BLENDER

A Proposed New Family of Cryptographic Hash Algorithms

Colin Bradbury, MBA, Ph.D.

25th October, 2008

Abstract:

This proposal is submitted in response to the Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family published in the Federal Register, Volume 72, No 212, Friday November 2nd, 2007.

This proposal introduces a family of cryptographic algorithms suitable for Digital Signatures, Message Authentication, and Random Number Generation. These algorithms can be used as a direct drop-in substitution for the existing SHA-2 family of algorithms. These algorithms were explicitly designed to run on 8-bit microcontrollers, using only the most primitive arithmetic operations. Consequently, they can be implemented on a wide range of platforms. These algorithms are believed to be resistant to all known attacks; a preliminary analysis of the security of these algorithms is included.

CONTENTS:

1. Summary and Checklist

1.1 Algorithm Specifications and Supporting Documentation

The complete specification of the submitted algorithm and the design rationale is presented in section 2 below.

1.2 Statements about Estimated Computational Efficiency and Memory Requirements

The following estimates are based on actual measurements made on the platforms indicated with a 1000 bit message. Different message lengths may yield different results.

1.2.1 Estimates (memory requirements and speed) on NIST Reference Platform (32 bit)

Platform/processor used:	Intel Pentium III
Clock speed:	664 MHz
Memory:	512 KB
Operating system:	Windows XP Professional
Compiler:	Microsoft Visual Studio 2005

(For 224-bit message digest) Number of clock cycles required to:

1.	generate one message digest, and	12,834
2.	set up the algorithm (e.g. build internal tables)	707
3.	memory requirements	118 bytes

(For 256-bit message digest) Number of clock cycles required to:

1.	generate one message digest, and	13,228
2.	set up the algorithm (e.g., build internal tables)	755
3.	memory requirements	126 bytes

(For 384-bit message digest) Number of clock cycles required to:

1.	generate one message digest, and	20,546
2.	set up the algorithm (e.g., build internal tables)	659
3.	memory requirements	178 bytes

(For 512-bit message digest) Number of clock cycles required to:

1.	generate one message digest, and	21,553
2.	set up the algorithm (e.g., build internal tables)	735
3.	memory requirements	210 bytes

Speed improvements can be obtained by making the following tradeoffs:

1. Allocating static memory for the state structure instead of passing in the structure as a parameter

2. Having the incoming message buffer be word aligned and having an exact multiple of the word size in partial messages
3. Completing all of the message preparation before hashing begins

1.2.2 Estimates (memory requirements and speed) on NIST Reference Platform (64-bit)

Platform/processor used:	Intel Pentium III
Clock speed:	664 MHz
Memory:	512 KB
Operating system:	Windows XP Professional
Compiler:	Microsoft Visual Studio 2005

(For 224-bit message digest) Number of clock cycles required to:

1.	generate one message digest, and	12,834
2.	set up the algorithm (e.g., build internal tables)	707
3.	memory requirements	118 bytes

(For 256-bit message digest) Number of clock cycles required to:

1.	generate one message digest, and	13,228
2.	set up the algorithm (e.g., build internal tables)	755
3.	memory requirements	126 bytes

(For 384-bit message digest) Number of clock cycles required to:

1.	generate one message digest, and	19,568
2.	set up the algorithm (e.g., build internal tables)	641
3.	memory requirements	178 bytes

(For 512-bit message digest) Number of clock cycles required to:

1.	generate one message digest, and	20,527
2.	set up the algorithm (e.g., build internal tables)	711
3.	memory requirements	210 bytes

Speed improvements can be obtained by making the following tradeoffs:

1. Allocating static memory for the state structure instead of passing in the structure as a parameter
2. Having the incoming message buffer be word aligned and having an exact multiple of the word size in partial messages
3. Completing all of the message preparation before hashing begins

1.2.3 Estimates (memory requirements and speed) on 8-bit processors

Platform/processor used:	Silicon Labs C8051F120
Clock speed:	24.5 MHz
Memory:	8,448 Bytes

Operating system: None
Compiler: Keil Professional Developers Toolkit

(For 224-bit message digest) Number of clock cycles required to:

1. generate one message digest, and 370,416
2. set up the algorithm (e.g., build internal tables) 3,924
3. memory requirements 118 bytes

(For 256-bit message digest) Number of clock cycles required to:

1. generate one message digest, and 382,224
2. set up the algorithm (e.g., build internal tables) 4,728
3. memory requirements 126 bytes

(For 384-bit message digest) Number of clock cycles required to:

1. generate one message digest, and 534,132
2. set up the algorithm (e.g., build internal tables) 4,476
3. memory requirements 178 bytes

(For 512-bit message digest) Number of clock cycles required to:

1. generate one message digest, and 589,296
2. set up the algorithm (e.g., build internal tables) 6,492
3. memory requirements 210 bytes

Speed improvements can be obtained by making the following tradeoffs:

1. Allocating static memory for the state structure instead of passing in the structure as a parameter
2. Allocating selected state variables in near data memory instead of extended memory
3. Completing all of the message preparation before hashing begins
4. Rewriting the innermost hash computation in assembly language

1.2.4 Estimates of gate counts in hardware implementations:

(For 224-bit message digest): 3178

(For 256-bit message digest): 3530

(For 384-bit message digest): 5642

(For 512-bit message digest): 7050

1.3 Statement of expected strength (i.e. work factor) of the algorithm for each of the security requirements specified in Sections 4.A.ii and 4.A.iii and for each message digest size (224, 256, 384, and 512 bits)

A summary of the strengths of the specified security requirements is given in the following table; the supporting rationale is presented in section 3 below. The security strengths of m-bit subsets are these figures pro-rated according to the message digest size.

Message digest size	**224 bits**	**256 bits**	**384 bits**	**512 bits**
First preimage resistance	224 bits	256 bits	384 bits	512 bits
Second preimage resistance	224 bits	256 bits	384 bits	512 bits
Collision resistance	112 bits	128 bits	192 bits	256 bits
Multi-collision resistance	224 bits	256 bits	384 bits	512 bits
Length-extension resistance	224 bits	256 bits	384 bits	512 bits
HMAC	224 bits	256 bits	384 bits	512 bits
PRF	224 bits	256 bits	384 bits	512 bits
Randomized hashing	224 bits	256 bits	384 bits	512 bits

1.4 Cryptanalysis with respect to known attacks and their results

An analysis of the resistance to each of the known attacks is presented in section 3 below.

1.5 Explanation of the provenance of any constants or tables used

The Initial Values used for each message digest size are copied verbatim from the corresponding SHA-2 algorithm described in FIPS 180-3. The rationale is that if these initial values are good enough for SHA-2 then they should also be good enough for SHA-3. All other variables are initialized to zero. No other constants or tables are used by this algorithm.

1.6 References to any published materials analyzing the security of the submitted algorithm

The submitted algorithm employs new techniques for computing hash functions, different from previous methods. Consequently, there are no published materials in existence at the time of submission. The security analysis presented in section 3 below necessarily had to start from scratch; all relevant references are given in that section.

1.7 Statement on the advantages and limitations of the algorithm

The submitted algorithm has the following advantages:

1. It is explicitly designed to run on 8-bit microcontrollers.
2. It can be implemented on a wide variety of platforms

3. It permits efficient hardware implementations
4. It provides a drop-in replacement for the SHA-2 algorithms in any applications that use SHA-2 algorithms as their hash function
5. A modified version provides a drop-in replacement for the SHA-1 algorithm in any application using that algorithm
6. It is extendable to larger message digest sizes

The submitted algorithm is limited to the requirements stated in the Request for Candidate Algorithm Nominations. Any additional requirements may require modifications to the proposed algorithm.

1.8 Statements about Optical Media

1.8.1 Reference Implementation in ANSI C with comments

The reference implementation is presented on the companion CD in the folder \Reference Implementation. This implementation includes the header file defining the NIST test API; the reference implementation uses that API.

1.8.2 Optimized Implementations in ANSI C with comments

Implementations for 64-bit and 32-bit computers are presented on the companion CD in their respective folders. No optimization of the submitted algorithm is possible within the constraints of ANSI C, so these implementations are copies of the reference implementation.

1.8.3 Supporting Documentation

Electronic copies of all written materials have been provided in PDF on the companion CD in the folder \Supporting Documentation.

2. Algorithm Specifications

This document specifies a hash algorithm for four message digest sizes: 224 bits, 256 bits, 384 bits and 512 bits. The first two use 32-bit words and a block size of 512 bits, similar to SHA-224 and SHA-256; the last two use 64-bit words and a block size of 1024 bits, similar to SHA-384 and SHA-512. Apart from the word size and the message digest size, the four algorithms are almost identical in their operations. They are presented separately below to avoid confusion.

2.1. Definitions

Unless noted otherwise, this document uses the same definitions, notation and conventions as used in The Secure Hash Standard, FIPS 180-3.

2.1.1. Operations on Words

The following operations are applied to w-bit words and other integer values in all the algorithms presented in this document:

1. Bitwise XOR ("exclusive-OR") operation: $\oplus$

2. Bitwise complement operation: $\neg$

3. The right shift operation: >>

4. The *rotate right* (circular right shift) operation $\mathbf{ROTR^n}$ (x) where x is a w-bit word and n is an integer or integer variable with value between 0 and w.

5. The *rotate left* (circular left shift) operation $\mathbf{ROTL^n}$ (x) where x is a w-bit word and n is an integer or integer variable with value between 0 and w.

6. Addition modulo 2^w

7. Addition with Carry

   The operation [v, z] = x + y + c is defined as follows. The words x and y represent integers X and Y, where $0 <= X < 2^w$ and $0 <= Y < 2^w$. The variable c represents an integer C with value either 0 or 1 (the carry in). Compute

   $S = X + Y + C$
   $Z = S \bmod 2^w$
   and $V = S >> w$

   Convert the integers Z and V to a word and a bit respectively and define z = Z and v = V. This operation corresponds to the assembly language ADDC instruction. The [v,z] notation is taken from Verilog hardware description language.

2.1.2. Parameters

The following parameters are used in the algorithm specifications in this document

a0, a1, … a7	Working variables that are the w-bit words of state used in the computation of the hash values
c1, c2	Working variables that are the one-bit carry in and carry out of addition-with-carry operations
F	Fill data derived from the message M
H0 … H7	Working variables that are the w-bit words which accumulate the hash value.

k	Number of fill bytes appended to a message during preparation
ℓ	Length of the message, M, in bits
$\ell\ell$	Length of the message length ℓ in bytes
M	The message to be hashed
P	The message M with padding bits appended
p	Length of the padded message in bytes
r	Working variable holding the number of bits to be rotated during the computation of hash values
T, T1, T2	Temporary w-bit words used in the hash computations
w	Number of bits in a word
W_t	The t^{th} w-bit word of the message schedule

2.1.3 Functions

This specification includes no functions other than the operations stated above.

2.1.4 Constants

These algorithms use no constants other than the Initial Values for each hash computation.

2.2 Message Length Management

The message length ℓ is stored in a multi-byte variable L such that the first byte of L holds the eight least-significant bits of ℓ, the second byte holds the next eight bits and so on until the entire message length is accommodated. This is equivalent to the memory organization on an 8-bit little-endian microprocessor. Trailing zero bytes are eliminated and the resulting size (in bytes) of L is stored in a one-byte variable $\ell\ell$, the length of the length. For messages less than 256 bits in length, L is one byte long and $\ell\ell$ holds the value 1. For messages longer than 255 bits but shorter than 65536 bits (2^{16}), L is two bytes long and $\ell\ell$ holds the value 2. Longer messages are handled similarly. The maximum size of L is 255 bytes. Implementers of these algorithms may impose a lower limit on the size of L, thereby restricting the maximum length of messages that can be processed by that implementation. Implementers may also aggregate the bytes of L into

larger sized unsigned integers provided that such aggregation does not result in high order zero bytes being introduced. Typical aggregates may be 32-bit or 64-bit words.

> Discussion: The intention here is that there should be no practical limit on the length of messages that can be hashed but short messages should not be burdened with the overhead of accommodating potentially very large message lengths.

2.3 Message Preparation

Message preparation should take place before hash computation begins. This preparation consists of five steps: padding the message M if necessary, filling out the padded message P to the requisite length, appending the message lengths, parsing the resulting message into w-bit words, and finally appending the message checksums.

2.3.1 Padding the Message

If the length of the message M is an exact multiple of 8 bits, no padding is added and the padded message P is identical to the original message M. Otherwise, the complement of the last bit of the message shall be appended repeatedly until the resulting length reaches the next exact multiple of 8 bits. The amount of padding added is at most seven bits. The length of the padded message in bytes is given by the formula

$$\boldsymbol{p} = (\boldsymbol{\ell} + 7) >> 3$$

After this step, the padded message is handled in terms of bytes as they exist in computer memory rather than bits. The method of packing bits into bytes is left to the system designer since such packing will typically be done before passing the message to this algorithm.

2.3.2 Filling Out the Padded Message

The fill data F is the padded message P truncated to 13 bytes if necessary, unless the message M has zero length in which case F is 13 bytes of all zeros. The amount of fill data to be appended to the padded message depends on the block size and the message length. For a block size of 512 bits and a message length less than 512 bits the number of fill bytes needed (k) is given by

$$k = 119 - \boldsymbol{p} - \boldsymbol{\ell\ell}$$

For a block size of 512 bits and a message length greater than 511 bits the number of fill bytes needed is given by

$$k = 64 - ((\boldsymbol{p} + \boldsymbol{\ell\ell} + 9) \bmod 64)$$

For a block size of 1024 bits and a message length less than 1024 bits the number of fill bytes needed is given by

$$k = 239 - \boldsymbol{p} - \boldsymbol{\ell\ell}$$

For a block size of 1024 bits and a message length greater than 1023 bits the number of fill bytes needed is given by

$$k = 128 - ((\boldsymbol{p} + \boldsymbol{\ell\ell} + 17) \bmod 128)$$

The fill data F is concatenated with itself repeatedly until it reaches or exceeds *k* bytes in length, then it is truncated to *k* bytes and appended to the padded message P.

> Discussion: Ideally, the fill data should be a full block, 512 bits or 1024 bits depending on the message digest length. Unfortunately, for long messages presented in multiple pieces, the fill data has to be preserved across calls within the algorithm state structure. Saving a full block is not acceptable on an 8-bit microcontroller, so 13 bytes is a compromise. It is simply a prime number greater than 8 bytes so that there are no easily discernable patterns generated in the filled message. A smaller value can be specified if there is a good reason to do so. Filling with a constant value such as zero is seen as a security risk and so should be avoided.

2.3.3 Appending the Message Lengths

After the message has been filled to the appropriate length, the message length as held in the byte array L is appended to the message least significant byte first. The single byte $\ell\ell$, the length of the length, is then appended to the result to complete the assembled message. The latter should be two w-bit words short of an exact multiple of the block size.

2.3.4 Parsing the Assembled Message

After a message has been assembled as described above, it must be parsed into a number of w-bit words before the hash computation can begin. The first byte of the message becomes the least significant byte of the first w-bit word and successive bytes of the message become the progressively higher order bytes within the word. Successive words are defined similarly. Note that if the message buffer is word aligned on a little-endian processor, this step can be achieved with almost zero work by casting the buffer from an array of bytes to an array of w-bit words (or the equivalent union declaration).

2.3.5 Appending the Checksums

The final step in preparing the message is to append two w-bit checksum words. The first checksum is the complement of the sum modulo 2^w of all the w-bit words in the parsed message. The second checksum is the sum modulo 2^w of the complement of all the w-bit words in the parsed message:

$$\text{checksum1} = \neg \sum W_t$$

$$\text{checksum2} = \sum (\neg W_t)$$

After the checksums have been appended to the parsed message, it should be an exact multiple of the block size for the selected algorithm. The message can now be given to the hash algorithm to compute the message digest.

> Discussion: The checksum calculations were chosen for the ease of implementation and efficiency of computation on 8-bit microprocessors. These checksums are the cornerstones of hash security.

2.4 Hash algorithm for 512-bit message digests

The 512-bit algorithm uses eight 64-bit working variables, a0 to a7, eight 64-bit result variables, H0 to H7, and two single-bit carry variables, c1 and c2; these constitute the "state" of the algorithm carried from round to round. This algorithm also uses three 64-bit intermediate values, T, T1 and T2, and one intermediate integer value ***r*** used to hold a rotation factor.

2.4.1 Initial Values

Before hash computation begins, the working variables, a0 to a7, are initialized to the following eight 64-bit words in hex:

```
a0 = 6a09e667f3bcc908
a1 = bb67ae8584caa73b
a2 = 3c6ef372fe94f82b
a3 = a54ff53a5f1d36f1
a4 = 510e527fade682d1
a5 = 9b05688c2b3e6c1f
a6 = 1f83d9abfb41bd6b
a7 = 5be0cd19137e2179
```

These values are identical to the initial values for SHA-512, copied verbatim from FIPS 180-3. These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the first eight prime numbers. All of the other variables are all initialized to zero.

2.4.2 Hash Computation

After the message has been prepared and the variables initialized, perform the following computations for each 64-bit word W_t in the prepared message:

1. Compute the preliminary intermediate values using add-with-carry:

$$[c1,T1] = (a5 \oplus W_t) + (a1 \oplus \mathrm{ROTL}^8 (a3)) + c1$$
$$[c2,T2] = (a0 \oplus \mathrm{ROTR}^8 (W_t)) + (a4 \oplus \mathrm{ROTR}^8 (a2)) + c2$$

2. Compute the rotation factor:

$$\boldsymbol{r} = 8 - (c1 + c2)$$

3. Rotate the intermediate values:

$$T1 = \mathrm{ROTL}^{\boldsymbol{r}} (T1)$$

$T2 = ROTR^{r} (T2)$

4. Compute the next state:

$T = ROTR^{7} (a0)$
$a0 = a1 \oplus T2$
$a1 = a2 \oplus T1$
$a2 = a3 \oplus T2$
$a3 = a4 \oplus T1$
$a4 = a5 \oplus T2$
$a5 = a6 \oplus T1$
$a6 = a7 \oplus T2$
$a7 = T \oplus T1$

5. Update the hash result variables:

$H0 = H0 + a0$
$H1 = H1 + a1$
$H2 = H2 + a2$
$H3 = H3 + a3$
$H4 = H4 + a4$
$H5 = H5 + a5$
$H6 = H6 + a6$
$H7 = H7 + a7$

These five steps constitute one round of the algorithm. After repeating these steps for each word in the prepared message, the resulting 512-bit message digest of the message M is

$H0 \,||\, H1 \,||\, H2 \,||\, H3 \,||\, H4 \,||\, H5 \,||\, H6 \,||\, H7$

Human-readable output is generated most significant byte first.

Discussion: In a pure block cipher, the hash result variables, H0 to H7, would be updated only once per block. However, doing that has been shown to be a serious flaw in the SHA-2 family of hash functions. Consequently, these variables are updated in every round in the algorithms presented here.

2.5 Hash algorithm for 384-bit message digests

The 384-bit algorithm uses six 64-bit working variables, a0 to a5, six 64-bit result variables, H0 to H5, and two single-bit carry variables, c1 and c2; these constitute the "state" of the algorithm carried from round to round. This algorithm also uses three 64-bit intermediate values, T, T1 and T2, and one intermediate integer value ***r*** used to hold a rotation factor.

2.5.1 Initial Values

Before hash computation begins, the working variables, a0 to a5, are initialized to the following six 64-bit words in hex:

```
a0 = cbbb9d5dc1059ed8
a1 = 629a292a367cd507
a2 = 9159015a3070dd17
a3 = 152fecd8f70e5939
a4 = 67332667ffc00b31
a5 = 8eb44a8768581511
```

These values are identical to the initial values for SHA-384, copied verbatim from FIPS 180-3. These words were obtained by taking the first sixty-four bits of the fractional parts of the square roots of the ninth through fourteenth prime numbers. All of the other variables are all initialized to zero.

2.5.2 Hash Computation

After the message has been prepared and the variables initialized, perform the following computations for each 64-bit word W_t in the prepared message:

1. Compute the preliminary intermediate values using add-with-carry:

$$[c1,T1] = (a5 \oplus W_t) + (a1 \oplus \mathrm{ROTL}^8 (a3)) + c1$$
$$[c2,T2] = (a0 \oplus \mathrm{ROTR}^8 (W_t)) + (a4 \oplus \mathrm{ROTR}^8 (a2)) + c2$$

2. Compute the rotation factor:

$$\boldsymbol{r} = 8 - (c1 + c2)$$

3. Rotate the intermediate values:

$$T1 = \mathrm{ROTL}^{\boldsymbol{r}} (T1)$$
$$T2 = \mathrm{ROTR}^{\boldsymbol{r}} (T2)$$

4. Compute the next state:

$$T = \mathrm{ROTR}^7 (a0)$$
$$a0 = a1 \oplus T2$$
$$a1 = a2 \oplus T1$$
$$a2 = a3 \oplus T2$$
$$a3 = a4 \oplus T1$$
$$a4 = a5 \oplus T2$$
$$a5 = T \oplus T1$$

5. Update the hash result variables:

H0 = H0 + a0
H1 = H1 + a1
H2 = H2 + a2
H3 = H3 + a3
H4 = H4 + a4
H5 = H5 + a5

These five steps constitute one round of the algorithm. After repeating these steps for each word in the prepared message, the resulting 384-bit message digest of the message M is

H0 || H1 || H2 || H3 || H4 || H5

Human-readable output is generated most significant byte first.

2.6 Hash algorithm for 256-bit message digests

The 256-bit algorithm uses eight 32-bit working variables, a0 to a7, eight 32-bit result variables, H0 to H7, and two single-bit carry variables, c1 and c2; these constitute the "state" of the algorithm carried from round to round. This algorithm also uses three 32-bit intermediate values, T, T1 and T2, and one intermediate integer value ***r*** used to hold a rotation factor.

2.6.1 Initial Values

Before hash computation begins, the working variables, a0 to a7, are initialized to the following eight 32-bit words in hex:

```
a0 = 6a09e667
a1 = bb67ae85
a2 = 3c6ef372
a3 = a54ff53a
a4 = 510e527f
a5 = 9b05688c
a6 = 1f83d9ab
a7 = 5be0cd19
```

These values are identical to the initial values for SHA-256, copied verbatim from FIPS 180-3. These words were obtained by taking the first thirty-two bits of the fractional parts of the square roots of the first eight prime numbers. All of the other variables are all initialized to zero.

2.6.2 Hash Computation

After the message has been prepared and the variables initialized, perform the following computations for each 32-bit word W_t in the prepared message:

1. Compute the preliminary intermediate values using add-with-carry:

$$[c1,T1] = (a5 \oplus W_t) + (a1 \oplus ROTL^{8} (a3)) + c1$$
$$[c2,T2] = (a0 \oplus ROTR^{8} (W_t)) + (a4 \oplus ROTR^{8} (a2)) + c2$$

2. Compute the rotation factor:

$$\boldsymbol{r} = 8 - (c1 + c2)$$

3. Rotate the intermediate values:

$$T1 = ROTL^{\boldsymbol{r}} (T1)$$
$$T2 = ROTR^{\boldsymbol{r}} (T2)$$

4. Compute the next state:

$$T = ROTR^{7} (a0)$$
$$a0 = a1 \oplus T2$$
$$a1 = a2 \oplus T1$$
$$a2 = a3 \oplus T2$$
$$a3 = a4 \oplus T1$$
$$a4 = a5 \oplus T2$$
$$a5 = a6 \oplus T1$$
$$a6 = a7 \oplus T2$$
$$a7 = T \oplus T1$$

5. Update the hash result variables:

$$H0 = H0 + a0$$
$$H1 = H1 + a1$$
$$H2 = H2 + a2$$
$$H3 = H3 + a3$$
$$H4 = H4 + a4$$
$$H5 = H5 + a5$$
$$H6 = H6 + a6$$
$$H7 = H7 + a7$$

These five steps constitute one round of the algorithm. After repeating these steps for each word in the prepared message, the resulting 256-bit message digest of the message M is

H0 || H1 || H2 || H3 || H4 || H5 || H6 || H7

Human-readable output is generated most significant byte first.

2.7 Hash algorithm for 224-bit message digests

The 224-bit algorithm uses seven 32-bit working variables, a0 to a6, seven 32-bit result variables, H0 to H6, and two single-bit carry variables, c1 and c2; these constitute the "state" of the algorithm carried from round to round. This algorithm also uses three 32-bit intermediate values, T, T1 and T2, and one intermediate integer value ***r*** used to hold a rotation factor.

2.7.1 Initial Values

Before hash computation begins, the working variables, a0 to a6, are initialized to the following seven 32-bit words in hex:

```
a0 = c1059ed8
a1 = 367cd507
a2 = 3070dd17
a3 = f70e5939
a4 = ffc00b31
a5 = 68581511
a6 = 64f98fa7
```

These values are identical to the initial values for SHA-224, copied verbatim from FIPS 180-3 Change Notice 1. All of the other variables are all initialized to zero.

2.7.2 Hash Computation

After the message has been prepared and the variables initialized, perform the following computations for each 32-bit word W_t in the prepared message:

1. Compute the preliminary intermediate values using add-with-carry:

$$[c1,T1] = (a5 \oplus W_t) + (a1 \oplus \mathrm{ROTL}^8 (a3)) + c1$$
$$[c2,T2] = (a0 \oplus \mathrm{ROTR}^8 (W_t)) + (a4 \oplus \mathrm{ROTR}^8 (a2)) + c2$$

2. Compute the rotation factor:

$$\boldsymbol{r} = 8 - (c1 + c2)$$

3. Rotate the intermediate values:

$T1 = ROTL^{r}(T1)$
$T2 = ROTR^{r}(T2)$

4. Compute the next state:

$T = ROTR^{7}(a0)$
$a0 = a1 \oplus T2$
$a1 = a2 \oplus T1$
$a2 = a3 \oplus T2$
$a3 = a4 \oplus T1$
$a4 = a5 \oplus T2$
$a5 = a6 \oplus T1$
$a6 = T \oplus T2$

5. Update the hash result variables:

$H0 = H0 + a0$
$H1 = H1 + a1$
$H2 = H2 + a2$
$H3 = H3 + a3$
$H4 = H4 + a4$
$H5 = H5 + a5$
$H6 = H6 + a6$

These five steps constitute one round of the algorithm. After repeating these steps for each word in the prepared message, the resulting 224-bit message digest of the message M is

$H0 \,||\, H1 \,||\, H2 \,||\, H3 \,||\, H4 \,||\, H5 \,||\, H6$

Human-readable output is generated most significant byte first.

2.8 Design Rationale

This proposal explicitly targets implementation on 8-bit microcontrollers, so the operations involved are restricted to those found in all microprocessors – addition, word rotation, bitwise complement and exclusive-or. The rotation factors are chosen so that they can be accomplished by a byte shuffle followed by an optional one-bit shift (occasionally repeated). Addition and exclusive-or operations do not commute with each other, so these operations are interleaved in the computation.

The proposed algorithms are intended as drop-in replacements for the corresponding SHA-2 algorithms, so they mimic the SHA-2 algorithms where it is prudent to do so – initial values, block size, and partitioning the message. The objective is to minimize the changes needed in applications currently using SHA-2 algorithms when upgrading to these algorithms.

The proposed algorithms differ from the SHA-2 family in that the internal state is kept separate and distinct from the hash value output, and the latter is not recirculated into any subsequent computations. In contrast to the Merkle-Damgard construct in which only the hash value passes from block to block, here both the hash value and the internal state are passed along. The internal state is discarded at the end of the hash computation.

Previous research indicates that a collision in the hash function often stems from a collision in the internal compression function, leading to a desire for a collision-free compression function. This proposal takes the alternative approach of deliberately permitting a multitude of collisions in the compression function, most of which are false trails leading nowhere. Evaluating all the possible predecessors to any given intermediate result is computationally difficult. The intermediate results are used to generate the next internal state and then discarded. The net effect in the pattern of 1s and 0s in the internal state is similar to an interference pattern obtained when shining light through a grating. Note that every variable changes in every round so that there are no short-cuts for working backward through the algorithm.

The objective of the hash computation algorithm is to deny computational feasibility in an attack. The objective of the message preparation is to deny flexibility in the choice of inputs to the hash computations. The two strategies acting together create the overall algorithm security.

The minimum number of rounds is 32, enforced by the message preparation procedure. A number smaller than this raises the concern that cryptanalysis may be possible.

3. Security Analysis

These algorithms are believed to be resistant to all known attacks and also to all the rumored attacks currently being investigated. A brief examination of each of the well known attacks is presented below. At the time of submission, there are no published materials analyzing the security of the submitted algorithm. Consequently, all analysis is from first principles but draws on material in the following publications:

"Design and Analysis of Hash Functions", Murali Krishna Reddy Danda
http://wallaby.vu.edu.au/adt-VVUT/uploads/approved/adt-VVUT20070911.160040/public/01front.pdf

"Hash functions: Theory, attacks and applications", Ilya Mironov
http://research.microsoft.com/users/mironov/papers/hash_survey.pdf

"Multi-collision Attack on the Compression Functions of MD4 and 3-pass HAVAL", Hongbo Yu and Xiaoyum Wang
Information Security and Cryptology – ICISC, ISBN 3540767878

"Second Preimages on n-bit Hash Functions for Much Less than 2^n Work", John Kelsey and Bruce Schneider
http://www.schneier.com/paper-preimages.pdf

"A Tutorial on Linear and Differential Cryptanalysis", Howard M. Heys
http://www.engr.mun.ca/~howard/PAPERS/ldc_tutorial.pdf

"Linear and Differential Cryptanalysis", Benjamin Toft Jakobsen, Mehdi Abyar, and Peter Sebastian Nordholt
http://www.daimi.au.dk/~ivan/LinDifAnalyse.pdf

3.1 First Preimage Finding

The message preparation method imposes rigid constraints on the last three words in the message schedule. As an attacker moves forward through the algorithm, his freedom of choice in the inputs disappears and the computation can no longer be directed towards the goal. In each round, the attacker has w bits of input to influence 6w to 8w bits of state – each change in the input is magnified and ripples outward through the rest of the computation.

Working backwards to find a message which hashes to a particular value, an attacker must choose an internal state for the start of the checksums round and a pair of checksums. Using the 256-bit message digest as an example, there are 2^{322} possible initial choices, each of which may or may not lead to a solution. Processing forward from the chosen internal state with the chosen checksums gives the increments which will be added to the previous partial hash value to create the given message digest. Subtracting these increments from the given message digest yields the partial hash value required going into the checksum rounds. The attacker must then find a message which generates the chosen internal state, generates the chosen checksums, and generates the derived partial hash value, all at the same time; one message must meet all three requirements. Finding messages that generate the chosen checksums is essentially a first-preimage attack on a two word message digest. The other two parts are equivalent to a first-preimage attack on a double sized message digest in SHA-2. The overall difficulty level for this approach is estimated at greater than a brute-force attack.

3.2 Second Preimage Finding

Given a first message m_1, an attacker can compute all the intermediate states and partial sums along the way to the message digest. Finding another message, different from the first, which generates some matching intermediate state and partial sum is not useful because the second message must also have the same checksums as the first message; otherwise the very last round will create a message digest different from that of the first message. Trying to match the checksums instead and working backward from there is a specific case of first preimage finding discussed above – the second message still has to generate the same internal state as the first message just prior to hashing the checksums. Directly or indirectly, the attacker is forced to chose a pair of checksums, effectively at random, and then hope for a solution – this is essentially a brute-force attack.

3.3 Collision Finding

A Birthday Attack will succeed against all hash algorithms, including this one; the only defense is to increase the size of the message digest. Apart from the Birthday attack, there is no systematic way with this algorithm for generating messages with the same hash value. There is no good starting point for an attack. Starting with a first message is equivalent to the second preimage finding case discussed above. Starting elsewhere leads to one of the previous cases.

3.4 Multi-Collision Attack

These algorithms have the property that in general a collision is a dead end that goes nowhere. Given two messages A and B such that Hash(A) = Hash(B) then, in general, for any third message C, Hash (A||C) is NOT equal to Hash (B||C) and Hash(C||A) is NOT equal to Hash(C||B) where || denotes concatenation of the messages. Consequently, an attempt to find a three-way collision is effectively a Birthday attack followed by a First Preimage attack. Additional collisions can only be found by additional First Preimage attacks.

Constructing multi-collisions by the method proposed by Antoine Joux first requires a Birthday attack on the message digest and the internal state together. These contain n bits and n+2 bits respectively, so the initial Birthday attack involves $2^{(2n+2)/2} = 2^{n+1}$ operations. The subsequent Birthday attack to get the second half of the construction requires an additional $2^{n/2}$ operations.

3.5 Length Extension Attack

The message digest from a particular message has no relevance for hashing an extended message. The internal state at the end of the hash computation for the original message never appears in the external world so an attacker has no place to start from. To obtain the message digest for an extended message, the attacker must first recover the original message. Lengthening the message then changes the location of the start of the fill zone, the length encoding, and the checksums. Hashing the lengthened message is unlikely to generate the original message digest as a partial sum at any time. The conclusion is that no such attack is possible.

3.6 Keyed Hashing and Differential Cryptanalysis

In keyed hashing, either the message to be hashed is prefixed by a secret key or the Initial Value for the hash function is replaced by a secret key; otherwise the hash function operates as before. Differential cryptanalysis is an attack aimed at discovering the secret key by hashing messages with small differences between them.

With the proposed algorithm, a change in a single bit in the input message also means a difference in one or both of the checksums – the prepared message then has multiple bit changes. There are messages that have the same checksums, but then the differences in the message body are not simple. An analysis of the algorithm core shows that changing a single bit in the input

leads to changes in approximately half the bits of internal state within 8 to 12 rounds. This translates to changes throughout the final message digest. Without monitoring the partial hash values for every round, a bit change in the message digest cannot be traced back to any particular round - an observed change in the message digest may be the end result of multiple bit changes. All paths through the compression function to the intermediate values T1 and T2 are deliberately equidistant (except for those that pass through the carry bits) so there is no bias towards any particular word of state. There are so many paths through the system feeding into each bit in the message digest that no meaningful analysis is possible and no meaningful data can be extracted.

3.7 HMAC

The Keyed-Hash Message Authentication Code (HMAC) is specified in FIPS publication 198. The proposed algorithm is suitable for use in the HMAC standard in place of the corresponding SHA-2 algorithm. In HMAC, a secret key K is used to create a derived key K_0 which is then used in the following operation

$$\text{HMAC}(K, \text{text})_t = \text{Hash}((K_0 \oplus \text{opad}) \,||\, (\text{Hash}((K_0 \oplus \text{ipad}) \,||\, \text{text}))_t$$

where opad and ipad are fixed byte patterns and the result is limited to t bytes. The HMAC result has a value known to an attacker, as is the block size of both the input and output of the hash function. The attacker wishes to find a key K' such that

$$\text{HMAC}(K', \text{text})_t = \text{HMAC}(K, \text{text})_t$$

The steps involved in the attack are

1. Add the input and output block sizes to get the intermediate message length
2. Find a new message of that length that hashes to the final result
3. Exclusive-or the first block of the new message with opad to get the proposed K'
4. Calculate V = Hash ((K' ⊕ ipad) || text)
5. Compare V with the second block from step 2; if they match, security is broken

All of these steps are relatively simple except for step 2. Step 2 is a first preimage attack on the underlying hash function with a message of a specified length. Consequently, the security of the HMAC construct is equal to the security of the underlying hash function multiplied by a factor that depends on the block size and the parameter t.

3.8 Pseudo Random Functions (PRFs)

NIST Special Publication 800-90, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators", specifies two constructs that can use the submitted hash function in place of the corresponding SHA-2 algorithm to generate pseudo-random bit streams. These constructs are the Hash DRBG and the HMAC DRBG. In both cases, the output from a first hash computation is used as part of the input to a second hash computation and the generated result of the construct is the message digest output from the second hash computation

or multiple such message digests concatenated together. The requirement for such a construct is that the generated bitstream be indistinguishable from random.

The submitted algorithm uses only the operations of addition modulo 2^w, word rotation, bitwise complement and exclusive-or. Over the set of all possible inputs, each of these operations generates all the possible output values with equal probability. In addition, each bit position within an output word can be either 0 or 1 independently of any other bits in the word. The final message digest of the hash computation is the net result of many operations, so each bit position in the message digest has many opportunities to change its value. The Central Limit Theorem in statistics indicates that the probability of any bit in the message digest being a specific value, either 0 or 1, is approximately one half. Consequently, the generated bitstream will give the appearance of being random.

3.9 Randomized Hashing

The draft NIST Special Publication 800-106, "Randomized Hashing for Digital Signatures", specifies the method of modifying a message Ms with a random variable *rv* prior to being input to the hash computation. The submitted algorithm can be used as the hash function in this randomization method, replacing the corresponding SHA-2 algorithm. The modified message M is defined by

$$M = rv \,||\, ((Ms \,||\, padding) \oplus Rv) \,||\, rv\_length\_indicator$$

Where Rv is *rv* concatenated with itself the appropriate number of times and the length indicator is the length of *rv* expressed as a 16-bit integer and then bit-reversed.

Given the message Ms and random variable *rv*, an attacker wishes to find a second message m' and a second random variable r', different from Ms and *rv*, such that the hash computation yields the same result. The steps involved in the attack are

1. Find a message M' such that Hash (M') = Hash (M) and M' has six to eight trailing zeros
2. Remove the last 16 bits of M' to get the length of r'
3. Remove r' from the start of M', leaving a remainder Mx
4. Concatenate r' with itself appropriately and exclusive-or with Mx
5. Remove trailing zeros from the result, and
6. Remove the last '1' bit to get m'

After the first step, the procedure is relatively simple, so the only difficulty is in finding the message M' such that Hash (M') = Hash (M). This is a first preimage attack on the hash function so the security of the randomization method is effectively equal to the security of the underlying hash function.

3.10 Digital Signatures

The Digital Signature Standard (DSS) is specified in FIPS publication 186-3. This standard specifies three algorithms for digital signatures: the Digital Signature Algorithm (DSA), the RSA

Digital Signature Algorithm, and the Elliptical Curve Digital Signature Algorithm (ECDSA). Each of these algorithms uses a hash function in its computations. The submitted algorithm is suitable for use in all three algorithms, replacing the corresponding SHA-2 algorithm wherever the latter is used. The security of the digital signatures generated by each of the digital signature algorithms is at least equal to the security of the underlying hash function

3.11 Key Establishment

NIST Special Publication 800-56A, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", is concerned with the distribution of secret keys to authorized users. During key establishment, an entity may be required to compute a *MacTag* using an Approved MAC algorithm such as HMAC, which in turn must use an approved hash function such as SHA-2. The proposed algorithm can be used in place of the SHA-2 function. Such replacement can also be done in all other applications that use the SHA-2 hash functions.

Acknowledgement

The author wishes to thank the members of the SHA-3 Hash Forum for their timely assistance in completing this project.

APPENDIX A

Given the large number of applications using SHA-1 at the time of submission, it seems prudent to specify a drop-in replacement for that algorithm to ease the transition to the SHA-3 algorithms. This specification is given below.

A.1 Hash algorithm for 160-bit message digests

The 160-bit algorithm uses six 32-bit working variables, a0 to a5, five 32-bit result variables, H0 to H4, and two single-bit carry variables, c1 and c2; these constitute the "state" of the algorithm carried from round to round. This algorithm also uses three 32-bit intermediate values, T, T1 and T2, and one intermediate integer value ***r*** used to hold a rotation factor.

A.1.1 Initial Values

Before hash computation begins, the working variables, a0 to a5, are initialized to the following six 32-bit words in hex:

```
a0 = 67452301
a1 = efcdab89
a2 = 98badcfe
a3 = 10325476
a4 = c3d2e1f0
a5 = 0    (zero)
```

These values, except the last, are identical to the initial values for SHA-1, copied verbatim from FIPS 180-3. All of the other variables are all initialized to zero.

A.1.2 Hash Computation

Message preparation is performed as specified above for the 256-bit hash computation. After the message has been prepared and the variables initialized, perform the following computations for each 32-bit word W_t in the prepared message:

1. Compute the preliminary intermediate values:

$$[c1,T1] = (a5 \oplus W_t) + (a1 \oplus \mathrm{ROTL}^8 (a3)) + c1$$
$$[c2,T2] = (a0 \oplus \mathrm{ROTR}^8 (W_t)) + (a4 \oplus \mathrm{ROTR}^8 (a2)) + c2$$

2. Compute the rotation factor:

$$\boldsymbol{r} = 8 - (c1 + c2)$$

3. Rotate the intermediate values:

$T1 = ROTL^{r}(T1)$
$T2 = ROTR^{r}(T2)$

4. Compute the next state:

$T = ROTR^{7}(a0)$
$a0 = a1 \oplus T2$
$a1 = a2 \oplus T1$
$a2 = a3 \oplus T2$
$a3 = a4 \oplus T1$
$a4 = a5 \oplus T2$
$a5 = T \oplus T1$

5. Update the hash result variables:

$H0 = H0 + a0$
$H1 = H1 + a1$
$H2 = H2 + a2$
$H3 = H3 + a3$
$H4 = H4 + a4$

These five steps constitute one round of the algorithm. After repeating these steps for each word in the prepared message, the resulting 160-bit message digest of the message M is

$H0 \,||\, H1 \,||\, H2 \,||\, H3 \,||\, H4$

Human-readable output is generated most significant byte first.