed.

If the message length is not a multiple of T words, then the message is implicitly padded with zeros. This implicit padding will be discussed in more detail in 5. If the number of words remaining in the message is less than T (a short block), then the message will be padded out to the next multiple of 256. In the above example, if the bit length of the message was 1024, the 16 words of the message are mapped to three (3) vectors. The remaining 256 bits are mapped to a single vector and processed.

5. THE WAMM ALGORITHM

The WaMM Hash Algorithm as pseudo-code:

Decide on, N , the size of the hash value in bytes; where $24 \leq N \leq 128$ is the size of the hash value in bytes and N is a multiple of 4.

Initialize the internal state of the WaMM Algorithm (Section 5.2)

Break the message, M , into a series of 256-bit blocks, with a possible short block. (Section 5.3)

For Each message block

Transfer the 256-bit block to the Input Vector, V_{in}

Update the State Matrix, SM with the Input Vector, V_{in}

End For Each

If the Message has a remainder (short message block) then

Transfer the short block to the Input Vector, V_{in}

Update the State Matrix, SM with the Input Vector, V_{in}

end if

Set the Count Vector, V_C , to the message length in bits

Transfer the Count Vector, V_C , to the Input Vector, V_{in} (Section 5.5)

Update the State Matrix, SM with the Input Vector, V_{in}

Determine the prime numbers, P_t and P_s , used for the tapping sequence. (Section 5.7)

Determine the number of bytes, N , in the hash value, H .

For $n = 0$ to N do

$a_n = (P_t n + P_s) \bmod 1024$

$j = \lfloor \frac{a_n}{32} \rfloor$

$k = a_n \bmod 32$

$H_n = SM[j][k]$

end for

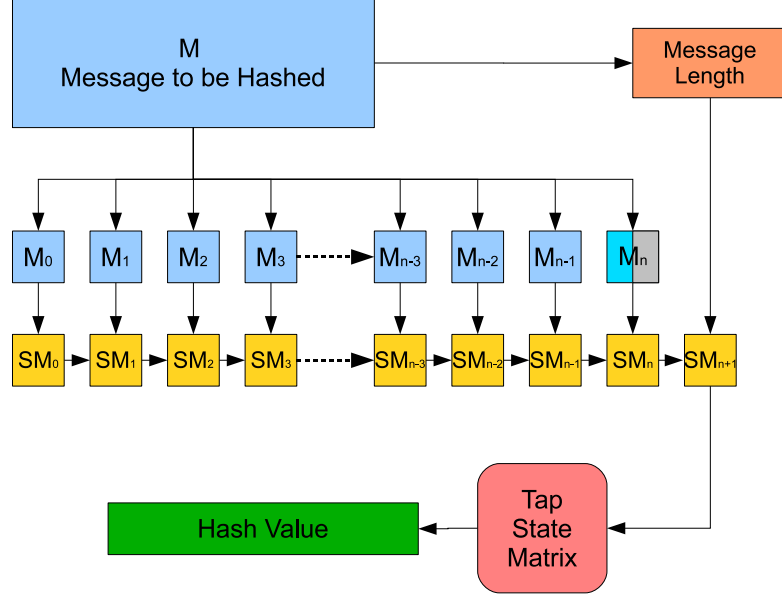


FIGURE 3. Internal State of the WaMM Algorithm

5.1. Internal State of WaMM Algorithm. The internal state of the WaMM algorithm consists of:

- A matrix, SM , called the State Matrix.
- A vector, V_{in} , called the Input Vector.
- A vector, V_C , called the Count Vector. The count vector contains the number of bits in the message.
- The lookup table, T , which contains the truth table for \otimes . The lookup table, T , is arrange such that $T[X][Y] = X \otimes Y$.
- A table of 1 and the first 256 primes,

$\{1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \dots, 97, 101, 103, 107, 109, 113, 127\}$.

designated in the customary fashion as:

$\{P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, \dots, P_{26}, P_{27}, P_{28}, P_{29}, P_{30}, P_{31}, P_{32}\}$.

The odd primes are used to tap the State Matrix, SM, for the hash value, H .

- A buffer, H , for the resultant hash value. The WaMM Hash Algorithm produces a hash value with at least 128 bits and no more than 1024 bits. The buffer, H , is between therefore between 16 and 128 bytes in length.

The internal state of the WaMM algorithm occupies $65 \frac{1}{8}$ kilobytes. If the look up table is excluded, the internal state is 1056 bytes. The bulk of this memory footprint is the 64K needed for the storage of the truth table for the operator, \otimes , in the lookup table, T . The State Matrix, SM, occupies 1 kilobyte.

5.2. Initializing the Internal State. The look up table, T , containing the truth table for \otimes is assumed to be available from message to message and, thus, does not need any special initialization prior to hashing a message. The table of primes is assumed to be available from message to message and, thus, does not need any special initialization prior to hashing a message.

Before a message is hashed, the internal state of the WaMM Hash Algorithm must be initialized.

- (1) The Input Vector, V_{in} , is initialized zero. (Every bit is set to zero.)
- (2) The Count Vector, V_C , is initialized zero. (Every bit is set to zero.)
- (3) The State Matrix, SM, is initialized such that $SM[j][k] = j \otimes k$

5.3. Read Data. The WaMM Hashing Algorithm processes messages of indefinite length by processed the message in 256-bit blocks. A 256-bit block is read into the Input Vector, V_{in} . How the bits (bytes or words) of the incoming message are mapped to the Input Vector V_{in} is described in section 4. Transferring a block of 256-bits from the message to the Input Vector, V_{in} , consists of three steps:

- (1) Set the input vector, V_{in} , to zero. Every bit of V_{in} is set to zero.
- (2) Map the bits of the 256-bit block of the message are mapped to the Input Vector, V_{in} , as described in section 4.
- (3) If the block of the message is less than 256 bits, then the bits of the short block are mapped to the Input Vector, V_{in} in the same manner for a 256-bit block. Because the Input Vector V_{in} is initialized to zero, the short block is zero padded to fill out the 256 bit block.

5.4. Updating the State Matrix. The State Matrix, SM, is updated by XOR'ing the the current row with the Input Vector, V_{in} . The current row of the State Matrix, SM, is incremented

If the current row of the State Matrix, SM, is wraps back to zero, then the State Matrix, SM, is squared twice.

5.5. The Count Vector, V_C . The Count Vector, V_C , contains the bit length of the message modulo 2^{256} . The bit length is the length of the message without padding. While the Count Vector, V_C , contains the message length modulo 2^{256} , message lengths greater than 2^{256} bits are unlikely. Therefore, it should not cause any confusion to say the Count Vector, V_C , contains the message length in bits of the unpadded message. The value in the Count Vector, V_C , can be maintained and updated in any manner provided that when the last block of the message is processed, the Count Vector, V_C , accurately contains the bit length of the message processed.

5.6. Post Processing. If the the message is not an even multiple of 256 bits, the remaining bits of the message are read into the Input Vector, V_{in} . The State Matrix, SM is updated with the Input Vector, V_{in} and possibly squared twice.

Once every bit of the message has been processed, the length of the message is processed in the same manner as a 256 bit message block. The Count Vector, V_C , contains the number of bits in the unpadded message.

- The Count Vector, V_C , is transferred to the input vector, V_{in} .
- The State Matrix, SM, is updated, and possibly squared twice.
- If the updating of the State Matrix, SM, did not cause the State Matrix to be squared twice, then the State Matrix, SM, is squared twice.

5.7. Tapping the State Matrix. With the post processing of the message length completed, the State Matrix, SM, can be tapped to produce the hash value. The WaMM Hash Algorithm can produce a hash value, H , of any bit length where the bit length of the hash value is a multiple of 32 and the bit length of the hash value is in the range $192 \leq L \leq 2048$. Given a bit length of L , the hash value, H , will contain N bytes; where $N = \frac{L}{8}$. The N bytes of the hash value, H , are an ordered sequence of bytes selected from among the 1024 bytes available in the State Matrix, SM. The formula for mapping the N

bytes of the State Matrix, SM to the hash value, H , is given by:

$$\begin{aligned}
 (8a) \quad & H_n = \text{SM}[j][k] \\
 (8b) \quad & s = \frac{L}{32} = \frac{\text{Hash Length}}{\text{HashLengthModulus}} \\
 (8c) \quad & t = s - 2 \\
 (8d) \quad & r = (nP_t + P_s) \bmod (N^2 = 32^2 = 1024) \\
 (8e) \quad & j = \left\lfloor \frac{r}{32} \right\rfloor \\
 (8f) \quad & k = r \bmod 32
 \end{aligned}$$

where P_t and P_s are prime numbers.

For example for a hash length of 192 bits, $L = 192$. The number of bytes, N , in the hash value, H , is $N = \frac{192}{8} = 24$. The values of s and t are 6 ($s = \frac{192}{32} = 6$) and 4 ($t = s - 2 = 4$); respectively. The two prime numbers of the arithmetic sequence, $r = (nP_t + P_s) \bmod 1024$, are $P_6 = 13$ and $P_4 = 7$. The resulting arithmetic sequence, $r = (7n + 13) \bmod 1024$ is :

$$(9) \quad r_n = \{13, 20, 27, 34, 41, 48, 55, 62, 69, 76, 83, 90, 97, 104, 111, 118, 125, 132, 139, 146\}$$

For a hash value of 192 bits, the State Matrix, SM, is tapped as follows:

$$\begin{aligned}
 H_0 &= \text{SM}[0][13] \\
 H_1 &= \text{SM}[0][20] \\
 H_2 &= \text{SM}[0][27] \\
 H_3 &= \text{SM}[1][2] \\
 H_4 &= \text{SM}[1][9] \\
 H_5 &= \text{SM}[1][16] \\
 H_6 &= \text{SM}[1][23] \\
 H_7 &= \text{SM}[1][30] \\
 H_8 &= \text{SM}[2][5] \\
 H_9 &= \text{SM}[2][12] \\
 H_{10} &= \text{SM}[2][19] \\
 H_{11} &= \text{SM}[2][26] \\
 H_{12} &= \text{SM}[3][1] \\
 H_{13} &= \text{SM}[3][8] \\
 H_{14} &= \text{SM}[3][15] \\
 H_{15} &= \text{SM}[3][22] \\
 H_{16} &= \text{SM}[3][29] \\
 H_{17} &= \text{SM}[4][4] \\
 H_{18} &= \text{SM}[4][11] \\
 H_{19} &= \text{SM}[4][18] \\
 H_{20} &= \text{SM}[4][25] \\
 H_{21} &= \text{SM}[5][0] \\
 H_{22} &= \text{SM}[5][7] \\
 H_{23} &= \text{SM}[5][14]
 \end{aligned}$$

The most significant bits of the Hash value, H , are in first byte, H_0 . The least significant bits of the Hash value, H , are in last byte, H_{N-1} .

The use of an arithmetic sequence to tap the State Matrix, SM, is to insure the hash values for various bit lengths (WaMM-128, WaMM-160, WaMM-192, \dots , WaMM-960, WaMM-992, WaMM-1024) are as unrelated to each other as possible.

6. DESIGN CONSIDERATIONS

The first design consideration was mixing operator and how much state information should be retained from message block to message block. The author decided on matrix multiplication for the following reasons:

- (1) Matrix multiplication is an efficient, but expensive mixing operation because each bit in the column or row affects every element of the result.
- (2) The operation of multiplying a square matrix with an arbitrary multiplication operator is intrinsically hard to invert.

The size of the state information was selected so that there would be sufficient state material available to tap in order that hash values of one size would not leak information about hash values of a different size. The danger posed by this scheme is that a great deal of information internal state of the WaMM algorithm is leaked if the same message is hashed, but hash values of several sizes are generated from the same internal state. Using the arithmetic sequence having prime numbers as the difference and initial value insures the 1024 bytes of the internal state are tapped in distinct ways without any common byte sequences. There may be some bytes in common between hash values of different sizes, but no two, consecutive bytes are common between two hash values. generated from the same internal state.

There is no reason why the state matrix, SM, could not be extended to 64 rows of 64 bytes each. The vectors would also increase to 64 bytes. But the cost of the operation is $O(n^3)$. The reason 32 was selected was for space considerations. The dimension is large enough to create significant mixing. The presence of a 64K byte look-up table for the WaMM multiplication operator, \otimes , is a sizable imposition. Requiring $4\frac{1}{4}$ for the internal state (4K for the State matrix, S_M , and $\frac{1}{4}$ K for the 4 vectors) was deemed to be too great. A second, competing consideration regarding the size of the state matrix, SM, was computational load. The number of table lookups (WaMM multiplications) is a function of N^3 where N is the number of bytes in a row of the State Matrix, SM. Doubling the size of SM, increases the computational load of the algorithm by a factor of eight. A size of 32 bytes seemed to be a reasonable trade off between making the State Matrix, SM, as large as possible (in order to tap the matrix without overlaps) and the computation load.

The repeating the squaring operation insures, that even a single bit change affects every bit in every byte of the resulting matrix.

Matrix multiplication requires two operators; multiplication and addition. The natural choice for the operations would be multiplication and addition modulo 256. This choice though leads to a great deal of algebraic structure because the two operations and the set of byte values form a field. Such a structure opens up the possibility of attacking the round function by linearizing the operations. In order to diminish the structure created by a field, multiplication modulo 256 was replaced by the WaMM multiplication operator, \otimes . The other problem with using the naïve field of multiplication and addition modulo 256 is that 256 is not a prime number. Because of this both multiplication and addition modulo 256 of non-zero operands can create a result of zero. Once the value of zero is achieved, multiplication modulo 256 will propagate the zero value. This includes a bias in the system for the results to tend to zero slightly. Such a bias interferes with the diffusion of information.

The WaMM multiplication operation, \otimes , was selected to minimize auto-correlation and to minimize cross-correlation between the WaMM multiplication operator and the flat operator. The flat operator is defined as: $j \otimes k = (j + k) \bmod 256$. A high cross-correlation with the flat operator means there exist a high correlation with an affine transformation of j and k . An affine transformation of j and k is $j \otimes k = (Aj + Bk + C) \bmod 256$ for some integers A , B , and C .

The candidate operators were treated as two dimensional arrays of 256 rows of 256 integers each. The auto and cross correlations were then calculated using the Fast Walsh-Hadamard Transformation (FWHT) for two dimensions and its inverse transformation. The result of the transformations were two matrices consisting of 256 rows of 256 bytes. One matrix contained correlation values of the operator with shifted versions of itself. One matrix contained correlation values of the operator with shifted versions of flat operator. Each matrix of 65,536 correlation values was reduced to a single large number by using the sum of the square of each matrix element. Scaled version of these two numbers (the sum of squares of the auto and cross correlations) were added together to form the fitness measure. The scaling was to insure the auto and cross correlation would have equal weight in determining if one operator was better than another. An operator with a smaller fitness measure (lower auto correlation and/or lower cross correlation) was deemed "better" than an operator with a larger fitness measure.

A candidate operator tested for improvement by swapping one pair of rows or swapping one pair of columns. If the new operator with the swap had a lower fitness measure than the original operator, the new operator with the swap was kept. The testing for improvement continued until no row swap or column swap would reduce the fitness

measure of the operator. If no single row swap and no single column swap would reduced the fitness measure of the operator, the operator is called a locally optimal operator because the operator is at a local optimum relative to the fitness measure. Once the operator was found to be locally optimal, the operator was saved, shuffled, and the search for a new locally optimal operator was begun. The operator was shuffled by shuffling the 256 rows of the operator matrix and then shuffling the 256 columns of the operator matrix. By swapping and shuffling whole rows and whole columns the closure of the operator is maintained. From all the locally optimal operators found, the locally optimal operator with the lowest fitness measure was then selected to be the WaMM multiplication operator.

The C# application which performed this search is available among the Supplemental Materials.

The author considered defining both multiplication and addition via truth tables. This would have degraded the algebraic structure further and would eliminate the bias toward zero still present with addition modulo 256. The slight bias toward zero present with addition modulo 256 is offset by:

- (1) The computational simplicity of addition modulo 256
- (2) opportunity to mix bits via two unrelated processes; addition modulo 256 and substitution via a complex set of 8-bit to 8-bit substitution boxes (S-Boxes).
- (3) Avoiding a second, 64K of storage for the truth table of the the WaMM Addition Operator, \boxplus

The number of iterations of matrix multiplication during the WaMM round function was set at 8 because this is 4 times the number of iteration needed for either a single bit change in the Input Vector, V_{in} or a single bit change in the State Matrix, SM, to propagate to every bit of the result, $V_L \oplus V_R$. The statement; *Every bit in the result: $V_L \oplus V_R$ is affected.* is not the same as the statement: *Every bit in the result: $V_L \oplus V_R$ tends to an expected value of $\frac{1}{2}$.* Eight (8) iterations of matrix multiplication was deemed an acceptable trade off between computational effort (more iteration; more effort) and the expectation that every bit tend to $\frac{1}{2}$ (more iteration; the closer the asymptotic approach). There is a point of diminishing returns where the slight improvement of the expected values to tend to $\frac{1}{2}$ is not worth the extra effort required to get the improvement. Eight was the somewhat arbitrary cutoff where the improvement in diffusion did not justify the addition effort.

7. ALTERNATE DESIGNS

An alternate design was to replace the basic unit of the internal state from bytes to words of either 16, 32, or 64 bits. If a word size larger than a byte is used the an operation similar to the WaMM multiplication operator, \otimes , because the truth table impractically large. For a word size of N bits, the truth table would require 2^N rows of 2^N words each. The number of bits in the truth table is $N2^{(2N)}$. A word size of 8 bits (one byte) is the practical upper limit for a truth-table implementation of a binary operator.

The naïvé selection of operators would be multiplication and addition modulo 2^R where R is the length of the word in bits; $R \in \{16, 32, 64\}$. For the reasons outlined in Section 6, the fact that two non-zero elements can be multiplied to produce a zero results is significant problem. The alternative to multiplication modulo 2^R is to do the multiplication and addition using the Galois Field, $\text{GF}(2^R)$ for some irreducible polynomial of order R . The tapping of the state material would still be at the byte level using a more complex version of the tapping sequence defined in Eq. (8). For the purposes of tapping the state matrix words would treated a collection of bytes.

The author likes this approach. The benefits are:

- (1) The word size is flexible.
- (2) The operation of bitwise-XOR, addition in $\text{GF}(2^R)$, is fast and easily implemented.
- (3) The the shift and XOR algorithm for multiplication in $\text{GF}(2^R)$, is not as fast as table lookups, but is reasonably fast.
- (4) The larger the word size the fewer computational operations are required for matrix multiplications. The computational effort of a multiplication in $\text{GF}(2^R)$ may be more than the effort for a table lookup, but there are fewer multiplications in $\text{GF}(2^R)$ required.

The drawbacks are:

- (1) The mathematics has a high degree of algebraic structure. This opens up the possibility the round function can be attacked by treating the round function as set of simultaneous, algebraic equations.
- (2) Identity elements exist for both operations and inverses exist for every element so inversion is possible.
- (3) Because inverses exist it is possible one iterations may be the inverse of prior iteration. Leaving open the possibility of weak

hashes because the mixing in one step is undone by the mixing operation of a later step.

The author believes the risks posed by the Galois field, $GF(2^R)$ are too great. The WaMM operator, \otimes , can be thought of a 256 S-Boxes which map eight bits to eight bits. The left hand operand of $j \otimes k$ selects the S-Box. The right hand operand of $j \otimes k$ is the input to the S-Box. The output of the S-Box is the result of $j \otimes k$.

The measurement and attacks on the non-linearity of S-Boxes are well understood. The attacks are differential analysis of the S-Box, linear approximation of the Boolean functions, and linear approximation of the whole S-Box. The measurements are the strict avalanche criterion and auto correlation of the 8 boolean functions which make up bytes of the S-Box. Using a fitness measure which minimizes auto correlation hardens the S-Box against attacks using differential analysis. Using a fitness measure which minimizes cross correlation to affine transformations hardens the S-Box against attacks using linear approximation.

The author acknowledges his preference for using S-Boxes over using algebraic constructions for introducing non-linearity to a cryptographic system. Linear analysis is a powerful tool and highly structured algebraic components only aid the possible effectiveness of an attack using linear analysis. Differential analysis is also a powerful tool against S-Boxes but one for which the hardening is well understood.

8. PROS AND CONS OF WAMM

Simultaneously the biggest drawback and strength of the WaMM hash algorithm is the choice to use matrix multiplication as the mixing operator. From a computation point of view, matrix multiplication is an expensive operation. The number of operations needed to square a matrix once are:

- (1) N^3 table lookups (WaMM multiplications); where N is the dimension of the square matrix.
- (2) N^3 table additions modulo 256; where N is the dimension of the square matrix.

Matrix multiplication though is great operator for mixing the information in the bits of the message block across the 8192-bit sub-blocks of the message. With only two squarings, the information in a single message bit affects all of the bits of the state matrix (i.e. the 8192-bit sub-block of the message).

The expense of the repeated squaring is mitigated because the operation is performed infrequently; once every 8192 bits.

The second drawback is the requirement for a 64K lookup table for the WaMM multiplication operator, \otimes . For the reasons described in Section 6, it is believed the advantages deliberate non-linearity outweigh this memory requirement.

9. KNOWN ATTACKS

Because of the design of the WaMM multiplication operator, \otimes , the operator is resistant to both differential and linear cryptanalysis. Differential and linear cryptanalysis are two most powerful techniques for attacking encryption and hashing algorithms.

Message extension is a problem endemic to hashes using the Merkle-Damgard construction. The WaMM algorithm blunts message length padding because the number of bits making up the internal state of the hash far exceed the number of bits in the hash value. Because of this excess of state information, knowing the hash value and length of a target message does not provide enough information to construct a second message such that:

$$H(M||M^*) = H(M)$$

where $||$ is concatenation. This makes the WaMM hash algorithm pre-image and second pre-image resistant even if using the message extension to attack the hash.

10. PERFORMANCE

On the NIST reference platform, the WaMM algorithm can hash one gigabyte of data in 150 seconds. The hash time is independent of the hash length. This is a rate of approximately 7.1 megabytes per second.

Appendix A

Here is the truth table for the \otimes operator used in the WaMM algorithm.

The operator has measures of: [0x051A5134891F23E0, 0x000000088AF8CC40, 0x00000491A1A7C51F, 0x000023E0]

N128W12795 HIGHLAND ROAD, GERMANTOWN, WI 53022

URL: <http://www.WashburnResearch.org>

E-mail address: crypto@WashburnResearch.org