A NEW HASH ALGORITHM: Khichidi-1

Natarajan Vijayarangan
Innovation labs, Tata Consultancy Services (TCS), Hyderabad, India
n.vijayarangan@tcs.com

Abstract

This is a technical report describing a new hash algorithm called Khichidi-1 and has been written in response to a Hash competition (SHA-3) called by National Institute of Standards and Technology (NIST), USA. This algorithm generates a condensed representation of a message called a Message Digest.

A group of functions used in the development of Khichidi-1 is described, followed by a detailed explanation of preprocessing approach. Differences between Khichidi-1 and NIST' SHA-2, the algorithm that is in use widely, have also been highlighted. Analytical proofs, implementations with examples, various attack scenarios, entropy tests, security strength and computational complexity of the algorithm are described as separate sections in this document.

Preface

Our research work describes a new design and analysis of hashing. A hash algorithm takes any message and produces a "fixed length value" in such a way that any two messages are unlikely to have the same fixed length value. This fixed length value is called a hash value / message digest. When two messages have the same hash value, this is known as a collision. A good hashing algorithm minimizes collisions for a given set of likely data inputs. To achieve this, we have started designing and analyzing a hash function that could be used for digital signature technology.

Attacks on MD-5, SHA-0 and SHA-1 by Wang et al [25, 26] has given a huge impetus to research in designing practical cryptographic hash functions as well as cryptanalysis of existing functions. In 1998, Hitachi Ltd had patented special purpose hash functions using collision free and one way properties. IBM had published SHA-IME (Improved Message Expansion) in 2005 to avoid differential attacks in SHA. Microsoft R&D had performed cryptanalysis of hash functions using Boolean Satisfiability (SAT) solvers. In 2007, Tata Consultancy Services (TCS) patented a cryptographic research work [27, 28], which introduces a message pre processing function (MP) that is bijective and used to reduce hash collisions. Later, TCS modified MP functions and applied them in designing Khichidi-1 algorithms [29].

All the research work on Khichidi-1 has been carried out in TCS Innovation labs, Hyderabad, India. The inventor, Dr. N. Vijayarangan, has handled the hash project work with the support from Dr. M. Vidyasagar, Mr. K. Ananth Krishnan, Mr. Sitaram Chamarty, Dr.Rajgopal Srinivasan, Dr. A. Sarangarajan, Mr. R. RamKumar, Mr. Chalamala Srinivasa Rao, Mrs. Priya V. Som, Ms. Divya Vyas and Ms. Mallika Choudhuri.

Table of Contents

1. Introduction

Our algorithm Khichidi-1 is an iterative, one-way hash function that can process a message to produce a condensed representation called a Message Digest. This algorithm determines the message's integrity: any change to the message will, with a very high probability, result in a different Message Digest. This property is useful in the generation and verification of digital signatures and message authentication codes, and in the generation of random numbers (bits).

The algorithm can be described in two stages:

- Preprocessing - Dividing a message into m-bit blocks, padding process takes place if number of bits of a message less than m-bits for the last block and setting up initialization values to be used in the hash computation.
- Hash Computation - is performed on each m-bit input block in the Cipher Block Chaining (CBC) mode. The final hash value generated by the hash computation is used to determine the message digest.

2. Definition

2.1 Glossary of Terms and Acronyms

Bit	A binary digit having a value of 0 or 1.
Byte	A group of 8 bits
Word	A group of 32 bits (4 bytes)

2.2 Algorithm Parameters, Symbols, and Terms

2.2.1 Parameters

The following parameters are used in the Khichidi-1 algorithm:

- $H^{(i)}$: The ith hash value. $H^{(0)}$ is the initial hash value; $H^{(N)}$ is the final hash value and is used to determine the Message Digest.
- $H_j^{(i)}$: The jth word of the ith hash value, where $H_0^{(i)}$ is the left-most word of hash value i.
- k: The number of zeroes appended to a message during the padding step.
- l: Length of the message, M, in bits.
- M: The message to be hashed.
- $M^{(i)}$: The message block i, with a size of m bits.
- $M_j^{(i)}$: The jth word of the ith message block, where $M_0^{(i)}$ is the left-most word of message block i.
- N: Number of blocks in the padded message.
- MDL: Message Digest Length
- GF: Galois Field
- MP: Message Preprocessing

2.2.2 Symbols

The following symbols are used in the Khichidi-1 algorithm:

^	Bitwise AND operation
V	Bitwise OR operation
⊕	Bitwise XOR ("exclusive -OR") operation
<<	Left-shift operation, where x << n is obtained by discarding the left-most n bits of the word x and then padding the result with n zeroes on the right.
>>	Right-shift operation, where x >> n is obtained by discarding the right- most n bits of the word x and then padding the result with n zeroes on the Left
||	Concatenation operation

3. Notation and Conventions

3.1 Bit Strings and Integers

The following terminology related to bit strings and integers is used:

1. A hex digit is an element of the set {0,1,...,9,a,...,f}. A hex digit is the representation of a 4-bit string. For example, the hex digit ‘7’ represents the 4-bit string ‘0111’, and the hex digit ‘a’ represents the 4-bit string ‘1010’.
2. A word is a 32-bit string that may be represented as a sequence of hex digits. To convert a word to hex digits, each 4-bit string is converted to its hex digit equivalent, as described in (1) above. For example, the 32-bit string 1010 0001 0000 0011 1111 1110 0010 0011 can be expressed as ‘a103fe23’. Throughout this specification, the *big-endian* convention is used when expressing words, so that within each word, the most significant bit is stored in the left-most bit position.
3. An integer between 0 and 2^{32} -1 inclusive may be represented as a 32-bit word. The least significant four bits of the integer are represented by the right-most hex digit of the word representation. For example, the integer 291 = $2^8 + 2^5 + 2^1 + 2^0$ = 256+32+2+1 is represented by the hex word 00000123.

3.2 Operations on Words

The following operations can be performed on words:

- Bitwise logical word operations: ^, v, $\oplus$
- The right shift operation SHR n(x), where x is a 32-bit word and n is an integer with $0 \leq n < 32$, is defined by SHR n (x) = x >> n.
- The left shift operation SHL n(x), where x is a 32-bit word and n is an integer with $0 \leq n < 32$, is defined by SHL n (x) = x << n.

4. Functions

Khichidi-1 Algorithm takes an input of any size and returns an output of fixed size (224, 256, 384, 512-bits). This algorithm has three functions, Shuffling(x), T-function(x), Lfsr(x), where x is a 32-bit word. The algorithm operates on 224/256/384/512 bits at a time in the CBC mode.

4.1 Shuffling Bits

The shuffling operation divides the message into 32-bit words and shuffles them. It interleaves the bits in two halves of each 32-bit word. The shuffling procedure used is an outer perfect shuffle, which means that the outer (end) bits remain in the outer positions. If the 32 bit word (where each element denotes a single bit) is $a_1a_2a_3a_4$ $a_5a_6a_7a_8$ $a_9a_{10}a_{11}a_{12}$ $a_{13}a_{14}a_{15}a_{16}$ $b_1b_2b_3b_4$ $b_5b_6b_7b_8$ $b_9b_{10}b_{11}b_{12}$ $b_{13}b_{14}b_{15}b_{16}$, then the outer perfect shuffle changes it to $a_1b_1a_2b_2$ $a_3b_3a_4b_4$ $a_5b_5a_6b_6$ $a_7b_7a_8b_8$ $a_9b_9a_{10}b_{10}$ $a_{11}b_{11}a_{12}b_{12}$ $a_{13}b_{13}a_{14}b_{14}$ $a_{15}b_{15}a_{16}b_{16}$. Similarly, shuffling is performed on each 32-bit word of the entire padded message.

4.1.1 Rationale

The shuffling function is simple to keep the overall algorithm fast, and has statistical effect (random properties) on the output. Shuffling of bits helps to improve the diffusion property of the input (*diffusion refers to the property that redundancy in the statistics of the input is "dissipated" in the statistics of the output*). The inverse of the shuffle operation can easily be accomplished by performing swaps in the reverse order. Therefore, the shuffling operation is bijective.

4.2 T-function

A T-function is a bijective mapping that updates every bit of the state, which can be described as $y_i = x_i+f(x_0,...,x_{i-1})$. In simple words, it is an update function in which every bit of the state is updated by a linear combination of the same bit and a function of a subset of its less significant bits.

If every single less significant bit is included in the update of every bit in the state, such a T- function is called Triangular. All the Boolean operations and most of the numeric operations in modern processors are T-functions, and all their compositions are also T-functions.

The T-function helps to achieve Avalanche effect. The Avalanche effect tries to mathematically abstract the much desirable property of high non-linearity between the input and output bits, and specifies that the hashes of messages from a close neighborhood are dispersed over the entire space. As T-functions are bijective, there are no collisions, and hence no entropy loss, regardless of the Boolean functions and the selection of inputs. After shuffling, the T-function takes the shuffled output and performs the Avalanche effect.

4.2.1 Rationale

The T-function used in Khichidi-1 hash algorithm is a quadratic function, $T(x) = (2x^2 + x) \bmod 2^{32}$, where x is a 32-bit word. The given T-function is an invertible mapping [1], which contains all the 2^n possible states on a single cycle for any word size n (where n = 32). This T-function T(x) maintains the Avalanche effect. Khichidi-1 hash algorithm controls a fixed point mapping (T(x) = x for some x) of the T-function

4.3 Linear Feedback Shift Registers

A degree-n polynomial over $GF(2^n)$ is primitive if and only if it has $2^n - 1$ periods. The period of a primitive polynomial of degree n is the maximum period $(2^n - 1)$ realized by its corresponding Linear Feedback Shift Register (LFSR) implementation [14]. Primitive polynomials over $GF(2^n)$ are useful in the design of Linear Feedback Shift Registers (LFSRs) to generate sequences for a maximum period. More taps are used in primitive polynomials in order to increase uncertainty on the outcomes of LFSRs. In our hash algorithm, a regularly clocked LFSR is used. Each time the 32-bit input is executed for different number of rounds; thus, even if the input bits are identical, the output will be random. This is known as irregularly clocked LFSR over GF(2^n).

4.3.1 Rationale

The primitive polynomial used is $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12}+ x^{11} + x^{10} + x^8 + x^7 +x^5 + x^4 + x^2 + x +1$. This is an irreducible polynomial of degree 32 whose period is 2^{32}-1. The taps of LFSR are used in random so that LFSR outputs could sustain the differential attack.

4.3.2 Pseudo code for LFSR:

```
1. Input: $f(x) = x^n + x^{s1}+ x^{s2}+ ... + x + 1$
2. Write: $x^n = x^{s1}+ x^{s2}+ ... + x + 1$ and express in binary form.
3. Compute one left shift for the entire binary value of $x^n$.
        3.1 P1: Left shift for all bits except nth position bit
        3.2 P2: Left shift on every nonzero nth position bit = binary expression of $x^n$
        3.3 Compute: $x^{n+1} = P1 + P2 \pmod 2$
```

In the above LFSR method, every bit in the binary expression of x^n should be shifted to the left, except the nth position bit. For the nth position bit, the same binary expression of x^n should be considered. On adding these two binary expressions with respect to mod 2, x^{n+1} can be obtained. The LFSR takes the input from the output of the T-function and produces the output containing a maximal period cycle.

5. PREPROCESSING

Preprocessing takes place before hash computation begins. It consists of three steps:

- Dividing a message into m-bit blocks
- Padding process takes place if number of bits of a message are less than m-bits for the last block
- Setting up the initial hash values to zero.

5.1 Dividing the Message

Message is divided into N m-bit blocks before the hash computation could begin.

5.1.1 Khichidi-1 (224-bits)

For Khichidi-1 (224-bits), the message is divided into N 224-bit blocks, $M^{(1)}$, $M^{(2)}$,..., $M^{(N)}$. Since the 224 bits of the input block may be expressed as seven 32-bit words, the first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on, up to $M_6^{(i)}$.

5.1.2 Khichidi-1 (256-bits)

For Khichidi-1 (256-bits) the message is divided into N 256-bit blocks, $M^{(1)}$, $M^{(2)}$,..., $M^{(N)}$. Since the 256 bits of the input block may be expressed as eight 32-bit words, the first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on, up to $M_7^{(i)}$.

5.1.3 Khichidi-1 (384-bits)

For Khichidi-1 (384-bits) the message is divided into N 384-bit blocks, $M^{(1)}$, $M^{(2)}$,..., $M^{(N)}$. Since the 384 bits of the input block may be expressed as twelve 32-bit words, the first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on, up to $M_{11}^{(i)}$.

5.1.4 Khichidi-1 (512-bits)

For Khichidi-1 (512-bits) the message is divided into N 512-bit blocks, $M^{(1)}$, $M^{(2)}$,..., $M^{(N)}$. Since the 512 bits of the input block may be expressed as sixteen 32-bit words, the first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on, up to $M_{15}^{(i)}$.

5.2 Padding the Message

The last block of a given message, M, is padded before the hash computation begins. The purpose of this padding is to ensure that the padded message is a multiple of 224, 256, 384 or 512 bits, depending on the algorithm.

5.2.1 Khichidi-1 (224-bits)

Consider that l (bits) is the length of the message M (expressed in 224 bits). Append the bit '1' at the end of the given message, followed by k zero bits, where k is the smallest, non-negative solution to the equation $l + 1 + k \equiv 0 \bmod 448$. For example, the (8-bit ASCII) message **abc** has length 8 X 3 = 24 bits; the message is padded with the bit '1', then by 224 – (24+1) = 199 zero bits, and then by the message length, to obtain the 448-bit padded message.

Block 1					Block 2	
01100001	01100010	01100011	1	00…00	00…00	00011000
"a"	**"b"**	**"c"**		**199 zeros**	**216 zeros**	**"*l*"**

The length of the padded message should now be 448 bits, a multiple of 224 bits.
For an empty file (data length = 0), Khichidi-1 algorithm considers 1 followed by 447 zeros in the padding process.

5.2.2 Khichidi-1 (256-bits)

Consider that l (bits) is the length of the message M (expressed in 256 bits). Append the bit '1' at the end of the given message, followed by k zero bits, where k is the smallest, non-negative solution to the equation $l + 1 + k \equiv 0 \mod 512$. The length of the padded message should now be 512-bits, a multiple of 256 bits.

5.2.3 Khichidi-1 (384-bits)

Consider that l (bits) is the length of the message M (expressed in 384 bits). Append the bit '1' at the end of the given message, followed by k zero bits, where k is the smallest, non-negative solution to the equation $l + 1 + k \equiv 0 \mod 768$. The length of the padded message should now be 768-bits, a multiple of 384 bits.

5.2.4 Khichidi-1 (512-bits)

Consider that l (bits) is the length of the message M (expressed in 512 bits). Append the bit '1' at the end of the given message, followed by k zero bits, where k is the smallest, non-negative solution to the equation $l + 1 + k \equiv 0 \mod 1024$. The length of the padded message should now be 1024-bits, a multiple of 512 bits.

5.3 Setting the Initial Hash Value

Before hash computation begins for each of the secure hash algorithms, the initial hash value, H(0), must be set to zero. The size and number of words in H(0) depends on the size of the Message Digest. *In order to avoid trivial pseudo collisions [2], we fix H(i) = 0 for every i.*

5.3.1 Khichidi-1 (224-bits)

For Khichidi-1 (224-bits), the initial hash value $H^{(0)}$ consists of the following eight 32-bit words:

That is 224/32 + 1 = 8 words.

$H_0^{(0)}$ = 0x00000000

$H_1^{(0)}$ = 0x00000000

$H_2^{(0)}$ = 0x00000000

$H_3^{(0)}$ = 0x00000000

$H_4^{(0)}$ = 0x00000000

$H_5^{(0)}$ = 0x00000000

$H_6^{(0)}$ = 0x00000000

$H_7^{(0)}$ = 0x00000000

5.3.2 Khichidi-1 (256-bits)

For Khichidi-1 (256-bits), the initial hash value $H^{(0)}$ consists of the following nine 32-bit words:

That is 256/32 + 1 = 9 words.

$H_0^{(0)}$ = 0x00000000

$H_1^{(0)}$ = 0x00000000

$H_2^{(0)}$ = 0x00000000

$H_3^{(0)}$ = 0x00000000

$H_4^{(0)}$ = 0x00000000

$H_5^{(0)}$ = 0x00000000

$H_6^{(0)}$ = 0x00000000

$H_7^{(0)}$ = 0x00000000

$H_8^{(0)}$ = 0x00000000

5.3.3 Khichidi-1 (384-bits)

For Khichidi-1 (384-bits), the initial hash value $H^{(0)}$ consists of the following thirteen 32-bit words:

That is 384/32 + 1 = 13 words.

$H_0^{(0)}$ = 0x00000000

$H_1^{(0)}$ = 0x00000000

$H_2^{(0)}$ = 0x00000000

$H_3^{(0)}$ = 0x00000000

$H_4^{(0)}$ = 0x00000000

$H_5^{(0)}$ = 0x00000000

$H_6^{(0)}$ = 0x00000000

$H_7^{(0)}$ = 0x00000000

$H_8^{(0)}$ = 0x00000000

$H_9^{(0)}$ = 0x00000000

$H_{10}^{(0)}$ = 0x00000000

$H_{11}^{(0)}$ = 0x00000000

$H_{12}^{(0)}$ = 0x00000000

5.3.4 Khichidi-1 (512-bits)

For Khichidi-1 (512-bits), the initial hash value $H^{(0)}$ consists of the following seventeen 32-bit words:

That is 512/32 + 1 = 17 words.

$H_0^{(0)}$ = 0x00000000

$H_1^{(0)}$ = 0x00000000

$H_2^{(0)}$ = 0x00000000

$H_3^{(0)}$ = 0x00000000

$H_4^{(0)}$ = 0x00000000

$H_5^{(0)}$ = 0x00000000

$H_6^{(0)}$ = 0x00000000

$H_7^{(0)}$ = 0x00000000

$H_8^{(0)}$ = 0x00000000

$H_9^{(0)}$ = 0x00000000

$H_{10}^{(0)}$ = 0x00000000

$H_{11}^{(0)}$ = 0x00000000

$H_{12}^{(0)}$ = 0x00000000

$H_{13}^{(0)}$ = 0x00000000

$H_{14}^{(0)}$ = 0x00000000

$H_{15}^{(0)}$ = 0x00000000

$H_{16}^{(0)}$ = 0x00000000

6. Khichidi-1 Algorithm

We have designed a new secure hash algorithm Khichidi-1, which is different from the existing hash algorithms. The new Khichidi-1 is a one-way hash function, which is a combination of message pre processing (bijective function) and Cipher Block Chaining (CBC) mode.

The algorithm has been designed in two stages, in a sequential manner: padding and hash computation. Message is divided into 'm' blocks of a fixed length (the padding is done for the last block if the length of the last block is less than messaged digest length) and setting up the initial hash values are set to zero ($H^{(i)} = 0$). Message preprocessing involves performing bijective operations such as Shuffling, T-function and Linear Feedback Shift Register (LFSR). These operations are used in the hash computation, which is performed on each input block of a fixed length in Cipher Block Chaining (CBC) mode, with six iterative rounds. The final value generated by the proposed hashing is used to determine the message digest / hash value.

Khichidi-1 has a final hash value of seven, eight, twelve or sixteen 32-bit words (224, 256, 384 or 512-bit message digest). Appendix A gives several detailed examples of Khichidi-1.

Note: In every round of Khichidi-1 algorithm, cnst = 0 (a constant value) is provided in the theoretical design (shown in Figures 2 & 3). This provision has been made to include symmetric keys in order to have Keyed hash functions. This constant value is not mentioned in the implementation.

6.1 Khichidi-1 (224-bits)

6.1.1 Khichidi-1 (224-bits) Preprocessing

To preprocess the Khichidi-1 (224-bits) algorithm:

1. Divide the padded message into N 224-bit message blocks, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$, by following the process mentioned in section 5.1.1
2. Pad the message, M, by following the process mentioned in section 5.2.1
3. Set the initial hash values $H^{(i)}$, as specified in section 5.3.1

6.1.2 Khichidi-1 (224-bits) Hash Computation

After preprocessing is completed, each message block, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$, is processed in a sequential order, by using the following steps:

```
Digest
for j = 0 to 6
{
        1. Perform Shuffling:
                $X_j^{(i)}$=Shuffling( $M_j^{(i)} \oplus H_j^{(i)}$)
        2. Pass it to the T-function
                $Y_j^{(i)}$= T-function( $X_j^{(i)}$)
        3. Pass this word to Linear Feedback Shift Register (LFSR)
                $Z_j^{(i)}$= LFSR( $Y_j^{(i)}$)
                $H_{j+1}^{(i)} = Z_j^{(i)}$
}
$H_0^{(i)} = H_7^{(i)}$
for round =1
{
        for i = 1 to N
        {
                Digest($M^{(i)}$)
        }
}
For round 2 to 6
{
        $M^x = Z_0^{(N)} || Z_1^{(N)} || Z_2^{(N)} || Z_3^{(N)} || Z_4^{(N)} || Z_5^{(N)} || Z_6^{(N)}$
        Digest($M^x$)
        $M^y =$     $Z_0^{(x)} \| Z_1^{(x)} \| Z_2^{(x)} \| Z_3^{(x)} \| Z_4^{(x)} \| Z_5^{(x)} \| Z_6^{(x)}$
        Digest ($M^y$)
}
```

Hash = $Z_0^{(y)} \| Z_1^{(y)} \| Z_2^{(y)} \| Z_3^{(y)} \| Z_4^{(y)} \| Z_5^{(y)} \| Z_6^{(y)}$

The resulting 224-bit message digest of the message, M, is

$$Z_0^{(y)} \| Z_1^{(y)} \| Z_2^{(y)} \| Z_3^{(y)} \| Z_4^{(y)} \| Z_5^{(y)} \| Z_6^{(y)}$$

6.2 Khichidi-1 (256-bits)

6.2.1 Khichidi-1 (256-bits) Preprocessing

To preprocess the Khichidi-1 (256-bits) algorithm:

1. Divide the padded message into N 256-bit message blocks, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$, by following the process mentioned in section 5.1.2
2. Pad the message, M, by following the process mentioned in section 5.2.2
3. Set the initial hash values $H^{(i)}$, as specified in section 5.3.2

6.1.2 Khichidi-1 (256-bits) Hash Computation

After preprocessing is completed, each message block, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$, is processed in a sequential order, by using the following steps:

Digest

```
for j = 0 to 7
{
        1. Perform Shuffling:
                $X_j^{(i)}$=Shuffling( $M_j^{(i)} \oplus H_j^{(i)}$)
        2. Pass it to the T-function
                $Y_j^{(i)}$ = T-function( $X_j^{(i)}$)
        3. Pass this word to Linear Feedback Shift Register (LFSR)
                $Z_j^{(i)} = \text{LFSR}( Y_j^{(i)})$
                $H_{j+1}^{(i)} = Z_j^{(i)}$
}
$H_0^{(i)} = H_8^{(i)}$
for round =1
{
        for i = 1 to N
        {
                Digest($M^{(i)}$)
        }
}
For round 2 to 6
{
        $M^x = Z_0^{(N)} || Z_1^{(N)} || Z_2^{(N)} || Z_3^{(N)} || Z_4^{(N)} || Z_5^{(N)} || Z_6^{(N)} || Z_7^{(N)}$
        Digest($M^x$)
        $M^y = Z_0^{(x)} || Z_1^{(x)} || Z_2^{(x)} || Z_3^{(x)} || Z_4^{(x)} || Z_5^{(x)} || Z_6^{(x)} || Z_7^{(x)}$
        Digest ($M^y$)
}

Hash = $Z_0^{(y)} || Z_1^{(y)} || Z_2^{(y)} || Z_3^{(y)} || Z_4^{(y)} || Z_5^{(y)} || Z_6^{(y)} || Z_7^{(y)}$
```

The resulting 256-bit message digest of the message, M, is
$Z_0^{(y)} || Z_1^{(y)} || Z_2^{(y)} || Z_3^{(y)} || Z_4^{(y)} || Z_5^{(y)} || Z_6^{(y)} || Z_7^{(y)}$

6.3 Khichidi-1 (384-bits)

6.3.1 Khichidi-1 (384-bits) Preprocessing

To preprocess the Khichidi-1 (384-bits) algorithm:

1. Divide the padded message into N 384-bit message blocks, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$, by following the process mentioned in section 5.1.3

2. Pad the message, M, by following the process mentioned in section 5.2.3
3. Set the initial hash values $H^{(i)}$, as specified in section 5.3.3

6.1.2 Khichidi-1 (384-bits) Hash Computation

After preprocessing is completed, each message block, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$, is processed in a sequential order, by using the following steps:

```
Digest
for j = 0 to 11
{
        1. Perform Shuffling:
                $X_j^{(i)}$=Shuffling( $M_j^{(i)} \oplus H_j^{(i)}$)
        2. Pass it to the T-function
                $Y_j^{(i)}$ = T-function( $X_j^{(i)}$)
        3. Pass this word to Linear Feedback Shift Register (LFSR)
                $Z_j^{(i)}$ = LFSR( $Y_j^{(i)}$)
                $H_{j+1}^{(i)} = Z_j^{(i)}$
}
$H_0^{(i)} = H_{12}^{(i)}$
for round =1
{
        for i = 1 to N
        {
                Digest($M^{(i)}$)
        }
}
For round 2 to 6
{
        $M^x = Z_0^{(N)} || Z_1^{(N)} || Z_2^{(N)} || Z_3^{(N)} || Z_4^{(N)} || Z_5^{(N)} || Z_6^{(N)} || Z_7^{(N)} || Z_8^{(N)} || Z_9^{(N)} || Z_{10}^{(N)} ||$
$Z_{11}^{(N)}$
        Digest($M^x$)
        $M^y = Z_0^{(x)} || Z_1^{(x)} || Z_2^{(x)} || Z_3^{(x)} || Z_4^{(x)} || Z_5^{(x)} || Z_6^{(x)} || Z_7^{(x)} || Z_8^{(x)} || Z_9^{(x)} || Z_{10}^{(x)} || Z_{11}^{(x)}$
        Digest ($M^y$)
}
```

Hash = $Z_0^{(y)} || Z_1^{(y)} || Z_2^{(y)} || Z_3^{(y)} || Z_4^{(y)} || Z_5^{(y)} || Z_6^{(y)} || Z_7^{(y)} || Z_8^{(y)} || Z_9^{(y)} || Z_{10}^{(y)} || Z_{11}^{(y)}$

The resulting 384-bit message digest of the message, M, is
$Z_0^{(y)} || Z_1^{(y)} || Z_2^{(y)} || Z_3^{(y)} || Z_4^{(y)} || Z_5^{(y)} || Z_6^{(y)} || Z_7^{(y)} || Z_8^{(y)} || Z_9^{(y)} || Z_{10}^{(y)} || Z_{11}^{(y)}$

6.4 Khichidi-1 (512-bits)

6.4.1 Khichidi-1 (512-bits) Preprocessing

To preprocess the Khichidi-1 (512-bits) algorithm:

1. Divide the padded message into N 512-bit message blocks, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$, by following the process mentioned in section 5.1.4
2. Pad the message, M, by following the process mentioned in section 5.2.4
3. Set the initial hash values $H^{(i)}$, as specified in section 5.3.4

6.1.2 Khichidi-1 (512-bits) Hash Computation

After preprocessing is completed, each message block, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$, is processed in a sequential order, by using the following steps:

Digest

```
for j = 0 to 15
{
        1. Perform Shuffling:
                $X_j^{(i)}$=Shuffling( $M_j^{(i)} \oplus H_j^{(i)}$)
        2. Pass it to the T-function
                $Y_j^{(i)}$ = T-function( $X_j^{(i)}$)
        3. Pass this word to Linear Feedback Shift Register (LFSR)
                $Z_j^{(i)}$ = LFSR( $Y_j^{(i)}$)
        $H_{j+1}^{(i)} = Z_j^{(i)}$
}
$H_0^{(i)} = H_{16}^{(i)}$
for round =1
{
    for i = 1 to N
    {
        Digest($M^{(i)}$)
    }
}
For round 2 to 6
{
    $M^x = Z_0^{(N)} || Z_1^{(N)} || Z_2^{(N)} || Z_3^{(N)} || Z_4^{(N)} || Z_5^{(N)} || Z_6^{(N)} || Z_7^{(N)} || Z_8^{(N)} || Z_9^{(N)} || Z_{10}^{(N)} || Z_{11}^{(N)} || Z_{12}^{(N)} || Z_{13}^{(N)} || Z_{14}^{(N)} || Z_{15}^{(N)}$
    Digest($M^x$)
    $M^y = Z_0^{(x)} || Z_1^{(x)} || Z_2^{(x)} || Z_3^{(x)} || Z_4^{(x)} || Z_5^{(x)} || Z_6^{(x)} || Z_7^{(x)} || Z_8^{(x)} || Z_9^{(x)} || Z_{10}^{(x)} || Z_{11}^{(x)} || Z_{12}^{(x)} || Z_{13}^{(x)} || Z_{14}^{(x)} || Z_{15}^{(x)}$
    Digest ($M^y$)
}
```

Hash = $Z_0^{(y)} || Z_1^{(y)} || Z_2^{(y)} || Z_3^{(y)} || Z_4^{(y)} || Z_5^{(y)} || Z_6^{(y)} || Z_7^{(y)} || Z_8^{(y)} || Z_9^{(y)} || Z_{10}^{(y)} || Z_{11}^{(y)} || Z_{12}^{(y)} || Z_{13}^{(y)} || Z_{14}^{(y)} || Z_{15}^{(y)}$

The resulting 512-bit message digest of the message, M, is

$Z_0^{(y)} || Z_1^{(y)} || Z_2^{(y)} || Z_3^{(y)} || Z_4^{(y)} || Z_5^{(y)} || Z_6^{(y)} || Z_7^{(y)} || Z_8^{(y)} || Z_9^{(y)} || Z_{10}^{(y)} || Z_{11}^{(y)} || Z_{12}^{(y)} || Z_{13}^{(y)} || Z_{14}^{(y)} || Z_{15}^{(y)}$

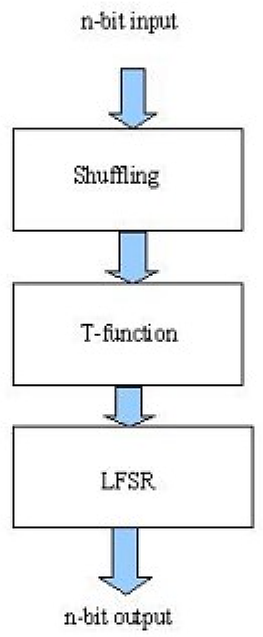
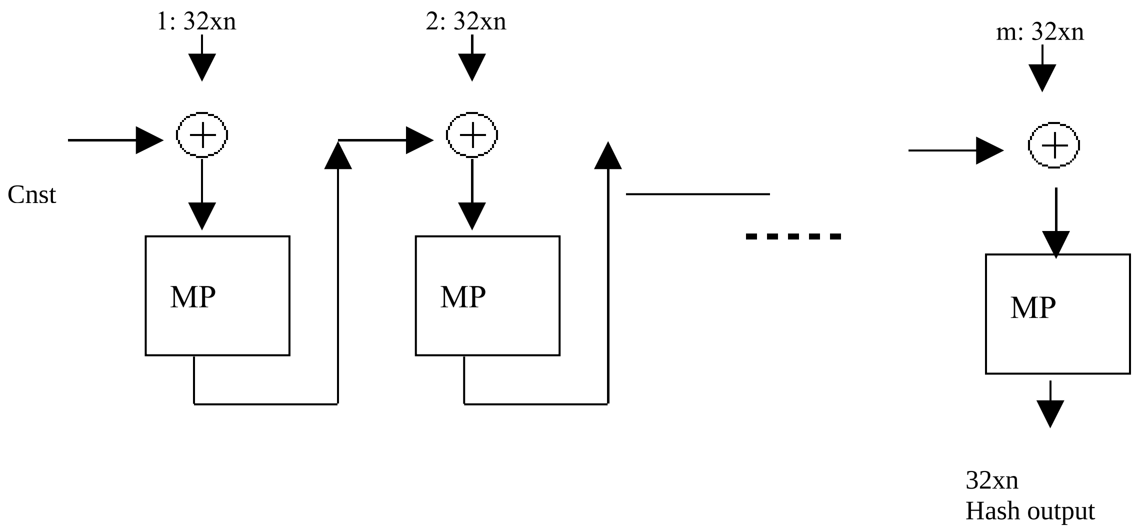
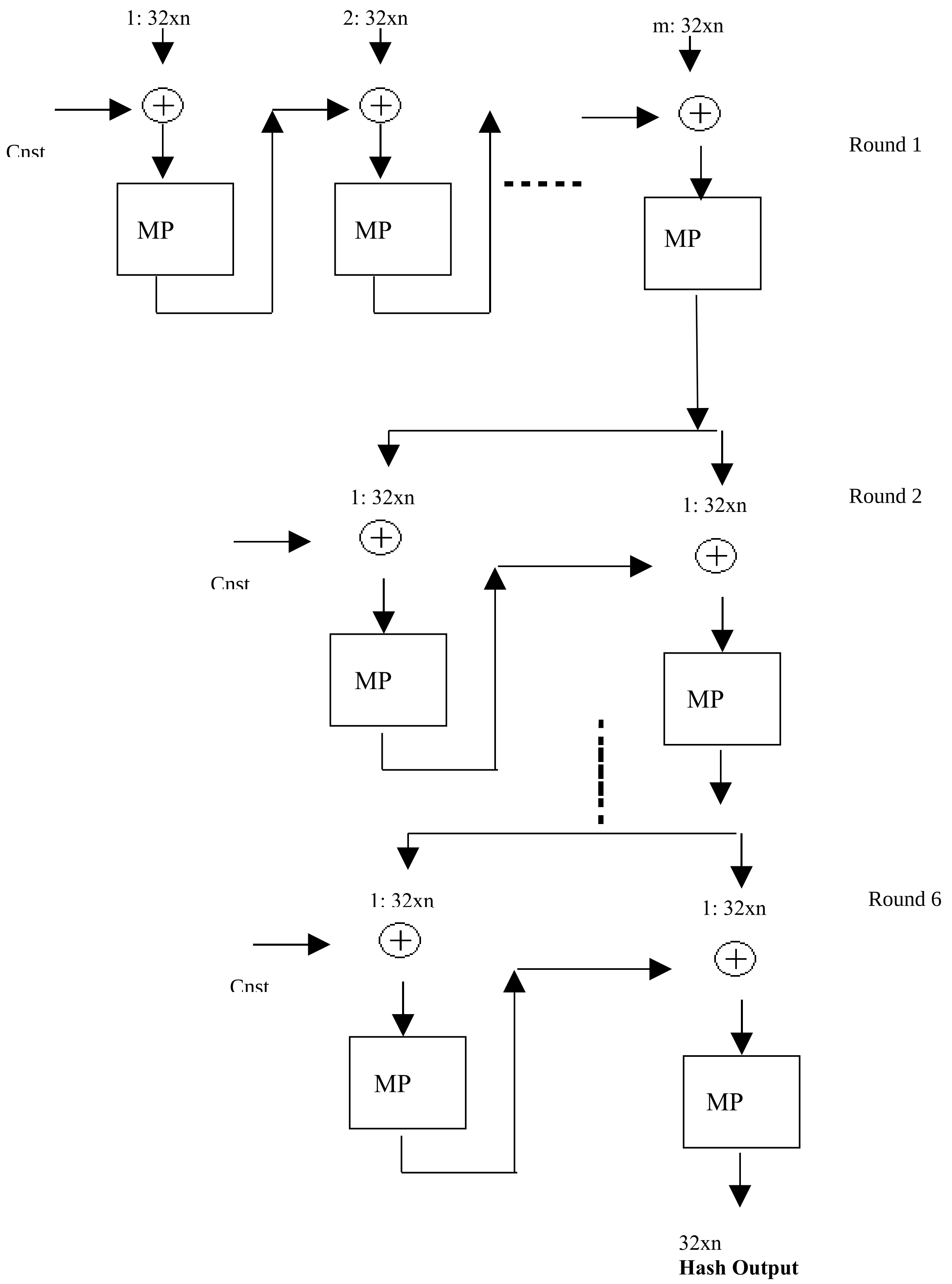


Figure 1: Message Prep rocessing (MP)



n = 7, 8 ,12 and 16 for 224, 256, 384 and 512-bit digests respectively and Cnst = 0 is a constant value

Figure 2: Khichidi-1 hash computation with one round



n = 7, 8, 12 and 16 for 224, 256, 384 and 512-bit digests respectively and Cnst = 0 is a constant value

Figure 3: Khichidi-1 Hash computation (6 rounds)

7. Analysis of Attacks on Khichidi-1

Padding Attack

Consider a pair: (M, MD), where MD is the Message Digest of a message M. Using Padding attack, a Cryptanalyst can develop a different pair (M′, MD′), where the difference between the messages M and M′ is just the padding. That is, just by changing the length of padding of a message, the same message digest can be obtained on claiming that the messages are different.

Khichidi-1 works as a random function where the Message Digest value is random. The output of each round will therefore be more random than the output of the previous rounds due to Avalanche effect. Thus, after the message is passed into this hash function six times, the final hash value is different even if the input values are almost the same. Since Khichidi-1 has a proper padding process before hashing, this padding attack can be avoided.

Example:

Khichidi-1 (224 bits)

Input 1 = 0
Data Length = 6 bits
Round 1 = EE14382903B2CFC60FEF9DCF501D37B5D734B1BDA8CA5E66712697A7
Round 2 = 519FA756C906A0E3636FD70375BA0A78ACB9258189EC06F59B2FE8EF
Round 3 = 27DA644D2DDF30288D4596EEDEFB5B8CC6F1BC76E797B4A50C22CDC8
Round 4 = F790947EDE989A3B78DA39BDBBDC4F9D277306251330FDF782FDF069
Round 5 = 513D037F2CF4D774E50CC216DA12435956162E03C655AB70D8B2BC31
Round 6 = 5A3E7CC5DEBD58562716D414BD1E75D14951D3487E8687778275831F
Message Digest= 5A3E7CC5DEBD58562716D414BD1E75D14951D3487E8687778275831F

Input 2 = 0
Data Length = 7 bits
Round 1 = C0D039B737876D0E47FEE4BB6977657EA2F3870C3739F34487A501D2

Round 2 = A9C08D2F3741D88D0C78638CD8C0B46C7D828013F21AEA42701BCAE4

Round 3 = 400570371C741ABF3E297CAEEDA0942B6CE1AB906CC21F9B386ED445

Round 4 = 8B90CBEB12700BD78AF7E9AA5ED2913A0366061B745EBDF7DF707E48

Round 5 = 3620E6DEA0BF968609A7F0170C55663DEF1057FB7B1802F017151B80

Round 6 = DF0BFEBAA68F7F57D75F31004DFED376DFC3C701BAB204FAB03FA7EC

Message Digest = DF0BFEBAA68F7F57D75F31004DFED376DFC3C701BAB204FAB03FA7EC

Khichidi-1 (256 bits)

Input 1 = 0

Date Length = 6 bits

Round 1 = 03B2CFC60FEF9DCF501D37B5D734B1BDA8CA5E669E1978CFB9CA67F250E11430

Round 2 = E2D05F974B58FF644E4590DD858B41A34618787D07717A9C257F427BAFA9E37E

Round 3 = 784B25E270CB6FFB6565AC0F39CC9AF054FC9C665EBFD65C58F0EF0EC079FAA3

Round 4 = DA4FA37C1C6EB1F9A73EFCFB6702C13E7A2A80D08EAC724326C68166EF9EDA6D

Round 5 = 017FCF152E78680341C4CE673A2DCAA1E629628D02017A7C81E8725BEB6DAB0F

Round 6 = D011BA6C980F7645237E442C8D63F61A53A62C5E4C1764DB96BAB8E443BCE0D5

Message Digest= D011BA6C980F7645237E442C8D63F61A53A62C5E4C1764DB96BAB8E443BCE0D5

Input 1 = 0

Data Length = 7 bits

Round 1 = 37876D0E47FEE4BB6977657EA2F3870C3739F3443B173F83A7768B550833B498

Round 2 = CA08ECDCC60FE69E5767FD00CDA4C8E01CCADC746BF282B212CBE9F9D18A7106

Round 3 = 7AB45CB6ED9BFB2488C95CD70C3AA3057B5D7606D4AE55B36FE28EEBFB0E9991

Round 4 = 885F973DD961DDC6DB4D42B10394430420F2599C52D7FA796FC0A16B23B83218

Round 5 = 3FF66928310C58BBD21E8D1C3559B490843965F3144F25AF1D10BE04CAB237A0

Round 6 = 56EC5028C50F27D9A76D59CAA68B3308C8B6B33E3631F69D38A0AD1D35439FC1

Message Digest= 56EC5028C50F27D9A76D59CAA68B3308C8B6B33E3631F69D38A0AD1D35439FC1

Khichidi-1 (384 bits)

Input 1 = 0

Data Length = 6

Round 1=

A8CA5E669E1978CFB9CA67F2D7E4FD5894261F77B78CA129AA535D7139604A4DAC4A97269ABB
457851F56D24191612DC

Round 2=

AA66916F4F6D00E460F8BE8C888C16D72B66A7A7EEE6D84FAA6655CB9D97A50F870B1F5D9558
90A6DEE3A26BD8A4D5B2

Round 3=
465A20F4A596A1E43F65DF67F0EFB314EE48A265B2941DAF6A20053004C1A3023E92C6DE07CE01
2773B744F4BA87124C
Round 4=
F42333F55319316A645831660730F2DB23AAFDE8563EEE292DAE2DA1D84C4A552E4F037E635100
AABBB343389DB8E260
Round 5=
8825D76A64FFA540C0B1893BE01CE8149BC0DFF0B96BFDA18C2A4FE302B372B88AF42FA87E61
BB6BBB433D09CCBB517A
Round 6=
AFD8EC9BB27936D56BAE3A5FB4D0213EC0A452E845665D0D7244119FA48B37353DBFF85A2E8F
0596C6190879BE7252BE
Message Digest=
AFD8EC9BB27936D56BAE3A5FB4D0213EC0A452E845665D0D7244119FA48B37353DBFF85A2E8F
0596C6190879BE7252BE

Input 1 = 0
Data Length = 7
Round 1 =
3739F3443B173F83A7768B5571A2DC8196EDD156C6A4B54B75FA8D99672B7B6181C106FA488B3
AAB70611A953FA0829C
Round 2 =
B4378925BCAC1E689E43BAC1D2A9EDA3F0392A51F5FECFEBCF42C33FC705728B0CD023463B94
76AD55F1E4D466AED523
Round 3 =
F38204703625816783F590B8307EEE99DE558DD31B9DA924AB922C62CA25EFBAC80339F45DF761
BA5D51848FB15F0E84
Round 4 =
39EC699D55495DABED8D78AE8A2BE8F55AA3D80F5BB93B7EBEBE4A4464AF5C7B9F4B6679547
87319AB9A22BC6E8F30D4
Round 5 =
3D1FA01A8973FD214299CE1E15B15489843F4DCFEA4E0C96636D564803F549097A73B850517B38C
D80EF322AEA4588E8
Round 6 =
2D450F79DB049FD089B8AACB01EC399CBECD51A2EF8324C436928F479AE9E90C264A40B0BDD
80FFD14A701350154F676

Message Digest=
2D450F79DB049FD089B8AACB01EC399CBECD51A2EF8324C436928F479AE9E90C264A40B0BDD
80FFD14A701350154F676

Khichidi-1 (512 bits)

Input 1 = 0
Data Length = 6
Round 1 =
94261F77B78CA129AA535D7139604A4DAC4A97269ABB457851F56D2478776374E051FEB36AE356
D2546BF89BD2AEA9222A256798562EB8D4265799C71394313A
Round 2 =
BFA1C55CB8A975BDC9BC4EB059ECAED168493C9E3F62487806987F849AD7CEE483E9C49E1B3B
E707C39AAAA0D1D1E888264E6640ED21562013D7669E3FCC12C4
Round 3 =
3D16E7F66D2EE2627E15D5EFF889668B1CC3D311AD8E1EAAE09F208FA75BFA59779E4CE9B966
CFCB19BF7C999A7F84216ABB0C2A2B6C4FD83F3C06FB34F6A78E
Round 4 =
73A8D8652AAD3A9E5ED826B1DFE9EB1CA99D3176768895441296C8D5C44F54F4EF507A4A31A8
6321882B73D36065FF3F19D6AB817AE45B6135DC7FC84DBAAD78
Round 5 =
9DF70E4AB2BC9030A67421464A74F25C3B98E3ABAB518D8E66FA1B62CAE653FC596FF6B871F0
CDBA7F52817AD70C5ADAFF1CB03E816BE9390E79833A12EB7E9B
Round 6 =
1A5AE9F2A3F8132F5311FD71FDF45A3D74C6701F00A8B68D27437E69C9750EBDD89B02F6581FC
5F39E90E6EAF50D9F40F8ED6F8B3621952F818C27A6E7303F5A
Message Digest =
1A5AE9F2A3F8132F5311FD71FDF45A3D74C6701F00A8B68D27437E69C9750EBDD89B02F6581FC
5F39E90E6EAF50D9F40F8ED6F8B3621952F818C27A6E7303F5A

Input 1 = 0
Data Length = 7
Round 1 =
96EDD156C6A4B54B75FA8D99672B7B6181C106FA488B3AAB70611A95D7AA327D339CC3767CD7
941F5B8D2B02A208AFDCC10EDA171CC904290E0AFBAA41D394C1
Round 2 =
444BB1DA9155A98DD7C38004FAC8A1A6496579035B033F673B68C1EF5884245867588BE844A53D
82D19D9100ACF33C43644396745E4656C1CC01BD2E01DB63DE

Round 3 =
E4E36723BC9DD08FD6CF1EC9221398B563CA2B0F2EA6F5942DD9617E1DE072FC5069DB2027BB
ED8816DB39D07D8F0F6E06699008CB534D0D3EC2A309E070AEC5
Round 4 =
2826F0C33CED3C4490414C7607EE2B09818C271F08DC623E3DCF32540C6FB16480E45170B7C2776
45DD4D183A6436B1A3517B8EF2694613F5495A6B88A4C9C3D
Round 5 =
81463240795D03C417C01388AAFE8F969227EA61CA0EB5754EB6A82ADE7C913CFED9D41711C32
B15FC2E928C6D5E08FC80518A652CC528EB8BB543D944382A34
Round 6 =
19E6EC871D05B9154426343D423F065949515237AC3103DBAE15FF3AC4E42D0B39DCDAF48D791
B8DE0CC113A08EE6DCF8A6B32ECDCDE7F9E273C3C42DA0274A8
Message Digest =
19E6EC871D05B9154426343D423F065949515237AC3103DBAE15FF3AC4E42D0B39DCDAF48D791
B8DE0CC113A08EE6DCF8A6B32ECDCDE7F9E273C3C42DA0274A8

Message Expansion Attack

Message Expansion Attack is a well known generic weakness of the iterative hash construction. This is also known as Length Extension Attack. Consider a message M is split into r -blocks such that, $M = m_1, m_2 ,.., m_r$.

An attacker can choose a message M′ such that, $M' = m_1, m_2 ,\ldots, m_r, m_{r+1}$. Since the first r-blocks of both the messages are equal, the chaining values produced by these messages by using the iterative construction will also be the same.

This method is not feasible in Khichidi-1 because MP core operations depend on the total input bits (the output changes significantly by flipping a single input bit due to Avalanche effect). A change in one bit will result in an entirely different hash value. It is clear that Khichidi-1 uses T-function which has a property of strict Avalanche criteria; the hash values of M and M’ are therefore different.

Example:

Khichidi-1 (224 bits)

Input = AaBbCcDdEeFfGgHhIiJjKkLlMmNn

Data Length = 224

Round 1 = F1E93BFD6162EB4A92B3CAD807796193BEA64665D1AF4B3646EE578C

Round 2 = 51317DE622182B44E773CA441B258C5EF249C422AE6B4370ABFD45F4

Round 3 = D417C4C52A4A4A3AFC7A5B3CB304FBDB1CFC08EA2ECF8E5BBC19D3DE

Round 4 = CFA3941C59B5E50B107C5D8234F6958679ED1F071A5E760788096DBA

Round 5 = DF0F45C4ADC75D69C4E46B228EAC08879EB5DDA3F98D0BC45C2F009F

Round 6 = 0E6E649667B27845D06C8B299ED3AB5F4B229BF02E6AC804644CBD25

Message Digest = 0E6E649667B27845D06C8B299ED3AB5F4B229BF02E6AC804644CBD25

Input = AaBbCcDdEeFfGgHhIiJjKkLlMmNnO

Data Length = 232

Round 1 = A22E2AF29A326998E69EBA2778A3B62911508DCDE2EFDB264A417FBB

Round 2 = DBF41071B70028A13D12700FE0FEC3BCFF402196AE55735315733CC0

Round 3 = 0651D3E852AF5AAA68BFE20AED9D5C0BFAAE373DCE5B827BC82C1C59

Round 4 = DE8B2696727343577F56EAD1C0CEF71B686BDE686F95CE6E515456C5

Round 5 = 241C09FA194ACFED7CEE13E0F40C915943C3F28D8B57E6493A85E2F3

Round 6 = 4BE5F9E68C544E62CA088758315811466694B322C14C2A227B249A9D

Message Digest = 4BE5F9E68C544E62CA088758315811466694B322C14C2A227B249A9D

Khichidi-1 (256 bits)

Input = AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPp

Data Length = 256

Round 1 = E22CD84D664960D1C564872A2B18750889955837326058 05CDE594A6294645FC

Round 2 = BB2E3BA0BC6D928D9E50CD72542F63C8BEF3CF6921B18B44D19FC08855FAB183

Round 3 = 02D91B6397A94F28F518EC1C9867DB0481D3A466B6DADA588B7819E3DAF734A7

Round 4 = 9443A090012E91279381F07E08230CAFA6220686F346885FDAAEA0F02BF76EC5

Round 5 = 165A6C63089C85D7014242EAD1DDB5CACFD06684C3BC05C9B1934461767AFC60

Round 6 = C26F8398EF1231C25C368C8046BA88AC4281CA90FDFCE4EA048C13D52891C936

Message Digest = C26F8398EF1231C25C368C8046BA88AC4281CA90FDFCE4EA048C13D52891C936

Input = AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQ

Data Length = 264

Round 1 = 993625F9917A3EA1D5C4F256A270311FEB91E239F04BFCAA306720AC9017D0B3

Round 2 = 512EE73953114B77D031F2DD1A9FE09C5446DD81C6DA2B92D25F3CCBE5D4AB80

Round 3 = 19D8A8C83DDD35F2992EFE79F52191CA3A166CD5ED18E14EDBAEEB6D773B50C1

Round 4 = 175E0900AF72DD2C670268E68E44574484729F5644B219274CAF9208C3DBDA71

Round 5 = EF68241246983E4F4DCBB4D6BF0FD1DB4E45608A50CF995A7D359248336549A1

Round 6 = BB2951566EDF791D2EC30999448C183FB1D314CA7BDA0AF7685B71B58B396870

Message Digest = BB2951566EDF791D2EC30999448C183FB1D314CA7BDA0AF7685B71B58B396870

Khichidi-1 (384 bits)

Input = AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXx

Data Length = 384

Round 1 =

F99A358D79F4E8C123E6C8F1C8CE3C960910D60FB9B1C58E8B27F58BAA180CD5D54F1A11B5537
98D4775F61AFB4994EB

Round 2 =

652DEC612A5828A30E6C2598B33A94F8DCE6845EB571F1E4A294511ECD1F552EDF5F4A3932BAF
2D7D33527ED5769921F

Round 3 =

BDBCF2D3F3BE8A40CE76F79099FD79C0C8AE59CCC17798D309A1B21A33E77070D91C4B868E37
5496FCFC128939A3DE4B

Round 4 =

9CC6002E5C88E65F294973A97E59A27121EF735A1957DB865A99F7F0CBAD0E509B5E4DBC3E69E
3ABA9335F6BCC930C00

Round 5 =

A459C0038ED87FB48E1C27FD46C89643F720DDBAE1707784EC391834EB2F4DD7C45B46E52FED7
8F8D0F5CF49C46ADD16

Round 6 =

D81A16AF3CC21A8965D5C070D37756A56D522B0223DE81CF4A534ED58496243FF3D453EB1DF59
BD25DC4F4EEC8705967

Message Digest =

D81A16AF3CC21A8965D5C070D37756A56D522B0223DE81CF4A534ED58496243FF3D453EB1DF59
BD25DC4F4EEC8705967

Input = AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxY

Data Length = 392

Round 1 =

897A510098C8EBA7833AC40E74CBC9F8D5C5E493A16C12121F250A3F99004D85612199118ED3F7
9D585CCDAD0F457B51

Round 2 =
BBB5C5A80FE52285AE753262793C1E183AEB705208FB2653298BAEF0CD3B472D43DE28B02FF80
48A0DF639F910074F93
Round 3 =
8CB7C78ECB90713E0C974DF48A1D604484E7330FD1182BE0C27EEA1331B289CBB4972A99223374
55BF94D3F47E468B42
Round 4 =
09AFFAECDE9F606C6E3EC380EB632773CFA27357374FA5E92BD3C6ABEF0F25A3C7615A7F2CD7
FB6F73F98195611F1982
Round 5 =
78533352DFCD0E285F0F8F936F6E8940FBBDEB4EF03084E9148DF32C785D1A69A776EC1FB65D0
A145814FCF1D9905F15
Round 6 =
926D1225D60CAA910DB71E9B57217A52F6001657982C6CE833F42883E79043AB506FD6C709F8AD
0926DB1677F3CE642C
Message Digest =
926D1225D60CAA910DB71E9B57217A52F6001657982C6CE833F42883E79043AB506FD6C709F8AD
0926DB1677F3CE642C

Khichidi-1 (512 bits)

Input = (AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz0123456789)
Data Length = 512
Round 1 =
CECFE605A3037E6D804497AD9535ADD12476516A2A456EAFAD4C9B85EF5AB75146B5E456BB9
BEE4B9E512E9568368E7DA2C25BB696E92BCFC0B903F24C1FBA2B
Round 2 =
ABBF5DBCEB2639F1DB5CB60759C782BCD4DB500BF1FE1E70615DFFF6B797BB35BA90870992D
F419B373EE2388F0DE9A6E14B4B53BD83D6075E0A64F642E6A03D
Round 3 =
11E27F3E43C92B793C05C5CB7ACF7F5E3FE27E51D3FECFF1CF54DF429DF5CCF5CEF00843B964
DB64729180B0956CFC1FA1D69984270A4C12682FB1976488EE1C
Round 4 =
46BC2A470C88887CBA693305E42E09C9D9ACCE779F5890784AD4D808B83B43FD0E66607BF5BC6
C69B8A330EAF793C5DA17A9EC7C3ECD1F7B002E6C07D200747E

Round 5 =
67490A754F7D9B1D32F196C0D3AC54BA70029A91E54494D6E8C2E95F423F959CD6983CC5F49E41
57817B3F3DB290B40E7CAB279C483C8F29D80BD0F87C2F1AFB
Round 6 =
37B350233FA7918FE41AA91E3EB04D70163330E1F245248209B2FAED6B42DFDD98F165277F0B53
538162B75FB2F00F562D415CAC8659BA0A87088DEFA4038AB3
Message Digest =
37B350233FA7918FE41AA91E3EB04D70163330E1F245248209B2FAED6B42DFDD98F165277F0B53
538162B75FB2F00F562D415CAC8659BA0A87088DEFA4038AB3

Input = (AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPpQqRrSsTtUuVvWwXxYyZz0123456789)=
Data Length = 520
Round 1 =
748EF48FD7D36A1E45C1BD7B5B855152BD5768DCFF012233C3D122D2F14E6B2BAFB4577E19E56
C9F6AFC3585BE1C0EE694EE67A8799E22D0E887205038AE2264
Round 2 =
214C525F82AF1E3A567EF22158F246A2BBB419CBB70A59633B3464FA5A56B4564713B427DD9E5
A67B9CC67EFED8896865D50D48F2849A9A9F66A39A13C3385E6
Round 3 =
42C664DF8D63EACAF7C4BF9249EDEAE74E374303A095B75463D44C3CC3E502D839730B1ECB7E
455DDAAA5DEF2E966BD6B681CAA1CBDC9C373A686EB83F1339CC
Round 4 =
D5C90C35FC66FEDE4D38B40720BF5E545242BBB94C34695AD70CFE68D749B3140063426F463910
6BE2A033739A5BBF36A94B005EF2776FBA0E8EBE6C611B8754
Round 5 =
CC21886EC85175E4463B35D2EB33AAA850365E2ACB936B10264AB76450AEBBA4D683A3928F33
F361F043449B581F759198508ACE871BE2B3D34B2C32AA6C3268
Round 6 =
252143BFB67EE5596F366692829580EB6362EC4DAFECEA4A89A68938A9B9A7A076FAF158163008
1D9164424CE086B0AD51B97BFE6D826C76BAF5AE341DCD9503
Message Digest =
252143BFB67EE5596F366692829580EB6362EC4DAFECEA4A89A68938A9B9A7A076FAF158163008
1D9164424CE086B0AD51B97BFE6D826C76BAF5AE341DCD9503

Joux's Multi Collisions Attack

Antoine Joux found that finding multi-collisions attack on an iterative hash function is not harder than finding an ordinary collision. With the help of two pairs of hash collision messages, Joux attack could find multi-collisions. However, Khichidi-1 fails to have multi-collisions that could be avoided due to message preprocessing operations.

Suppose there exist two collision message (of same block size) pairs (a, b) and (c,d) such that Khichidi-1(a) = Khichidi-1(b) and Khichidi-1(c) = Khichidi-1(d). Then Khichidi-1 (a||c) = Khichidi-1(c$\oplus$ Khichidi-1 (a)) = Khichidi-1 (c$\oplus$ a') where a' = Khichidi-1 (a). Similarly, Khichidi-1(a||d) = Khichidi-1(d$\oplus$Khichidi-1(a)) = Khichidi-1(d$\oplus$a'). It follows that Khichidi-1(c$\oplus$ a') is not necessarily equal to Khichidi-1(d$\oplus$a') due to message preprocessing operations. This shows that Khichidi-1 fails to hold Joux's Multi Collisions Attack.

Fixed Point Attack

Finding second pre-image attacks for the iterative hash function is not too hard if the compression function of the hash function is such that computing fixed points is easy. Dean's attack accurately suits for designs based on Davies-Meyer block cipher construction, because it is easy to find fixed points in this type of block cipher construction [18]. His attack consists of the following two steps:

Step-1: Finding a particular number of fixed points denoted by 'A' and selecting one message block and computing chaining value denoted by 'B'.
Step-2: Once collision between a chaining value and a fixed point (that is, between chaining values in 'A' and in 'B') is found, the length extension attack is applied for adding blocks that cause the same chaining values as the original message.

Once such a message is found, it is easy to expand the number of blocks in the message to the appropriate length, by repeating the fixed points as many times as needed.

Our Khichidi-1 hash function is a layer of processes in which many parameters can be altered. In Khichidi-1, the number of shifts of LFSR can be incremented each time, which will prevent fixed point attack. LFSR which uses a 32 degree primitive polynomial, produces different non-zero 32-bit numbers (2^{32} -1 distinct numbers). Though a fixed point pair can be found, it produces different output. With the help of different random Initialization vectors, fixed point mappings are avoided in Khichidi-1.

For instance, $f(H_{i-1}, x_i) = H_{i-1}$ is a fixed point pair where H_{i-1} is the output of the previous round and x_i is the input to the current round. The fixed point pair produces different output when every time the number of shifts in LFSR can be altered. Therefore, it is computationally difficult to find fixed point attacks in Khichidi-1

Consider the following fixed point pairs on the input block x_i.

$f(H_{i-1}, x_i) = H1_{i-1}$ (1 shift of LFSR)
$f(H_{i-1}, x_i) = H2_{i-1}$ (2 shifts of LFSR)

.

.

.

$f(H_{i-1}, x_i) = H2^{g}_{i-1}$ (2^g shifts of LFSR, $g < 32$)

Repeating the same input block x_i will not result in the same hash value, due to maximal cycle property of LFSR. In the present design of Khichidi-1 algorithm, it has been considered that one shift LFSR operation is carried out in all hash rounds.

Herding Attack

Kelsey and Kohno [14] noticed that it is possible to perform a time-memory trade-off for several instances of pre-image attacks. An attacker using this attack can commit to a digital value available publicly, which corresponds to a meaningful message. After the announcement of the result, the attacker publishes a message that has the pre published digital value, and the correct information, along with a suffix. The main idea behind this attack is to start with a possible number of chaining values and select the digital value, which helps the attacker to perform a pre-image attack on the actual result obtained. Unlike Dean's attack, this attack can be applied to a shorter message as well.

In Khichidi-1, such an attack is not possible because Khichidi-1 algorithm avoids patterns or inner collisions during the round operations. Since the hash output is random, it is difficult to find pre-image attacks in Khichidi-1.

Example:

Khichidi-1 (224 bits)

Input = 0

Data Length = 8

Round 1 = 663E46C6119B9DB81A463D24FB3064C7310FCB0E4CE0EFBB213DFFD9

Round 2 = 60EFB75DD44F0C513E2EE97D89D996189318030F03AEA320A9ECF3CD

Round 3 = 347E84B8A155D2AA122FAA42A09050AFAD708900E2D286502FBCE501

Round 4 = B66384BB1FDE48962F9F2B34B8D37862EA1B7B2C03BDECB4C9E787CA

Round 5 = 2CAC5D2E78B5F2278B7BEC092A86EA597F2DCC0A2E105CE8E8B880DB

Round 6 = 55C6C4C66A384DFA4789C5A2BA190062254ECCC0A7FA0B674B9903B0

Message Digest = 55C6C4C66A384DFA4789C5A2BA190062254ECCC0A7FA0B674B9903B0

Input = 1

Data Length = 8

Round 1 = 3ECA6761F3E72E56BB081E8C7C5F25A03AA36D53ADB7F01581F04B5A

Round 2 = CA05D054EDED4FEFCDB1ADA51A23B581851FC248F2D72626BD39B384

Round 3 = 120A23AA29F9FB6D75CA3C3789642ABC1AE8BC73542D3C6DE53AF271

Round 4 = EC585EB2ED2220F21386F9A67DD10CD8787C185234CFD145DE94F574

Round 5 = 6C4686FA6765FD771DFB8BFA0E7FCB5830CB421B552C9528956B5BA0

Round 6 = 6D4721FD529AF76F7026327D59AB337DC54BFE685CA516537D7246FE

Message Digest = 6D4721FD529AF76F7026327D59AB337DC54BFE685CA516537D7246FE

Khichidi-1 (256 bits)

Input = 0

Data Length = 8

Round 1 = 119B9DB81A463D24FB3064C7310FCB0E4CE0EFBB1516315947012B16AFF43B8B

Round 2 = 438FDD6B0F16B3F2F78D753697889B909E548A590F664110BA57113B0D91ABA4

Round 3 = 2CE01FCACDF2C7D02D964C3BFC5D76C937FC86A3DBE59DBE6EE36DF6853785EF

Round 4 = E5A96E3BA249C695C4B35E0C2EDCD5813EEE85FF18BACEFAC25D0B0343C5CF07

Round 5 = 8AC07FD6681BBDD24F110C5F549D58CBCEAC98C40B19028F79969957A46D4A13

Round 6 = AD2B2AFFEC18EE91BBD66D3C2008D3F5A1B98EF10B71C8E8DB1D6F023CF2B3B1

MD = AD2B2AFFEC18EE91BBD66D3C2008D3F5A1B98EF10B71C8E8DB1D6F023CF2B3B1

Input = 1

Data Length = 8

Round 1 = F3E72E56BB081E8C7C5F25A03AA36D53ADB7F0153818896D8EED95911A87D0EA

Round 2 = AB5B3BFB50D2D8F58F6D2DBDED0A578FE474FFEB6F33CBA871A585A15DA95089

Round 3 = 1F79B422DCE06BF5A0193FD429466D955B28FF4F0EDC95F532A7E983843875AF

Round 4 = BFD0BA7A3DA3958E178EFC6F9EF7FE7F597442E45BE45F06E6CCD62DCF532888

Round 5 = B9B1D84BF6BBA041B448967C4F14BE55E45842069B3AA540F5A6D1B6090C95CB

Round 6 = 473A03CC4029EC4AD78A4C0B94B6A838CD039C3807E566231A08A59AA4F9BB46

Message Digest =

473A03CC4029EC4AD78A4C0B94B6A838CD039C3807E566231A08A59AA4F9BB46

Khichidi-1 (384 bits)

Input = 0

Data Length = 8

Round 1 =

4CE0EFBB1516315947012B16CD32B9BCC66457B0254AF7F7006A745EA8AE5A4F521FC7A9A7C7F
CBA500CAA5B400574AE

Round 2 =

2103151F164FAC4B62EA6901CD902F91DDA8DD4696519EE1F12630C26FA62C2614768A45F7F161
457C39997C28FBA63F

Round 3 =

9CCA27A45989FB8523D15AC48A2A627DC63B6F3E78348B690B5E4489D7C5CF3960EAED44E23C
FF4D6F4088DD5AC9059B

Round 4 =

0328DC245287A59DA1B376A146241503D8E6D05A465A35094E869040EF37E7CD9A45075D19140F
14F9CD07C6F64CF266

Round 5 =

8D829A1C4BED8EDDBA8270F00E67BC491B513AD2144939225098026CA343823AA3C4829E52C44
CC81D31D08BCFFC753D

Round 6 =
3DA353F2643BCFA970CA11AADA298D12C0A7F500D0BFA798D6BF7F0493447F9FB92F000625106
E7B44128A145A5E17A1
Message Digest =
3DA353F2643BCFA970CA11AADA298D12C0A7F500D0BFA798D6BF7F0493447F9FB92F000625106
E7B44128A145A5E17A1

Input = 1
Data Length = 8
Round 1 =
ADB7F0153818896D8EED9591A6B44A6A4A5799084FE3946417AF5F7366C6AC55BDB1C08156D99
99D462B474A3895EC2C
Round 2 =
6A72562E2A7A67BE3F0CC4B5EDF729394A4D820640F49A2F4CA9A3F34E6F503BB844D6185982A
A0AB1D0A4A3045A9E32
Round 3 =
86EC12B48E4B66AF12226FEC84B44ABD46E69252EE55BFC117B7E388E5ACFB6EDF97ECDF0880
0ABDB2EC94E851A18D18
Round 4 =
C06B81AC6803506EEA38109C7212BE460EC5FA60C541BC06DEF6988F38205FE109E63ABB070F88
B8D5425E13F4836F95
Round 5 =
7B7D385E0490D850B45055A0BE2169CF29F38ECE34C48590889F8869D3140D64C600E869CAEB443
A6FC48BAA829D02B5
Round 6 =
AB44E6722908208808DC8D809EE0DF32FE7FDD44A79519C4C83F94C166512D2DF41F9483CBEAE
9C6DC64FAF0617936CA
Message Digest =
AB44E6722908208808DC8D809EE0DF32FE7FDD44A79519C4C83F94C166512D2DF41F9483CBEAE
9C6DC64FAF0617936CA

Khichidi-1 (512 bits)

Input = 0
Data Length = 8
Round 1 =
C66457B0254AF7F7006A745EA8AE5A4F521FC7A9A7C7FCBA500CAA5BC828FF2E715C57E8C73A
E0779601D85E17C35B8B331122225B8C312BB903C099270FC64D

Round 2 =

C232397D06D86DC8DF885FCF2F90A95013405F0D99CABD35C949D938256D940B1246516583A946
3F3D9200B19423EB1C1025C2F87D015A65A357FE071F04FDC8

Round 3 =

E7955E48DF665B544DA6E97E1696508F33D6A2981E0454D11639F35DBF9570A84A518BAD32B647
4884B535C67C967D18EE100013849005BC6266E362B3ACAF6F

Round 4 =

D979B23C20D6656425A2E37FC281CE4068A79BD3ECE6932C76283D8D0C1B620ACA26EA5EC079
F1DFCB5462C62A5946F878DCB452845A119FD94ADED103EE368F

Round 5 =

D462649D0BF449C6438E6EA1A2B7523A9AD88331E30C6501CB0049578EAF1D1A760C73832C8E6
C43E976DF411CFA6F0D1778CF1EEA40D1563CD31FE951E3FF4B

Round 6 =

BB4FB2F1B7AF3242FBB4D8D25AAE239305A83A9853487071210A1F6891DBC0DC0C52CB0224648
A46996EFCE9D9A0D0605CDF905FED3914BADF5BD1BECCD17FA6

Message Digest =

BB4FB2F1B7AF3242FBB4D8D25AAE239305A83A9853487071210A1F6891DBC0DC0C52CB0224648
A46996EFCE9D9A0D0605CDF905FED3914BADF5BD1BECCD17FA6

Input = 1

Data Length = 8

Round 1 =

4A5799084FE3946417AF5F7366C6AC55BDB1C08156D9999D462B474AB2C748AC9276EC337B3695
FE864BCF38545A5924BE092B87C9A0F8C5D40874A652A4DA5F

Round 2 =

7BBBC5F33E1C07B981C86E1B5BDF1F8F01582F8F1AF82F2B544EE04D60F7889DB3C1A18B4805D
CDA39EC3798003BA2471EB935103BE05D7D475339EE32BC3E8D

Round 3 =

C33B8A1B0922E4DB83680FA71BE82C64B912B6EE02120341242DD45725FB2FD827F578099F112E6
E4975494B040B1F47CAA8A6E98AC887203D5187CC058A506A

Round 4 =

940A8D87576E4A17145CE350E8D9C26095DBDE7B156AA01C2DFABCD574026C4A16C7DB5505D
CE07ED48824CA38271FAD7E2A3E0496446130B7567443681394C6

Round 5 =

CC99A432D15E62DA53FEDAD892585A64022887C246A95B4F3A25681B0295E73ADA8DE3DEE624
31A37A462D717A1276036180000E3BF23D28C618382EE7C56724

Round 6 =
6EA5A4B49017827B7CEA95D20155D7CFC2D6E08A089F1A6DC6A5DB7807B269FFE638B63A5711
AB717649E25373E180A4ACBD651CA5317804F080BE1CE89D2D63
Message Digest =
6EA5A4B49017827B7CEA95D20155D7CFC2D6E08A089F1A6DC6A5DB7807B269FFE638B63A5711
AB717649E25373E180A4ACBD651CA5317804F080BE1CE89D2D63

Differential Attack

Differential attack is the study of how difference in an input can affect the resultant difference in the output. Let x and x' be two messages such that $x' = x \oplus \Delta x$. Then their hash values are X = h(x) and Y = h(x'). Suppose the hamming distance between X and Y is less than the hamming distance between x and x', the internal hash values of X and Y, MP(x) and MP(x') could not be random. Then the entropy properties of MP(x) and MP(x') would be lower than x and x' respectively – contradicting to the design of Khichidi-1 hash algorithm. The hamming distance between X and Y is higher than the hamming distance between x and x' in the Khichidi-1 algorithm due to shuffling, avalanche and nonlinear properties. Since X and Y have good entropies due to the property of maximum hamming distance, it is difficult to compute Δx from X and Y. Also, there are no common message patterns in X and Y due to high entropy occurring in X||Y.

Example:

Let x = *fox* and x' = *box* be two text messages whose hamming distance differs by one character apart. Then compute the following hash values of x and x':

X= Khichidi-1-224(x) = 9315A63FD570C5ACCAC80E5694FA40CB4587B1E434FF11F316398157
Y= Khichidi-1-224(x') = FEBB7B6D63FA77C9FBDC3FEC3CD888BA966B65A9F351E64634D79389

X = Khichidi-1-256(x) =
C6B0227DE9E55466BD7DFC0023FA46CA6439F8571E83A61AEF5541D9FDF49BCA

Y = Khichidi-1-256(x’) =
7A23A0F38280D462CB5D14D960DAC5A5B84A400AD437419CF144739281B0C442

X = Khichidi-1-384(x) =
92A46AAAC94F2901BE987DC2E29A7301965958DCB8AA6C64A3892B9B250BC6AD9B43EC895938
C8EC6D95E8F62A5045FB

Y = Khichidi-1-384(x’) =
D0C9B2DFAF36993BCDFC4196114F328EF566ED3C3063E2B528514816C96B1BE13C0331FAA79D5
A56BF3E2555C01D3DA5

X = Khichidi-1-512(x) =
A8B60D598C829B5801B4BDD2D2DF3F6A8038F326F46AB7B2258BFB52FD7119DBE855711829A67
54AB28F85BED6F519868F47C5891A175E3B0B901FB673A390F9

Y = Khichidi-1-512(x’) =
61EC4681622987DCA0A3E1A98323A756E20681EF7A7E4CA90E60AE0A4C850788BA6446D73CA4
A6D391BC1AA20564DA51EA9A67CAFE1A46D8EA67BB470B420EFF

Hence the hamming distance (X,Y) is higher than (x,x’)

8. Security Reduction Proof

Khichidi-1 can be described as the message preprocessing (MP) function used in Cipher Block Chaining (CBC) mode. In this mode, all the blocks are chained together. The output of the first block $M^{(1)}$ is obtained from $H^{(1)} = \text{Digest}(M^{(1)} \oplus H^{(0)})$, where $H^{(0)}$ is the starting variable. In general, the output values of all input blocks are calculated as $H^{(i)} = \text{Digest}(M^{(i)} \oplus H^{(i-1)})$ where i =1, 2, . . . ,n and $M^{(1)}, M^{(2)},..., M^{(N)}$ are the input blocks of equal size. Then the final hash value becomes $H^{(n).}$

Following are the properties of the CBC mode:

- The chaining operation makes the output of a block dependent on all preceding input blocks, and therefore the output of the last block (final hash value) will depend on all the input blocks.
- With the help of internal hash values $H_j^{(i)}$, CBC round operation prevents the same input to result in the same Message Digest value.

The defined Khichidi-1 is robust, with respect to MP and CBC operations. Therefore the security reduction proof lies on MP and CBC operations. Khichidi-1 algorithm sets $H^{(i)} = 0$ for every i due to trivial pseudo collisions. Unless otherwise stated that Khichidi-1 has to select a suitable collection of random values to $H^{(i)}$.

Lemma 8.1: For any composition f of primitive functions, the mapping $x \rightarrow x + 2.f(x) \pmod{2^n}$ is invertible [1].

Theorem 8.2: Let m1 and m2 be two different messages. Then MP(m1) ≠ MP(m2)

Proof: Suppose $MP(m_1) = MP(m_2)$.Then computing Inverse LFSR function with respect to a primitive polynomial (used originally in the LFSR) on $MP(m_1)$ and $MP(m_2)$ is same. Since the primitive polynomial is used and has maximal length [14,21,22], it follows that LFSR has bijection. It can be shown that T-function (by Lemma 8.1), and De-shuffling have bijection so that the inverse of MP function maps to a unique element. Therefore, $m_1 = m_2$ – a contradiction; hence, the theorem.

Theorem 8.3: An n-bit hash Message Digest size of Khichidi-1 produces an n-bit output irrespective of the message length.

Proof: Suppose a given message ‘m’ is less than an n-bit hash Message Digest size, then the padding operation converts ‘m’ into an n-bit message. Our Khichidi-1 algorithms use MP and CBC operations on ‘m’. By theorem 8.2, MP is bijective so that MP+CBC rounds induce Khichidi-1 family of hash functions (224 / 256 / 384 /512-bits) to produce respective message digest output.

If $m \geq$ n-bit hash Message Digest size, then MP operations and CBC rounds cause Khichidi-1 to produce an n-bit Message Digest output. It follows that Khichidi-1 acts as an iterative compression function. Hence the result.

Theorem 8.4: Khichidi-1 hash algorithm is a one way function.

Proof: By theorem 8.3, Khichidi-1 hash algorithm produces a fixed length output irrespective of message length. Suppose there exists an inversion algorithm A for h, then $h(A(y)) = y$ where y is an n-bit message digest. This implies that h is bijective – a contradiction. Hence the theorem.

9. Comparison of SHA-2 and Khichidi-1

SHA-2-256 and Khichidi-1-256 are compared with the help of round operations. Both of them produce an output of 256 bits. SHA-2 has 64 rounds whereas our implementation of Khichidi-1 has six rounds. These algorithms are compared for the first four rounds. We have chosen two input values: Input 1 and Input 2 that differ in one byte. There are some similarities in the output result for input 1 and input 2 for the first three rounds. From the fourth round onwards, the values are entirely different for SHA-2 and Khichidi-1 (256 bits)

Let us take Input 1 = 8-bit ASCII string "0”and Input 2 =16-bit ASCII string "00"

Round operations in SHA-2-256	**Input 1**	**Input 2**
Round 1	2c13084d 6a09e667 bb67ae85 3c6ef372 c8d262a2 510e527f 9b05688c 1f83d9ab	2c3892cd 6a09e667 bb67ae85 3c6ef372 c8f7ed22 510e527f 9b05688c 1f83d9ab
Round 2	95daf8c 2c13084d 6a09e667 bb67ae85 d67c5e7e c8d262a2 510e527f 9b05688c	a1a3c7a3 2c3892cd 6a09e667 bb67ae85 74bca289 c8f7ed22 510e527f 9b05688c
Round 3	4d67678f 95daf8c 2c13084d 6a09e667 d82c569c d67c5e7e c8d262a2 510e527f	3fbfafcc a1a3c7a3 2c3892cd 6a09e667 789224fa 74bca289 c8f7ed22 510e527f
Round 4	42a85e98 4d67678f 95daf8c 2c13084d 2c9d8585 d82c569c d67c5e7e c8d262a2	2865ee54 3fbfafcc a1a3c7a3 2c3892cd d58325fe 789224fa 74bca289 c8f7ed22

Table 1: SHA-2-256

Round operations in Khichidi-1-256	**Input 1**	**Input 2**
Round 1	119B9DB8 1A463D24F B3064C73 10FCB0E4 CE0EFBB1 51631594 7012B16A FF43B8B	039D3FA7 FB05E8B2 682A90DC 75E15107 70E81872 EFB22B48 178A0590 C6A9BFA7
Round 2	438FDD6B 0F16B3F2 F78D7536 97889B90 9E548A59 0F664110 BA57113B 0D91ABA4	B15C6636 01882F83 78C97C69 84036A5F EF85671A 1BEA3ADB D16A6759 EAEA24CC
Round 3	2CE01FCA CDF2C7D0 2D964C3BF C5D76C93 7FC86A3D BE59DBE6 EE36DF6 853785EF	70EF9453 4921DDE4 2A4CA191 D646CFBB 1805344D F04A6A3B C388A5FF 5DF4A4BE
Round 4	E5A96E3B A249C695 C4B35E0C 2EDCD581 3EEE85FF 18BACEFA C25D0B03 43C5CF07	0E820B68 50D192F3 59E7E66B D4985FA3 4C3C7F7C 65C93E9E BB96559D 558D2709

Table 2: Khichidi-1-256

The performance of SHA-2 and Khichidi-1-256 is highlighted. Both algorithms are executed on an Windows XP, Intel Pentium 4 PC, 2.4GHz CPU with 512MB RAM

Size of file (Megabytes)	**Time taken (seconds) SHA-2-256**	**Time taken (seconds) Khichidi-1-256**
1.4	0.04	0.04
2.2	0.07	0.06
3.3	0.13	0.09
5.9	0.21	0.17
6.9	0.24	0.22

Table 3: Time Analysis of SHA-2-256 and Khichidi-1-256

The entropy test [8] is performed on SHA-2 -256 and Khichidi-1-256 algorithms.

Size of file (Megabytes)	**SHA-2-256 (bits per byte)**	**Khichidi-1-256 (bits per byte)**
1.4	3.805037	3.754988
2.2	3.840422	3.908771
6.9	3.751464	3.763552
66.3	3.885934	3.848738
66.7	3.777115	3.841328

Table 4: Entropy Analysis of SHA-2-256 and Khichidi-1-256

SHA-2-384 and Khichidi-1-384 are compared with the help of round operations. Both of them produce an output of 384 bits. SHA-2-384 has 80 rounds whereas our implementation of Khichidi-1-384 has six rounds. These algorithms are compared for the first four rounds. It is important to know that SHA-2-384 produces 512-bits during the internal rounds whereas Khichidi-1-384 produces 384-bits.

Let us take Input 1 = 24-bit ASCII string *"abc"* and Input 2 =896-bit ASCII string *"abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklmnopjklmnopqklmnopqrlmnopqrsmnopqrstnopqrstu"*

Round operations in SHA-2-384	**Input 1**	**Input 2**
Round 1	470994ad 30873f88 cbbb9d5d c1059ed8 629a292a 367cd507 9159015a 3070dd17 bd03f724 be6075f9 67332667 ffc00b31 8eb44a87 68581511 db0c2e0d 64f98fa7	47099491 95eda6f0 cbbb9d5d c1059ed8 629a292a 367cd507 9159015a 3070dd17 bd03f709 23c6dd61 67332667 ffc00b31 8eb44a87 68581511 db0c2e0d 64f98fa7
Round 2	2e912303 06a12ae0 470994ad 30873f88 cbbb9d5d c1059ed8 629a292a 367cd507	78d3f8bc 03a38303 47099491 95eda6f0 cbbb9d5d c1059ed8 629a292a 367cd507

Round operations in SHA-2-384	Input 1	Input 2
	5e1b4e16 95372b9e bd03f724 be6075f9 67332667 ffc00b31 8eb44a87 68581511	ae067f07 1cd18a36 bd03f7092 3c6dd61 67332667 ffc00b31 8eb44a87 68581511
Round 3	eebe5d37 9be707ad 2e912303 06a12ae0 470994ad 30873f88 cbbb9d5d c1059ed8 54074a65 aef34336 5e1b4e16 95372b9e bd03f724 be6075f9 67332667 ffc00b31	ed59d30b eff95306 78d3f8bc 03a38303 47099491 95eda6f0 cbbb9d5d c1059ed8 c180c7a7 4ed5cf1f ae067f07 1cd18a36 bd03f709 23c6dd61 67332667 ffc00b31
Round 4	e3084831 53e15ad6 eebe5d37 9be707ad 2e912303 06a12ae0 470994ad 30873f88 086c5b2d 36a89178 54074a65 aef34336 5e1b4e16 95372b9e bd03f724 be6075f9	8e7fe2ab a3168f2b ed59d30b eff95306 78d3f8bc 03a38303 47099491 95eda6f0 d92d1966 7920b327 c180c7a7 4ed5cf1f ae067f07 1cd18a36 bd03f709 23c6dd61

Table 5: SHA-2-384

Round operations in Khichidi -1-384	Input 1	Input 2
Round 1	B7E6305C E55013F0 3353F3B7 C60FB8F5 11A6986D 16A3FDB6 69C7D84C 0DC54984 95D5E774 F6C18574 F80C2283 A180DF2E	0B0DD7F1 2B7C5FDD 27FDC6D1 59B21E6A 8FB5CB6F 3303AA32 0BE3209B 64887195 C93B53A6 5AF05EFB 50853D59 4355CE5D
Round 2	F1808E40 8466D3B7 4E4FDBE3 3183F5DF 19133D6B 2BF56891 0E7357C1 45CFB736 8B601758 245441B4 0ACC5FB9 C2CF6B4E	394362FC 45A6B63A A69FC695 033B5085 65CB8DAF 0FA7F626 22D23A66F CB560092 C380FA93 7138EFFC 88304B8E 3D44F36
Round 3	7F487C61 F56EA60F 02D3FC03 27DACAA0 2C6BD1E5 01C18070 C234CAE1 8B64AA42 AB08A2BF B47DD499 2B02E1A3 DA9E648A	DAD903A0 088B015B 18D4B728 29E6D25A 6790E80A 8F2E6584 CAF76188 7FF5C156 0F59A085 F6A1C86F 66047AE9 BBC8D239
Round 4	6967B6AE 2797F52A F18BFBC6 9CF68E0B 961C7F7C EA8AE543 DDF0AB98 16ABA308 2C9A4142 FE8176D5 3D6EE84C 0E623A68	D54B153A 9BFA15B1 57A671A1 E1437CEE B249D6C4 9F091653 A5725D7E 9526F864 53C50B8D ECFA8C88 4316AB96 FBDB3BA9

Table 6: Khichidi-1-384

The performance of SHA-2-384 and Khichidi-1-384 is highlighted. Both algorithms are executed on an Windows XP operating system , Intel Pentium 4 PC, 2.4GHz CPU with 512MB RAM.

Size of file (Megabytes)	Time taken (seconds) SHA-2-384	Time taken (seconds) Khichidi-1-384
1.4	0.06	0.04
2.2	0.09	0.06
3.3	0.13	0.09
5.9	0.21	0.17
6.9	0.26	0.22

Table 7: Time Analysis of SHA-2-384 and Khichidi-1-384

The entropy test [8] is performed on SHA-2-384 and Khichidi-1-384 algorithms.

Size of file (Megabytes)	**SHA-2-384 (bits per byte)**	**Khichidi-1-384 (bits per byte)**
1.4	3.805037	3.754988
2.2	3.840422	3.908771
6.9	3.751464	3.763552
66.3	3.885934	3.848738
66.7	3.777115	3.841328

Table 8: Entropy Analysis of SHA-2-384 and Khichidi-1-384

SHA-2-512 and Khichidi-1-512 are compared with the help of round operations. Both of them produce an output of 512 bits. SHA-2-384 has 80 rounds whereas our implementation of Khichidi-1-384 has six rounds. These algorithms are compared for the first four rounds.

Let us take Input 1 = 24-bit ASCII string *"abc"* and Input 2 =896-bit ASCII string *"abcdefghbcdefghicdefghijdefghijkefghijklfghijklmghijklmnhijklmnoijklmnopjklmnopqkl mnopqrlmnopqrsmnopqrstnopqrstu"*.

Round operations in SHA-2-512	**Input 1**	**Input 2**
Round 1	f6afceb8bcfcddf5 6a09e667f3bcc908 bb67ae8584caa73b 3c6ef372fe94f82b 58cb02347ab51f91 510e527fade682d1 9b05688c2b3e6c1f 1f83d9abfb41bd6b	f6afce9d2263455d 6a09e667f3bcc908 bb67ae8584caa73b 3c6ef372fe94f82b 58cb0218e01b86f9 510e527fade682d1 9b05688c2b3e6c1f 1f83d9abfb41bd6b
Round 2	1320f8c9fb872cc0 f6afceb8bcfcddf5 6a09e667f3bcc908 bb67ae8584caa73b c3d4ebfd48650ffa 58cb02347ab51f91 510e527fade682d1 9b05688c2b3e6c1f	0b7056a534ae5f62 f6afce9d2263455d 6a09e667f3bcc908 bb67ae8584caa73b f8c7198fe39e4c8c 58cb0218e01b86f9 510e527fade682d1 9b05688c2b3e6c1f
Round 3	ebcffc07203d91f3 1320f8c9fb872cc0 f6afceb8bcfcddf5 6a09e667f3bcc908 dfa9b239f2697812 c3d4ebfd48650ffa 58cb02347ab51f91 510e527fade682d1	2ca82233760c9942 0b7056a534ae5f62 f6afce9d2263455d 6a09e667f3bcc908 303eccccd65953de f8c7198fe39e4c8c 58cb0218e01b86f9 510e527fade682d1
Round 4	5a83cb3e80050e82 ebcffc07203d91f3 1320f8c9fb872cc0 f6afceb8bcfcddf5 0b47b4bb1928990e dfa9b239f2697812 c3d4ebfd48650ffa 58cb02347ab51f91	a023f17ce52cda7b 2ca82233760c9942 0b7056a534ae5f62 f6afce9d2263455d ffdee5eedcc9ca42 303eccccd65953de f8c7198fe39e4c8c 58cb0218e01b86f9

Table 9: SHA-2-512

Round operations in Khichidi-1-512	**Input 1**	**Input 2**
Round 1	11A6986D 16A3FDB6 69C7D84C 0DC54984 95D5E774 F6C18574 F80C2283 72353719 19F8B7EA 96F7087F 8A4D9EB5 EBF3B80A 5D2413AC 0B195CE0 FE431514 64DDA2F3	99683E59 B2BB9C31 5A22694D 911C71C1 0B0DD7F1 2B7C5FDD 27FDC6D1 59B21E6A 8FB5CB6F 3303AA32 0BE3209B 64887195 C93B53A6 5AF05EFB 50853D59 4355CE5D
Round 2	6E289E3A AC3A6844 72A913D4	5947FA71 0BB018F7 29C488D8

Round operations in Khichidi-1-512	**Input 1**	**Input 2**
	41BF0340 F0DB8430 DCB81387 42E325D7 9C449538 4814A63F 0933EC13 BF34580F 486876DA B497981F DB4081A9 D3BD441A 5BA91A86	FA49B831 1CDFEDF7 1212B7AC 6000D1ED B6CEB15E 322CA94D 9DE54C02 E98A7716 8FB79F09 5110CE79 7D72D048 F6F8268D 0434C0E3
Round 3	C6602486 6B75CAEF 77B08699 57E4DABA 83D66AEC 6A8F5AC9 B74A9628 7329E1E7 E41885D4 BC657C42 77F79DFD 48D80A35 0496635B E3F4B12F 3688897D 19087CE5	0EC664B5 7C845298 0ACC6000 DDB788DD 9BD631C8 1D7B8C60 6157331 E5E89234 B160A5C6 FE98C774 25AC6E58 60E0C14D E70AE220 D05FA037 D6DE1A5B D5098C90
Round 4	CC737D91 4DD47B38 AE2A4946 76A26CF0 BA268757 58DA8F3C 597D9797 26B37E40 903855C40 573374B5 7BDD4A89 E8AF98A2 F3FD4464 E32D0CD8 0799DA78 EF0492B	B7EC9690 17269180 30E2F210 1B07638A 81FAC49C 4E06FE03 9E2AC91D 694CA661 C142794E DFED646F 0DB6EA72 644286A9 F74A0786 1478496^{a} BA1BDC96 8CE4CDCF

Table 10: Khichidi-1-512

The performance of SHA-2-512 and Khichidi-1-512 is highlighted. Both algorithms are executed on an Window XP operating system , Intel Pentium 4 PC, 2.4GHz CPU with 512MB RAM.

Size of file (Megabytes)	**Time taken (seconds) SHA-2-512**	**Time taken (seconds) Khichidi-1-512**
1.4	0.07	0.04
2.2	0.13	0.06
3.3	0.19	0.09
5.9	0.27	0.19
6.9	0.36	0.24

Table 11: Time Analysis of SHA-2-512 and Khichidi-1-512

The entropy test [8] is performed on SHA-2-512 and Khichidi-1-512 algorithms.

Size of file (Megabytes)	**SHA-2-512 (bits per byte)**	**Khichidi-1-512 (bits per byte)**
1.4	3.805037	3.754988
2.2	3.840422	3.908771
6.9	3.751464	3.763552
66.3	3.885934	3.848738
66.7	3.777115	3.841328

Table 12: Entropy Analysis of SHA-2-512 and Khichidi-1-512

10. Strength of Khichidi-1

The following table compares Khichidi-1 algorithm with other hash algorithms.

Name of the hash algorithm	Block size (bits)	Word size (bits)	Output size	Rounds	Year of the standard / publication
MD4	512	32	128	48	1990
MD5	512	32	128	64	1992
SHA-0	512	32	160	80	1993
SHA-1	512	32	160	80	1995
RIPEMD-160	512	32	160	5	1996
SHA-224	512	32	224	64	2004
SHA-256	512	32	256	64	2002
SHA-384	1024	64	384	80	2002
SHA-512	1024	64	512	80	2002
Whirlpool	512	-	512	10	2003
Khichidi-1-224	224	32	224	6	2008
Khichidi-1-256	256	32	256	6	2008
Khichidi-1-384	384	32	384	6	2008
Khichidi-1-512	512	32	512	6	2008

Table-13

The performance of Khichidi-1 algorithms for various input sizes. The test is executed on an Windows XP operating system, Intel Pentium 4 PC, 2.4GHz CPU with 512MB RAM.

Size of file (MB)	Time taken (sec) Khichidi-1-224	Time taken (sec) Khichidi-1-256	Time taken (sec) Khichidi-1-384	Time taken (sec) Khichidi-1-512
1.4	0.04	0.04	0.04	0.04
2.3	0.06	0.07	0.07	0.07
4.3	0.12	0.12	0.13	0.13

Table-14

Statistical random test for Khichidi-1

The output of hash value behaves like a random number that can be used in digital signature protocols. For instance, consider H = concatenation of 100 distinct hash values for Khichidi-1 (256-bit) = 25600 bits. Compute cipher text $C = H^{\oplus} M$, where M is a large size message. It is observed that the cipher text C behaves like a random value due to distinct hash values. This method is called data processing stream mode / stream cipher encryption. The computed H can act as a symmetric key in the stream cipher.

The following statistical reports satisfy the relationship $C = H \oplus M$.

<u>Entropy of the original file (M)</u>

Entropy = 3.124613 bits per byte.

Optimum compression would reduce the size of this 28672 byte file by 60 percent.

Chi square distribution for 28672 samples is 2489498.79, and randomly would exceed this value 0.01 percent of the times

Arithmetic mean value of data bytes is 71.1914 (127.5 = random).
Monte Carlo value for Pi is 3.174550021 (error 1.05 percent).
Serial correlation coefficient is 0.806530 (totally uncorrelated = 0.0)

<u>Entropy of the cipher text = $H \oplus M$, where H = 100 different Hash values</u>

Entropy = 6.045497 bits per byte.

Optimum compression would reduce the size of this 28672 byte file by 24 percent.

Chi square distribution for 28672 samples is 157791.80, and randomly would exceed this value 0.01 percent of the times.

Arithmetic mean value of data bytes is 98.8841 (127.5 = random).
Monte Carlo value for Pi is 3.506906656 (error 11.63 percent).
Serial correlation coefficient is 0.589537 (totally uncorrelated = 0.0).

11. Computing the number of rounds in Khichidi-1

Cyclic points

Given a 32-bit word x, compute MP(x) = y € $\{0,1,\ldots, 2^{32} -1\}$ in Khichidi-1. Consider an iterated MP operations on a 32-bit word, as follows:

$MP(n_1) = n_2$
$MP(n_2) = n_3$
...
$MP(n_{j-1}) = n_j$
$MP(n_j) = n_i$

If ni €$\{n_1,n_2, \ldots,n_j\}$, then n_i is a cyclic point of order p, where p = number of distinct elements in $\{n_1, n_2, \ldots,n_j\}$. This order 'p' will determine the number of round operations in Khichidi-1. Practically, it is difficult to obtain cyclic points of maximum order p (where $p = 2^{32}$). If there are many number of cyclic points of different orders, then the smallest order of a cyclic point will be considered.

More number of rounds for Khichidi-1

To compute the number of rounds in Khichidi-1, we define pre-image collision oracle algorithm on MP operations. Since Khichidi-1 is a one way function, it has some collisions that are computationally infeasible to find. We assume that Khichidi-1 has some pre-image collisions that will be helpful to find cyclic points. Without the assumption of collisions, it is difficult to find the cyclic points as well number of rounds. It will be difficult to conclude the number of rounds in Khichidi-1 algorithm for arbitrarily large messages through Entropy approach, since the entropy values of some inner round outputs are almost close. Sometimes, the hash values are good with respect to random properties, but they may have been arisen out of inner collisions that are unnoticed during hash operations. Therefore, Khichidi-1 has found the limitation of cyclic points (32-bit level) through a different approach in order to avoid inner collisions.

Algorithm: Pre-image collision oracle on MP operations

```
Input n   /* number of MP operations to be taken up (approximate) */
For (i=0; i≤ $2^{32}$ –1; i++)  /* i € {0,1,…, $2^{32}$ –1} */
   For (count=0;count ≤ n; count++)
       x = Shuffling (i);
       y = T-function (x);
       z = LFSR (y);
       {
            If z < 0x80000000  /* z is less than $2^{31}$ */
               d = z $\oplus$ 0x8000000;  /* make the most significant bit of z as 1 */
              i = d;
       }
       If i = z  /* check MP_Preimage_oracle (i)  = i */
           Output: z, count;
```

In Khichidi-1, we have considered six rounds, because a cyclic element of order 6 could be found during iterative MP operations using pre-image collision oracle. That is, one of the numbers in [0, 2^{32}] repeats as a cycle on the 7th MP operation. For instance, we have found the smallest order of a cyclic point in the message pre processing operations, mentioned below:

MP_Preimage_oracle(a99c9f70) = b5e7bf40
MP_Preimage_oracle(b5e7bf40) = bd110be4
MP_Preimage_oracle(bd110be4) = fb7f1134
MP_Preimage_oracle(fb7f1134) = ba9dac04
MP_Preimage_oracle(ba9dac04) = ba161474
MP_Preimage_oracle(ba161474) = a99c9f70

Fixing up the number of rounds in Khichidi-1 depends upon the occurrence of cyclic points in MP operations. Khichidi-1 algorithm takes 6 rounds of hash computation in which at least 12 MP operations are required. The maximum number of rounds can go up to 42 (6 cyclic order for each 32-bit x 7 times) for 224-bits hash output. Similarly, for other hash outputs have to follow the same manner. However, the number of rounds that can be increased or decreased depends upon core functions of the Khichidi-1 algorithm. The number of rounds can vary from 1 to 2^{32} -1. On choosing a suitable T-function and Primitive polynomial in Khichidi-1, we may obtain more number of rounds (varying from 10 to 64) that increase the security of hash design. Too many rounds in the Khichidi-1 sometimes may not be helpful for cryptographic applications.

12. Flexibility of Changing Parameters in Khichidi-1

Shuffling function

Any shuffling function can be used in Khichidi-1. The present shuffling function used in Khichidi-1 handles a 32-bit word operation. A 64-bit word operation could also be considered in a shuffling function to produce 224/256/384/512 bits message digest in the Khichidi-1. The lower bound for a shuffling operation should have 32-bit word operation, diffusion property, less computational complexity, efficient and good random properties.

T-function

In our Khichidi-1, the T-function presently used is $(2n^2 + n) \bmod 2^{32}$, which is bijective and the underlying operations are 32-bit. Any other invertible T-function (even 64-bit operation) could also be used in the Khichidi-1. The lower bound for a T-function should have 32-bit operation, good Avalanche effect and nonlinear properties.

LFSR operation

In our Khichidi-1, a 32-degree primitive polynomial $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12}+ x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x +1$ is used for the LFSR operation. We could also consider a 64-degree primitive polynomial that will increase the security of Khichidi-1. The lower bound for an LFSR operation should have a 32-bit primitive polynomial consisting of minimum 15 terms.

Number of shift operations in LFSR

The number of times a 32-bit input is shifted in the LFSR operation that has to be shifted (according to our choice) from 1 to $2^{32} - 1$ times. Presently, one shift LFSR operation has been used in Khichidi-1. There are other ways to improve the security of Khichidi-1 by using different set of shift operations. For every 32-bit input, it is possible to construct different number of shifts in the LFSR operation. This method will increase the uncertainty of the output. The designer should fix the number of shifts during LFSR operations. This adds more randomness to the hash value and prevents fixed point attacks.

The Khichidi-1 could be modified by choosing suitable parameters in Shuffling, T-function and LFSR operations. This will improve the overall efficiency and security of Khichidi-1.

13. Diagrams

This section illustrates MP operations, CBC rounds, different message digest lengths of Khichidi-1 and comparison of Khichidi-1 with SHA-2.

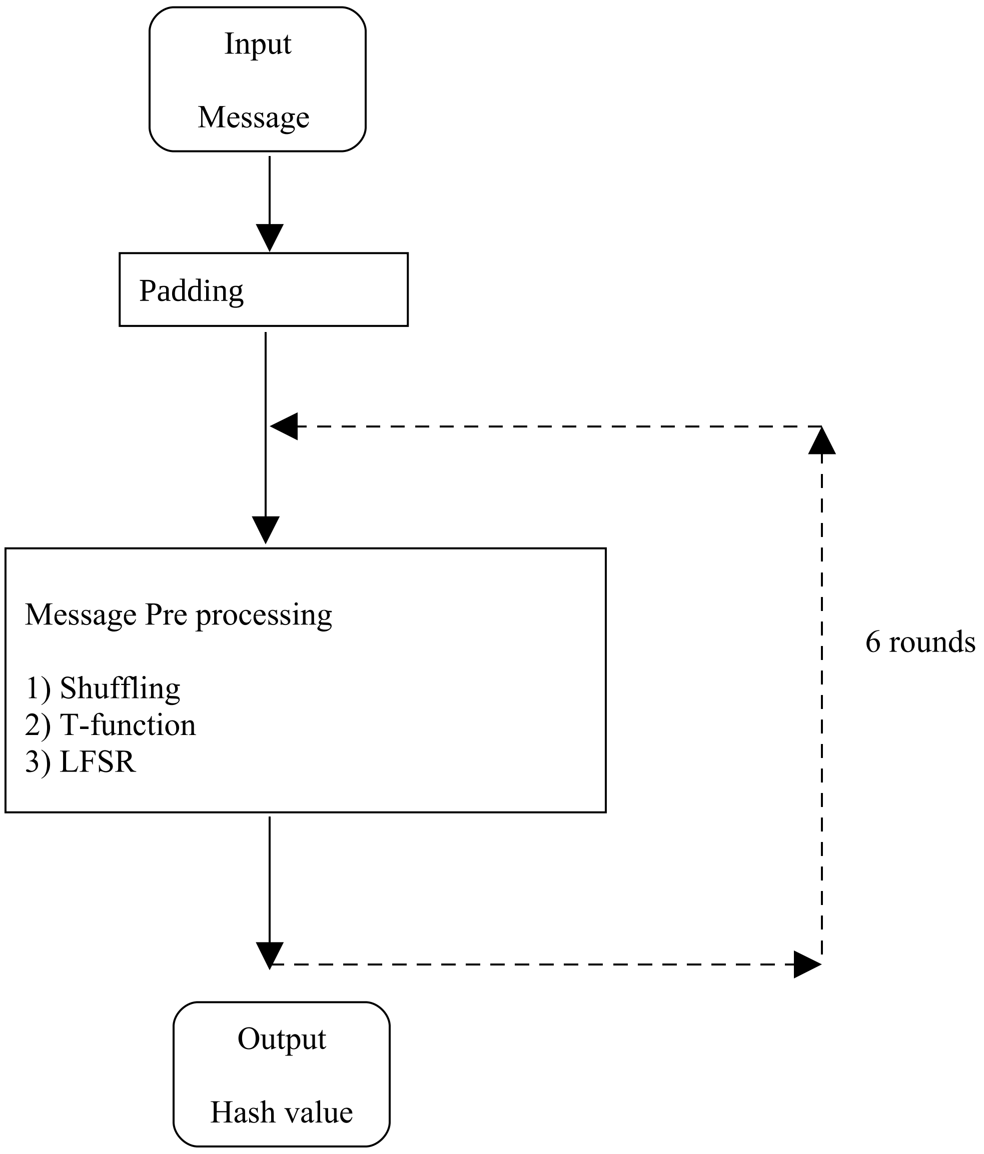


Figure 4: Khichidi-1 hash process

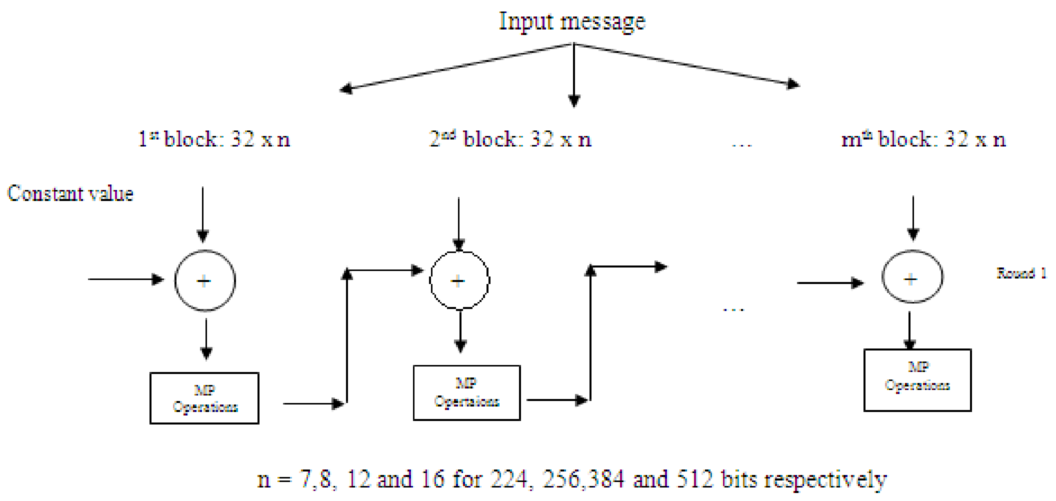


Figure 5: CBC Round One

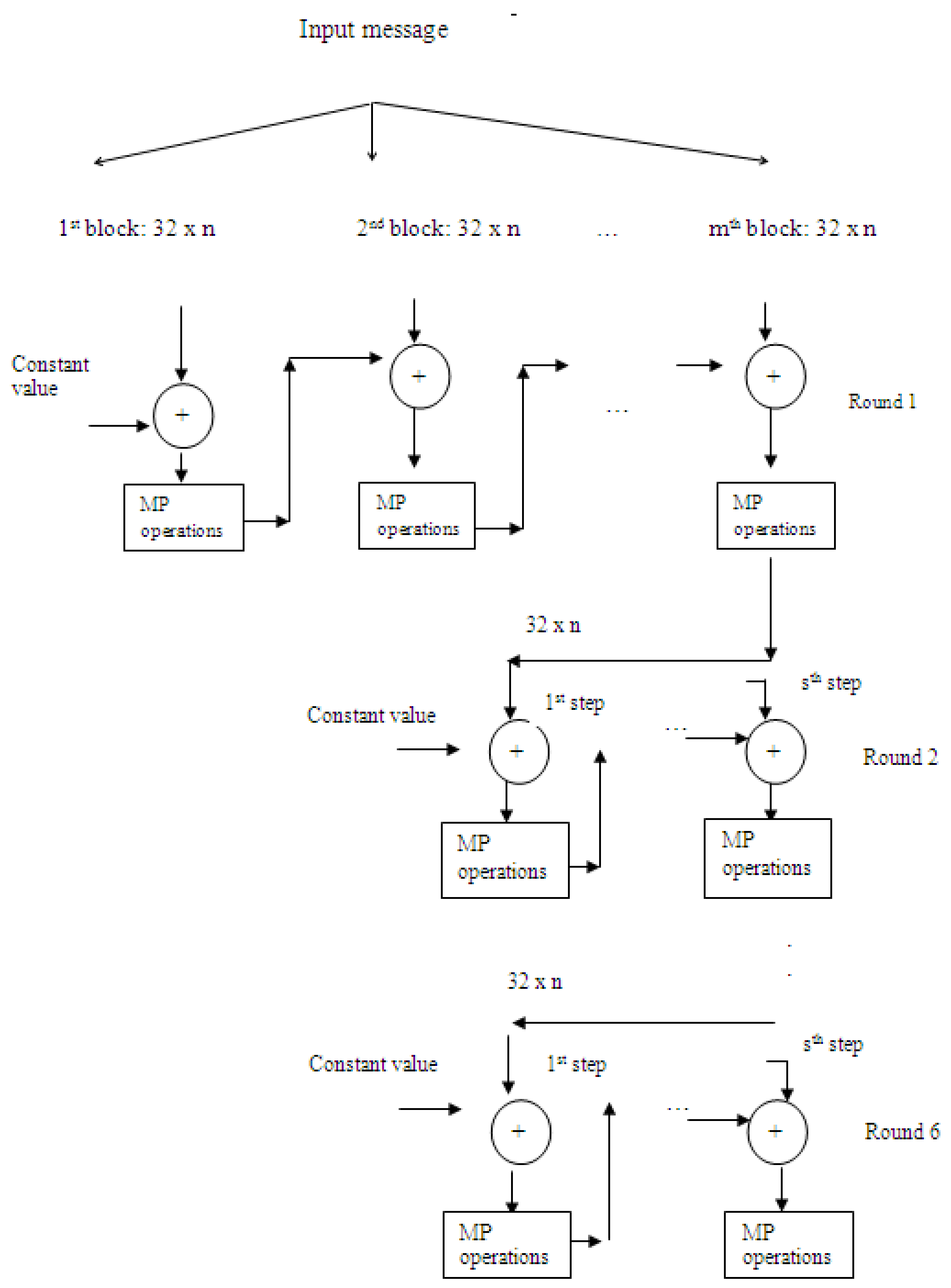


Figure 6: CBC Mode with Iterative Rounds

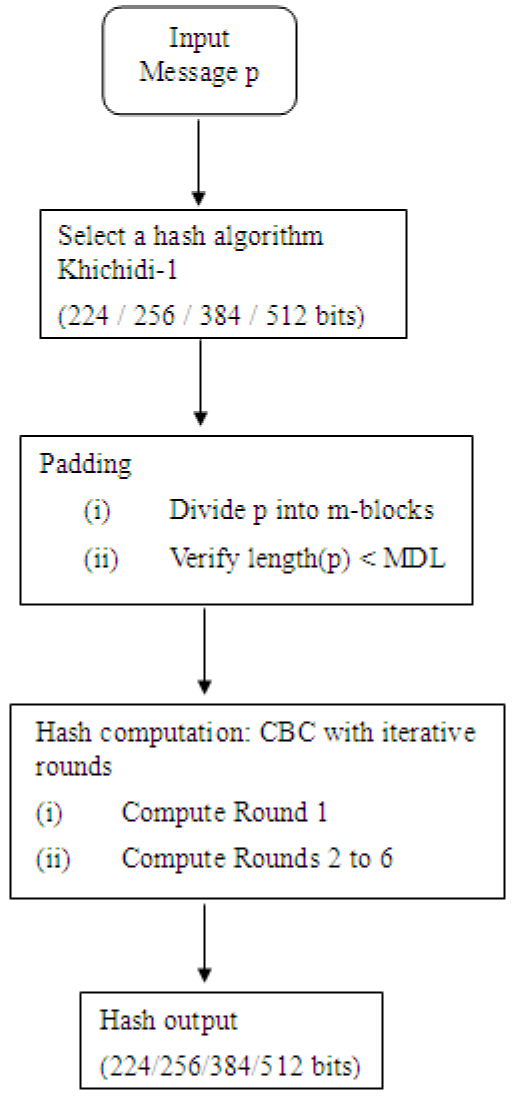

Input
Message p
Select a hash algorithm Khichidi-1
(224 / 256 / 384 / 512 bits)
Padding
(i) Divide p into m-blocks
(ii) Verify length(p) < MDL
Hash computation: CBC with iterative rounds
(i) Compute Round 1
(ii) Compute Rounds 2 to 6
Hash output
(224/256/384/512 bits)

Figure 7: Khichidi-1 Algorithms

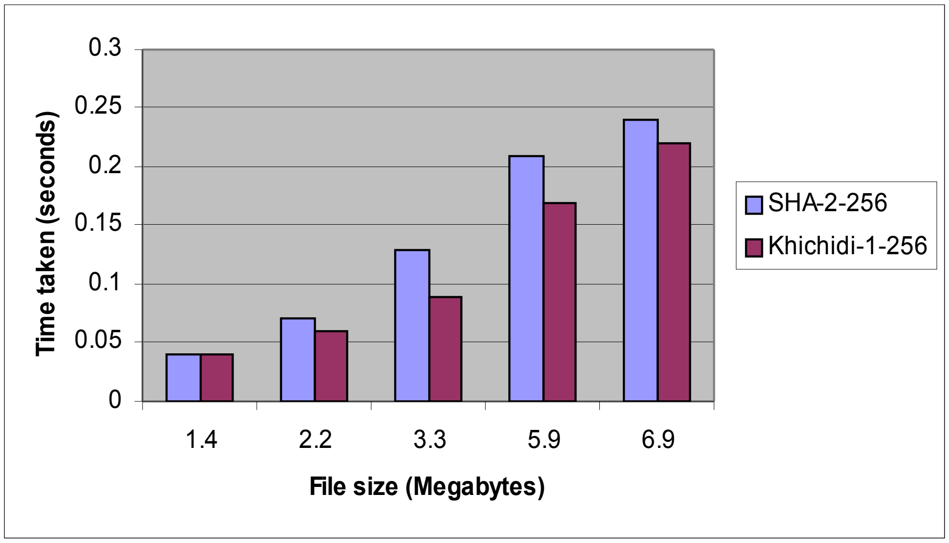


Figure 8: SHA-2-256 and Khichidi-1-256 Performance Comparison

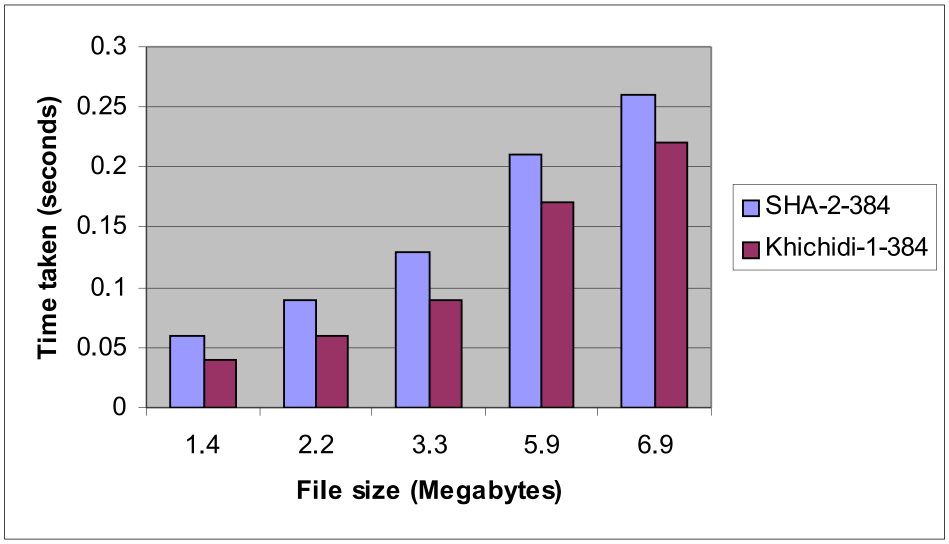


Figure 9: SHA-2-386 and Khichidi-1-386 Performance Comparison

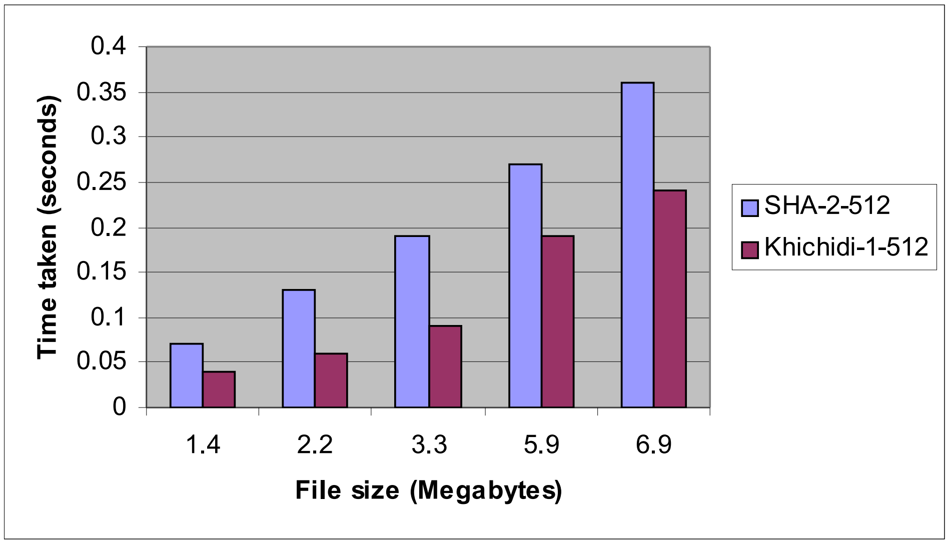


Figure 10: SHA-2-512 and Khichidi-1-512 Performance Comparison

14. Computational complexity of Khichidi-1

The message pre processing operations of one round in Khichidi-1 are shown:

Mathematical operations	Number of times applied
Left shift <<	4
Right shift >>	4
AND &	12
OR |	8

Table 15: Shuffling operation

Mathematical operations	Number of times applied
Multiplication *	2
Addition +	1
AND &	1

Table 16: T-function

Mathematical operations	Number of times applied
Left shift <<	1
XoR mod 2	1

Table 17: LFSR operation

MP operations for a given n-bit input	Computational time complexity
Shuffling	O(n-1)
T-function	$O((n^2))$
LFSR	O(n)

Table 18: Order of complexity

For a given n-bit input message, each MP operation of Khichidi-1 produces an n-bit output with computational time complexity $O(n-1) + O(n^2) + O(n) \approx O(n^2)$. Therefore, Khichidi-1 hash algorithm for a given n-bit input produces an n-bit output with the complexity $O(n^2)$ + €, where € > 0.

Using Intel VTune performance analyzer on Khichidi-1 algorithms, we have analysed computational complexity of the algorithms in Linux 32-bit_Intel Xeon 1.8GHz_3GB RAM. The test performed on a file size of 500 MB is given as follows:

Functional algorithms for Khichidi-1 (224 bits)	**Time (μs)**
Khichidi-1 (224)	15175197
Shuffling	3280542
T-function	1615830
LFSR	2104375

Table 19: Khichidi-1-224

Functional algorithms for Khichidi-1 (256 bits)	**Time (μs)**
Khichidi-1 (256)	15645672
Shuffling	3265468
T-function	1605123
LFSR	2165345

Table 20: Khichidi-1-256

Functional algorithms for Khichidi-1 (384 bits)	**Time (μs)**
Khichidi-1 (384)	16264813
Shuffling	3391435
T-function	1702614
LFSR	2245368

Table 21: Khichidi-1-384

Functional algorithms for Khichidi-1 (512 bits)	**Time (μs)**
Khichidi-1 (512)	16869567
Shuffling	3492342
T-function	1842812
LFSR	2344373

Table 22: Khichidi-1-512

Disassembly code of Khichidi-1 reveals the number of cycles executed for each basic operation. To perform this, we have used MS Visual Studio 2005

Basic Operations	**Number of times applied during one Shuffling operation**
MOV	19
AND	12
SHL	8
SHR	4
OR	4

Table 23: Disassembly code of Shuffling operation

Basic Operations	**Number of times applied during one T-function operation**
MOV	4
IMUL	1
ADD	1
XOR	1

Table 24: Disassembly code of T-function operation

Basic Operations	**Number of times applied during one LFSR operation**
MOV	4
SHL	1
XOR	1

Table 25: Disassembly code of LFSR operation

15 Collision resistance properties of Khichidi-1

Collision resistance

Khichidi-1 has the property of collision resistance. Since Khichidi-1 is not a bijective function, it produces a fixed length output by Theorem 8.3. The security reduction proof of collision resistance can be accomplished with a fairly weak assumption on the relative sizes of the domain and range of the hash algorithm Khichidi-1. We will assume that the hash function Khichidi-1: $X \rightarrow Y$, where X and Y are finite sets and $|X| \geq 2|Y|$. Then there exists a probabilistic Las Vegas algorithm, which finds a collision for Khichidi-1 with probability at least ½ [7].

Preimage resistance

Given an n-bit message digest y, define Khichidi-1(m) = y, where m is a message. Finding out a suitable m for the given y is difficult. In the Round 6 of Khichidi-1, y is obtained from the expression $MP(x)^{\oplus}$ x for some x, an n-bit element. In order to find a suitable message m for the given y, we have to perform at least 2^n hash operations.

Second-preimage resistance

Given a q-bit message m, define Khichidi-1(m) = y, where y is an n-bit output and $q<n$. Finding out another input message m' ($m \neq m'$) such that Khichidi-1(m) = Khichidi-1 (m') is known as Second-preimage attack. In our algorithm, an n-bit value $m_1 = \{ m \| padding(m) \}$ is passed into MP core operations and finally Khichidi-1 produces an n-bit output y, where padding(m) consists of n-q bits. To compute m', at least 2^{n-q} hash operations are to be performed.

Resistance to Length-extension attack

Let Khichidi-1 be an iterated n-bit hash function with CBC mode. Let m be a message of t blocks. Then a second-preimage for Khichidi-1(m) can be found in time (2^n /s)+ s operations of CBC mode, for any s in the range $1 \leq s \leq \min(t, 2^{n/2})$. In particular, when t = s on computing an input message $z||x_{j+1}||x_{j+2}|| \ldots ||x_t$ to collide m, the proof follows from Birthday paradox that at least 2^n /s hash operations have to be performed [2].

Let us take a j-bit hash output specified by taking a fixed subset of the Khichidi-1 algorithm's output bits. It will be difficult to find collisions for j-bit hash value taken out from an n-bit hash output of Khichidi-1. Suppose we find collisions for j-bit hash value in less number of hash operations, then it will be easy to find collisions for n-bit hash output. This is not true because MP core operations and CBC rounds of Khichidi-1 hash algorithm are robust against Preimage, Second-preimage and Length-extension attacks. Therefore, finding collisions for the j-bit subset is as hard as n-bit hash function

Khichidi-1 hash algorithms	**Collision resistance (at least)**	**Preimage resistance (at least)**	**Second Preimage resistance (at least)**	**Resistance to length extension when t = s (at least)**
Khichidi-1-224	112 bits	224 bits	224 bits	224-bit length (s)
Khichidi-1-256	128 bits	256 bits	256 bits	256-bit length (s)
Khichidi-1-384	192 bits	192 bits	192 bits	384-bit length (s)
Khichidi-1-512	256 bits	256 bits	256 bits	512-bit length (s)

Table 26: Work factor of Khichidi-1 with respect to resistance properties

16 Applications of Khichidi-1

HMAC: Keyed-Hashing for Message Authentication

A keyed-hash message authentication code (HMAC) is a type of message authentication code calculated using a cryptographic hash function in conjunction with a secret key. It is a mechanism that provides integrity checks. HMAC uses a secret key for that calculation and verification of MACs. The cryptographic strength of the HMAC depends on the cryptographic strength of the underlying hash function, on the size and quality of the key and the size of the hash output length in bits. The size of the output of HMAC is the same as that of the underlying hash function. We have used NIST approved HMAC [10] with Khichid-1 hash algorithm. HMAC-Khichidi-1 has applications in Data origin authentication and Integrity verification usage.

HMAC Parameters and Symbols

HMAC uses the following parameters:

B	Block size (in bytes) of the input to Khichidi-1.
ipad	Inner pad; the byte x'36' repeated B times.
K	Secret key shared between the originator and the intended receiver(s).
K_0	The key K after any necessary pre-processing to form a B byte key.
L	Block size (in bytes) of the output of the Approved hash function.
opad	Outer pad; the byte x'5C' repeated B times.
t	The number of bytes of MAC.
text	The data on which the HMAC is calculated; text does **not** include the padded key. The length of text is n bits, where $0 \leq n < 2^{B} - 8B$.
x'N'	Hexadecimal notation, where each symbol in the string 'N' represents 4 binary bits.
||	Concatenation
$\oplus$	Exclusive-Or operation.

HMAC Specification

To compute a MAC over the data ‘text’ using the HMAC function, the following operation is performed:

$MAC(text)_t = HMAC(K,text)_t = Khichidi\text{-}1((K_0 \oplus opad) \| Khichidi\text{-}1((K_0 \oplus ipad) \| text))_t$

The step by step process in the HMAC algorithm is shown below.

Step 1 If the length of K = B: set $K_0 = K$. Go to step 4.

Step 2 If the length of K > B: hash K to obtain an L byte string, then append (B-L) zeros to create a B-byte string K_0 (i.e., $K_0 = Khichidi\text{-}1(K) \| 00...00$). Go to step 4.

Step 3 If the length of K < B: append zeros to the end of K to create a B-byte string K_0 (e.g., if K is 20 bytes in length and B = 28, then K will be appended with 8 zero bytes 0x00).

Step 4 Exclusive-Or K_0 with ipad to produce a B-byte string: $K_0 \oplus ipad$.

Step 5 Append the stream of data 'text' to the string resulting from step 4: $(K_0 \oplus ipad) \| text$.

Step 6 Apply Khichidi-1 to the stream generated in step 5: $Khichidi\text{-}1((K_0 \oplus ipad) \| text)$.

Step 7 Exclusive-Or K_0 with opad: $K_0 \oplus opad$.

Step 8 Append the result from step 6 to step 7:

$(K_0 \oplus opad) \| Khichidi\text{-}1((K_0 \oplus ipad) \| text)$.

Step 9 Apply Khichidi-1 to the result from step 8:

$Khichidi\text{-}1((K_0 \oplus opad) \| Khichidi\text{-}1((K_0 \oplus ipad) \| text))$.

Step 10 Select the leftmost t bytes of the result of step 9 as the MAC.

The security of the message authentication mechanism presented here depends on cryptographic properties of the hash function Khichidi-1. Since Khichidi-1 is collision resistant, HMAC-Khichidi-1 is also secure. HMAC-Khichidi-1 follows truncation operation as per RFC 2104 and takes block size of a message = MDL. Forgery attacks are doable with complexity $2^{n/2}$ through birthday attacks and therefore HMAC-Khichidi-1 has the collision resistance with a minimum complexity $2^{n/2}$

Pseudo-Random function (PRF) usage

IKE and IKEv2 use PRFs for generating keying material and for authentication of the IKE Security Association. PRF-HMAC-Khichidi-1 produces the same MDL of Khichidi-1 algorithm without truncation. It takes block size = MDL and follows RFC 4868 [17] for PRF usage with IKE protocols. For security considerations, the PRF-HMAC-Khichidi-1 depends upon the underlying hash algorithm Khichidi-1, the strength lies in the correctness of the algorithm implementation, the security of the key management mechanism, the strength of the associated secret key, and upon the correctness of the implementation in all of the participating systems. The PRF-HMAC-Khichidi-1 produces the output without truncation so that PRF must resist attacks, which are fewer than 2 $^{n/2}$ queries.

Algorithm ID	**Block Size**	**Output Length**	**Truncation Length**	**Key Length**	**Algorithm Type**
HMAC-Khichidi-1-224-112	224	224	112	224	Authentication / Integrity verification
HMAC-Khichidi-1-256-128	256	256	128	256	Authentication / Integrity verification
HMAC-Khichidi-1-384-192	384	384	192	384	Authentication / Integrity verification
HMAC-Khichidi-1-512-256	512	512	256	512	Authentication / Integrity verification
PRF-HMAC-Khichidi-1-224	224	224	none	variable	PRF
PRF-HMAC-Khichidi-1-256	256	256	none	variable	PRF
PRF-HMAC-Khichidi-1-384	384	384	none	variable	PRF
PRF-HMAC-Khichidi-1-512	512	512	none	variable	PRF

Table 27: HMAC-Khicidi-1 algorithms with parameters

HMAC-Khichidi-1 Examples

The following example is provided in order to promote correct implementations of HMAC-Khichidi-1.

Khichidi-1-224 with 20-Byte Key

```
Text:   53616D70 6C652023 32800000 00000000 00000000
        00000000 00000000 00000000 00000000 00000000
        00000000 00000000 00000000 00000000 00000000
        00000248
Key:    30313233 34353637 38393A3B 3C3D3E3F 40414243
K0:     30313233 34353637 38393A3B 3C3D3E3F
        40414243 00000000 00000000
```

$K_0 \oplus$ ipad:

```
06070405 02030001 0E0F0C0D 0A0B0809
76777475 36363636 36363636
```

(Key $\oplus$ ipad)||text:

```
06070405 02030001 0E0F0C0D 0A0B0809
76777475 36363636 36363636 53616d70
6c652023 32800000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000248
```

Khichidi-1-224((Key $\oplus$ ipad)||text):

```
BE4677715C7E27C4A6B0ED6A016457FEAC5E1EF68C27B70C04C291AA
```

$K_0 \oplus$ opad:

```
6C6D6E6F 68696A6B 64656667 60616263 1C1D1E1F 5C5C5C5C 5C5C5C5C
```

($K_0 \oplus$ opad) || Hash((Key $\oplus$ ipad)||text):

```
6C6D6E6F 68696A6B 64656667 60616263 1C1D1E1F 5C5C5C5C 5C5C5C5C
BE467771 5C7E27C4 A6B0ED6A 016457FE AC5E1EF6 8C27B70C 04C291AA
```

HMAC(Key, Text) = Khichidi-1-224(($K_0 \oplus$ opad) || Khichidi-1-224((Key $\oplus$ ipad)||text)):
6F085C9D 5BC2B869 CC427D83 0CBAF736 B5F883A3 7B7A2A99 A4417D44

28-byte HMAC(Key, Text):
6F085C9D 5BC2B869 CC427D83 0CBAF736 B5F883A3 7B7A2A99 A4417D44

HMAC-Khichidi-1-224-112
6F085C9D 5BC2B869 CC427D83 0CBA

PRF-HMAC-Khichidi-1-224-224

6F085C9D 5BC2B869 CC427D83 0CBAF736 B5F883A3 7B7A2A99 A4417D44

Randomized Hashing

To generate a digital signature for a message, the message is hashed by using a hash algorithm first, and then signing the resulting hash value using one of the Approved digital signature algorithms [9]. We follow NIST SP 800-106 Draft [16] to generate a randomized message M, which is hashed by using Khichidi-1. Then the hash output is used in the approved digital signature algorithms. The object of the randomizing method is independent of the hash algorithm Khichidi-1.

The attacker calculates hashed values for messages until a pair of messages is found to have the same hash value. At least one of the messages (message *A*) must be chosen such that a signer would be willing to sign it; the other message (message *B*) would be chosen such that its contents would be beneficial to the attacker in some manner. The attacker asks the signer to sign message *A*, but the attacker uses the signature computed on message *A* for the other message (message *B*) that the signer did not sign. This attack is computationally possible if the hash function is not strongly collision resistant.

When using Khichidi-1 in the randomized hashing technique described for the digital signatures schemes, the attacker does not know what the hash value of the randomized message will be before the digital signature is generated. Therefore, the attacker cannot conduct the collision attack. However, the attacker may be able to conduct other attacks on the hash functions. Since Khichidi-1 has Preimage resistant of minimum computational complexity 2^n, the attacker finds difficult to forge signatures. It is secure to use Khichidi-1 hash algorithm in the randomized hashing technique.

Industrial applications

Khichidi-1 hash algorithm finds a number of applications in Information Security. Some specific areas where our invention can be applied are:

1. Signature protocols
2. Digital Identity
3. Access Control
4. Multifactor Authentication
5. Message Authentication Code (MAC)
6. Data integrity in a relational database

17 Advantages and Limitations of Khichidi-1

Advantages

- Entropy properties and performance of Khichidi-1 algorithm are better than SHA-2 algorithm. The comparison is made (see Chapter 9).
- Khichidi-1 algorithm is based on MP function, which is bijective (see Chapter 8) and keeps maximum cycle due to LFSR operation.
- Khichidi-1 is not based on logical operations whereas SHA-2 is based on them.
- Khichidi-1 algorithm takes at least 6 rounds of CBC operations (see Chapter 11).
- Khichidi-1 allows proprietary functions (see Chapter 12) whereas SHA-2 won't be.
- Khichidi-1 sets the initial hash values $H^{(i)} = 0$ whereas SHA-2 sets random values.
- Suitable Initialization values $H^{(i)}$ can be taken as constants without invoking trivial pseudo collisions.
- Shuffling operation could be extended to 64-bit manipulation.
- T-function: $f(n) = (2n^2+n)$ mod FFFFFFFFFFFFFFFF could also be extended to a 64-bit manipulation.
- A 64-bit primitive polynomial could be used to generate maximal cycles in LFSR operation.
- Khichidi-1 hash algorithm is easily portable in hardware and mobile devices because the core operations of Message Preprocessing can be implemented in micro controllers.
- Khichidi-1 hash algorithm takes 6 rounds so that the execution speed is very fast.
- Khichidi-1 hash algorithm can be extended to a 1024-bit hash function.
- Khichidi-1 can be converted into a keyed hash function.
- Khichidi-1 has a non-correlation property, which means that Input bits and output bits should not be correlated.
- Khichidi-1 has near-collision resistance and partial-preimage resistance properties.

Limitations

- Khichidi-1 does padding that is different from SHA-2 (see Chapter 5)
- In Khichidi-1, cyclic points can easily be computed.
- Khichidi-1 can be parallelizable like AES-CBC algorithm.
- Initialization values $H^{(i)}$ could not be taken arbitrarily.
- For a 64-bit T-function, a proper Math library has to be created to perform basic arithmetic operations over 8/16/32/64 platforms.
- While using 32-bit and 64-bit primitive polynomials, Khichidi-1 produces different hash outputs for the same message.
- For security reasons, 64-bit T-functions and primitive polynomials are safer.

18. Performance Estimates

18.1 Speed

The speed analysis of Khichidi-1 algorithm is performed using Vtune on an Intel Xeon 1.86 GHz clock speed, 4GB RAM running 32 and 64 bit Linux operating systems with kernel version 2.6. Similar performance can be expected for NIST reference platform.

Operations	**Time taken (seconds) per call**
Message Preprocessing	115×10^{-9}
convertLength	7.5408×10^{-8}
Update	84.82×10^{-6}
Final	14×10^{-6}
digest	1.04×10^{-6}
uadd	0.121×10^{-6}
bitcpy	89.7×10^{-6}
shuffling	35.123×10^{-9}
t-function	20.7×10^{-9}
lfsr	28.11×10^{-6}
Init (algorithm setup)	0

Note: The performance variation between 32 bit and 64 bit OS is negligible

18.2 Cycles

The clock cycle analysis of Khichidi-1 algorithm is performed using Vtune on an Intel Xeon 1.86 GHz clock speed, 4GB RAM running 32 and 64 bit Linux operating systems with kernel version 2.6.

Operations	Cycles for the entire hash process performed on a large size file (159 MB)	Khichidi-1-224 (Cycles per byte)	Khichidi-1-256	Khichidi-1-384	Khichidi-1-512
Message Preprocessing	172950000	29	33	49.8	66.4
convertLength	1590000	0.27	0.31	0.46	0.61
Update	34490000	6	6.6	9.93	13.24
digest	384270000	65	74	110.7	148
uadd	3140000	0.53	0.6	0.90	1.2
bitcpy	44310000	74	8.5	12.8	17
shuffling	430100000	72	82.6	124	165
t-function	156290000	26	30	45	60
lfsr	212170000	36	40.7	61	81.5
Init(algorithm setup)	0	0	0	0	0

Note: Message Preprocessing takes 29 cycles per byte on a given file size 159 MB = 166680580 bytes i.e., (172950000 / 166680580)*28 = 29

18.3 Hardware estimates

For most applications, Khichidi-1 algorithm can be implemented in hardware and software. Khichidi-1 uses add, multiply, exclusive-or, left and right shifts, and logical operations (AND and OR), which are well supported on modern microprocessors.

The basic 32-bit arithmetic operations (used to compute hash functions) of Khichidi-1 algorithm with chip area dimension on a hardware platform are mentioned below. The hardware estimation of our algorithm is taken on 120 x 100 microns in area with a standard 0.25 micron CMOS process [20].

Shuffling operation	**Time (nanoseconds)**	**Area (mm^2)**
shift operation (8)	8	0.048
OR (4)	4	0.024
AND (12)	12	0.036
Total	24	0.108

T-function	**Time (nanoseconds)**	**Area (mm^2)**
multiplication (2)	6	0.014
XoR (1)	0	0.000
shift operation (1)	1	0.006
add (1)	1	0.003
Total	8	0.016

LFSR operation	**Time (nanoseconds)**	**Area (mm^2)**
shift operation (1)	1	0.006
XoR (1)	0	0.000
Total	1	0.006

From the above calculation, Khichidi-1 occupies 0.130 mm^2 area for one MP operation. For 6 rounds, Khichidi-1 algorithm occupies minimally 12 x 0.130 = 1.560 mm^2, which is a theoretical estimation.

19. Initialization vectors of Khichidi-1

The following Initialization vectors of Khichidi-1 hash family of algorithms are mentioned below. These values are taken randomly and they do not have fixed point mappings.

For Khichidi-1-224, Initialization vectors are

H_0 = 0xF3A4D825
H_1 = 0x01A967BD
H_2 = 0x72351F0B
H_3 = 0xB6431C8F
H_4 = 0x9D7D102E
H_5 = 0x256A6E47
H_6 = 0x4F5B8629

Cnst =E319250F 0B836F1D 2A9F3151B D583CB17 83BF211E 970AB13D 6D42A8AF

For Khichidi-1-256, Initialization vectors are

H_0 = 0xFD54588A
H_1 = 0x689AB1CE
H_2 = 0xD8547F0C
H_3 = 0x96458ACF
H_4 = 0x6B425976
H_5 = 0x497A6E45
H_6 = 0x095F6817
H_7 = 0xC14E3D6A

Cnst =D2493F69 A1319D5B F24A9635 8A93F51B B38F121E D070C9573 5A09F627 661AF874

For Khichidi-1-384, Initialization vectors are

H_0 = 0x8194BCFC
H_1 = 0xC45B7C43
H_2 = 0x E2C6D391
H_3 = 0x A73D8C0B
H_4 = 0x B50137AB
H_5 = 0x D9267551
H_6 = 0x C1CD90E3
H_7 = 0x 77E0C125
H_8 = 0x2F960E5C
H_9 = 0x43EA4E65
H_{10} = 0x0A3F6827

H_{11} = 0xB843ED6C

Cnst =B82EA2D9 ED1C690D BD6230D6 3F69A138   E88FEA0C FD99D645
0AD36B97 F5249A36 8A39F15B B32EEA2A C2A537FB 7A09F265

For Khichidi-1-512, Initialization vectors are

H_0 = 0x2953316F
H_1 = 0xF37B8C43
H_2 = 0x E8643D1A
H_3 = 0x 7C5D0C8B
H_4 = 0x 5B1037BA
H_5 = 0x 8AB91DEF
H_6 = 0x 4B2F3EE8
H_7 = 0x 3BE345ED
H_8 = 0x 2F960E5C
H_9 = 0x12CE95A3
H_{10} = 0x08AD7F27
H_{11} = 0xE345ED5F
H_{12} = 0xF519A5CB
H_{13} = 0xF519A5CB
H_{14} = 0xC27B0D5A
H_{15} = 0xF62995CD

Cnst = E90EB183 C853E237 E5D0F331 2B1D379A 3A8726AE CEF1D031 D7546291
A62813B5 9F017B26 3C28708A 91518F8F 9056AB79 C8F1B0D3 CD4AE1BA
7546297D ED0C691D

20. References

[1] Alexander Klimov and Adi Shamir, *A New Class of Invertible Mappings*, Lecture Notes In Computer Science; Vol. 2523, 2002. www.wisdom.weizmann.ac.il/~ask/t0.ps.gz

[2] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 2001.

[3] Antoine Joux, *Multi-collisions in iterated hash functions, Application to Cascade Constructions*. In Matt Franklin, editor, Advances in Cryptology-CRYPTO'04, volume 3152 of Lecture Notes in Computer Sciences, pages 306-316, Springer 2004.

[4] S. Bakhtiari, R. Safavi-Naini and J. Pieprzyk, *Cryptographic hash functions: A Survey, Technical* Report 95-09, Department of Computer Science, University of Wollongong, July 1995.

[5] D. Coppersmith, *Analysis of ISO/CCITT Document X.509 Annex D*, IBM T. J. Watson centre, Yorktown Heights, N. Y., 10598, Internal Memo, June 11, 1989.

[7] Douglas Stinson, *Theory and Practice: Cryptography*, Chapman & Hall / CRC, 2002.

[8] ENT, *A pseudorandom Number Sequence Test Program*. http://www.fourmilab.ch/random/

[9] FIPS PUB 186-3, *Secure Hash Standard*, June 2007.

[10] FIPS PUB 198, *The Keyed-Hash Message Authentication Code (HMAC)*, March 2002.

[11] FIPS PUB 180-3, *Secure Hash Standard*, October 2008.

12] Henry S. Warren, Jr, *Hackers Delight*, Addison-Wesley, 2002.

[13] John Kelsey and Bruce Schneier, *Second pre images on n-bit hash functions for much less than 2n work*, In Ronald Cramer, editor, Advances in Cryptology-EUROCRYPT'05, volume 3494 of Lecture Notes in Computer Science, pages 474-490, Springer, 2005.

[14] John Kelsey and Tadayoshi Kohno, *Herding hash functions and the Nostradamus attack*, Eurocrypt 2006.

[15] Nirmal Saxena & Edward J. McClusky, *Primitive Polynomial Generation Algorithms -Implementation and Performance Analysis*, CRC Technical Report 2004.

http://www.crc.stanford.edu/crc_papers/CRC-TR-04-03.pdf

[16] NIST SP 800-106 Draft, *Randomized Hashing Digital Signatures*, July 2007.

[17] RFC 4868, *Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 wi*, May 2007.

[18] Richard Drews Dean, *Formal Aspects of Mobile Code Security*, Ph. D Thesis, Princeton University, 1999.

[19] R.L. Rivest, *The MD4 message-digest algorithm,* Request for Comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force, April 1992.

[20] Ronald L. Rivest1, M.J.B. Robshaw, R. Sidney, and Y.L. Yin, *The RC6TM Block Cipher Version 1.1*, 1998.
http://people.csail.mit.edu/rivest/Rc6.pdf

[21] *Secure Hash Standard*, Federal Information Processing Standard (FIPS), Draft, National Institute of Standards and Technology, US Department of Commerce, Washington D.C., January 31, 1992.

[22] SHA-1 Cryptographic Hash Update, *A perspective on practical security by Landon Curt Noll*, 2005.
www.systemexperts.com/tutors/CryptographicHashUpdate.pdf

[23] N. Vijayarangan and S. Kasilingam, *Random number generation using primitive polynomials*, proceedings of SSCCII-2004, Italy.

[24] N. Vijayarangan and R. Vijayasarathy, *Primitive polynomials testing methodology*, Jour. of Discrete Mathematics Sciences & Cryptography, Vol.8, No.3, pp. 427-435, 2005.

[25] Xiaoyun Wang and Hongbo Yu, *How to break MD5 and other hash functions*, www.infosec.sdu.edu.cn/paper/md5-attack.pdf

[26] Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu, *Collision Search Attacks on SHA-1*, February 13th 2005.
http://theory.csail.mit.edu/~yiqun/shanote.pdf

[27] India patent application no. 1937/MUM/2007 (dated October 2007) by N. Vijayarangan, TCS, *Cryptography and Information security*.

[28] USA patent application no. 12 / 079,537 (dated March 2008) by N. Vijayarangan, TCS, *A method for preventing and detecting hash collisions of data during the data transmission*.

[29] India patent application no. 891/MUM/2008 (dated July 2008) by N. Vijayarangan, TCS, *A method for designing a secure hash function and a system thereof.*

Appendix A: Khichidi-1 (224 bits) Examples

A1. Let the message be the 24-bit ASCII string ‘abc’, which is equivalent to the binary string
01100001 01100010 01100011.

Round 1 =
C69FC1230BF9BBE2E669A82B1A2579E57CBD810AB7E6305C3ED6BCC7
Round 2 =
6A81CF7F3D565A2612CE5E5AF39E7CC23EF39B93A3196F85EB1E3A6C
Round 3 =
B1D9BA91273F9D826729EA941778DF3BE31C5A935A16C24B8EE919B9
Round 4 =
67065FD0585F422B56DA76A9B5F7E9EFA8D8EFF387164967AB16380C
Round 5 =
E66CE512C4199E7AEE87C01566C83B7FF0E4F517236822A7976AFB7E
Round 6 =
EEFFAC89F540FC49BE3C06339EFE255782ADC610F8C28C053C853C59

The resulting 224-bit Message Digest is
MD = EEFFAC89F540FC49BE3C06339EFE255782ADC610F8C28C053C853C59

A2. Khichidi-1 (224-bits) ("The quick brown fox jumps over the lazy dog")

Round 1 =
E1AB359418A455C41E26C9D732A98929D3C3961D2A5B9C1A7A509D9B
Round 2 =
26B1CD9A453B107C234A022563D952F35802E1F8CEE2259CB7CE315E
Round 3 =
5890BBF58D2ED57AAA17B0CE3EE9E10138BC6ACDBE1460214BF642DE
Round 4 =
4B6E89659B0C5B8E40DDF3139AC88A885DF87451960C0B40AF5F764F
Round 5 =
CA606D116E0865CECD27FD8A5563B7DB9EEFE11B213791953EC66BC8
Round 6 =

0D0AE72A091BB130665BB1BFCD159C97E2B62EB0EBB56DCA336FAA9B

The resulting 224-bit message digest is

MD = 0D0AE72A091BB130665BB1BFCD159C97E2B62EB0EBB56DCA336FAA9B

Even a small change in the message will (with overwhelming probability) result in a completely different hash; for example, changing d to c:

Khichidi-1 (224-bits) ("The quick brown fox jumps over the lazy cog")

Round 1 =

0657B1D0739CEC5345A1D51967532D1D76874DADB13DD55AC188BE6B

Round 2 =

BA8B04C9BFAE8E5B2263E7DB7F54FA68C6741F9A1984E914237A9DCE

Round 3 =

D739BD783B76E355E488BE7C821AB6D7CA80E701D4B1AFE6990B0724

Round 4 =

D7256EE5FB16FFA4C679AD6770E1F613E30E029C03AC595BDC6D4F02

Round 5 =

332CAAA3E08DBBE9A21BFA9FF0B13307FA133C623491BB0D1A4A8FCE

Round 6 =

AD9F426DCE8910876814179049FE2405923AA4B2904F3CC255ABD643

The resulting 224-bit message digest is

MD = AD9F426DCE8910876814179049FE2405923AA4B2904F3CC255ABD643

A3. The hash value of Null string / Zero-length message (“”) is

Round 1 =

CD0C4F0253CFA5888CD004D3336AC3D9021572D63E49FE4BC88451E5

Round 2 =

08B1163625F1FD8F613AABB484289E914256F0B29A4FEB0D06A0A724

Round 3 =

0D7955C6C6E7E78690FD4DDC3012FE3DC2B98B790853EF70C58C6D73

Round 4 =

0DBB260EF216C877FAE40A95114805D0509072AAAE22AC1B018582D0

Round 5 =

0A1F632600CCC72A30C1E88E8CC6828B4FC00B21BC0E94D807777F24

Round 6 =

2E6A0131F8F0747EE96BB3A37357EA63A295256CD326B1C4A10151A3

The resulting 224-bit Message Digest is

MD = 2E6A0131F8F0747EE96BB3A37357EA63A295256CD326B1C4A10151A3

Appendix B: Khichidi-1 (256 bits) Examples

A1. Let the message be the 24-bit ASCII string ‘abc’, which is equivalent to the binary string
01100001 01100010 01100011.

Round 1 =
0BF9BBE2E669A82B1A2579E57CBD810AB7E6305CE55013F03353F3B762A73475
Round 2 =
F834752DE2F5AAFCF163BD31B81088C9E52CA136EBC92FA84BD0BADE924A30D5
Round 3 =
39285FA268CF297C5CF060D17C968F375DBD0A56E4252C9FB8A8F76A71D929DE
Round 4 =
944E9B56A3AB08ABE0DD16AC3875C4952344756EF10EB22AC2A549501B10B828
Round 5 =
DA6DDC52EFFF62A11D1BD1C103B444705B89B95F2B4B056DBC9F4CB6D61879BB
Round 6 =
99C9B9744FC81C819B0A1813CD8940FED69DB5B14CFA07F0BA591DFC3C02E872

The resulting 256-bit Message Digest is
MD = 99C9B9744FC81C819B0A1813CD8940FED69DB5B14CFA07F0BA591DFC3C02E872

A2. Khichidi-1 (256-bits) ("The quick brown fox jumps over the lazy dog")

Round 1 =
1E26C9D732A98929D3C3961D2A5B9C1A46382EAC100641A00916BCE7C779C6A9
Round 2 =
758E6E2058809E0DA5038E1317AB9B86B64E0BACFB6BF817C8F894E311F970AB
Round 3 =
65E34D64DF83707A8AC717313E027EF9A3BF54E60E64B7463A1CA6E65262623D
Round 4 =
87FDABC63A2EE0F085DD209DDF1077B34E06706D959D8889845FFEF28ABA0285
Round 5 =
6B10B4600AD50DE74ABD23C88CE9D59203B511B55C4D3FA077CC55EFC9316EAF
Round 6 =

5BC619118D9A4C4CA0B14504B22FFD68C5F35F760B327FB8D3E10029511617DF

The resulting 256-bit message digest is

MD = 5BC619118D9A4C4CA0B14504B22FFD68C5F35F760B327FB8D3E10029511617DF

Even a small change in the message will (with overwhelming probability) result in a completely different hash; for example, changing d to c:
Khichidi-1 (256-bits) ("The quick brown fox jumps over the lazy cog")

Round 1 =
45A1D51967532D1D76874DADB13DD55A62DF825CC7F7A884422E2904C5D8A947
Round 2 =
DAF71A0F6FEC5B3FCEE3FE2CF178B7B884F6AD0481EA801405FD4FF7BE9F61E3
Round 3 =
7D5CBC463FAB6531CCE56B91741FDF5DB9971DAA2EEEFCCB06263D440FFF1172
Round 4 =
77B7628112531F40056F75219A2E62A0D3775A1C1707AEC5DFBD502A90F5EE27
Round 5 =
D4D17DCC27A8E407DDC0593B133DF4C9E3EA98F6ACF5E7128243963733E3F2E7
Round 6 =
6EA429257F9D303F317154741271E2D11D8868428CAEC154452B50A53C716CFF

The resulting 256-bit message digest is

MD = 6EA429257F9D303F317154741271E2D11D8868428CAEC154452B50A53C716CFF

A3. The hash value of Null string / Zero-length message (“”) is

Round 1 =
53CFA5888CD004D3336AC3D9021572D63E49FE4BC88451E590106DE647C96FDF
Round 2 =
A68540993D60675C5EA2F341CFAF7C82FF2D40A1B63DFE34D59F0C6B41533F70
Round 3 =
8E974702F8BABF0C8638602652FDBCB3CD9CBBFF933E22E5CD92998BC00CCCE5
Round 4 =
1E04A99E1D2285984B44BE4F77EAFD0318DB0247908F336B4F71A75492C0FBDD

Round 5 =

C7B7AC05D80C003DCBFD06FF686B47843D7DC46EE67C2DF61E4EF7DCF806BF96

Round 6 =

33BD038ED2D9FB65CF2F4F0811E49957207809AAD5F17B57C59CDFD5A268B609

The resulting 256-bit Message Digest is

MD = 33BD038ED2D9FB65CF2F4F0811E49957207809AAD5F17B57C59CDFD5A268B609

Appendix C: Khichidi-1 (384 bits) Examples

A1. Let the message be the 24-bit ASCII string ‘abc’, which is equivalent to the binary string
01100001 01100010 01100011.

Round 1 =
B7E6305CE55013F03353F3B7C60FB8F511A6986D16A3FDB669C7D84C0DC5498495D5E774F6C185
74F80C2283A180DF2E
Round 2 =
F1808E408466D3B74E4FDBE33183F5DF19133D6B2BF568910E7357C145CFB7368B601758245441B4
0ACC5FB9C2CF6B4E
Round 3 =
7F487C61F56EA60F02D3FC0327DACAA02C6BD1E501C18070C234CAE18B64AA42AB08A2BFB47
DD4992B02E1A3DA9E648A
Round 4 =
6967B6AE2797F52AF18BFBC69CF68E0B961C7F7CEA8AE543DDF0AB9816ABA3082C9A4142FE81
76D53D6EE84C0E623A68
Round 5 =
576687B32BFB391DF4A765597F27F69CF61D46A4DB9827354D75F9911729F016FB282D73E39D38B
CC425AFA4202EC984
Round 6 =
8E9F3D720491B3799249B550A0F5AF47D15570D92004BCF423224C7D82242722BE9740467748086F
B06C73921B292E78

The resulting 384-bit Message Digest is
MD =
8E9F3D720491B3799249B550A0F5AF47D15570D92004BCF423224C7D82242722BE9740467748086F
B06C73921B292E78

A2. Khichidi-1 (384-bits) ("The quick brown fox jumps over the lazy dog")

Round 1 =
C0886ABEAE8969E8E1AB359418A455C41E26C9D732A98929D3C3961D2A5B9C1A46382EAC10064
1A00916BCE7C779C6A9

Round 2 =
5F226476C93105CCD4FF452761E17CA2997158DA82179B6EA8B71BAE5A793F609C335403E289C4
12973439DAA73CB712
Round 3 =
2AE78BF073B57247F59D97A737EC8EE691FD92F410CBBDD839D1C41F2DB3088EA01BE72643C54
A87D28CEA4AD9406B37
Round 4 =
5D7E7AD618A9554D278204BF580ECC16C50C029B2ABA02E2C90B611A087B46971EB24206759361
3A2F4E22E468774349
Round 5 =
6B10B4600AD50DE74ABD23C88CE9D59203B511B55C4D3FA077CC55EFC9316EAF
Round 6 =
B40E35023F9D6D02348E4DEDDB1B3CEEF5C7D16D3C2DA2856908AF1ED6179C7A0EFC48433084
CF0D8ECF00D659D08939

The resulting 384-bit message digest is
MD =
3BBBDDB487B42F0F65092F58717DA7CF4C4112EFF32E3BBF78530F212F40B799220F4408AC0078
9AE67432848B9157D1

Even a small change in the message will (with overwhelming probability) result in a completely different hash; for example, changing d to c:
Khichidi-1 (384-bits) ("The quick brown fox jumps over the lazy cog")

Round 1 =
0224C1AE7C06EB280657B1D0739CEC5345A1D51967532D1D76874DADB13DD55A62DF825CC7F7
A884422E2904C5D8A947
Round 2 =
D9E9A3F61FE749BB55FB2B595BB48DEC1C9E62BD8F73A354C5A69471C1716695B4ADC216FCB3
19589BD06AF4D7E6676E
Round 3 =
2379D5DD1825880F3F312FB839EFD9744406AAAAD400287CDB79280126E24720F6E42F8CA1AC96
B3934A854E5146A940
Round 4 =
B0BD28A2A2607591083C22AA604F5D7B3CB6872A1D53130FDDB5184F97ECA4B5D54A79869A2B
B53675079323B632D565

Round 5 =

CA4BAA0196BA8CA4A4A8EC8FECC3A2B39CEA77EA4423C37D7A6DD12F52A1FF1CD101471B5
C6144C1F7788B2F0C20EBD8

Round 6 =

031FFEC53806F6DD86847C2FFEEF7B841BBAC27CEF455C9DB5294B487E47FA979E5EBC47199B3
43B2499B173CA7D1852

The resulting 384-bit message digest is

MD =

031FFEC53806F6DD86847C2FFEEF7B841BBAC27CEF455C9DB5294B487E47FA979E5EBC47199B3
43B2499B173CA7D1852

A3. The hash value of Null string / Zero-length message (“”) is

Round 1 =

3E49FE4BC88451E590106DE647C96FDF2CE0F04589A334A68BC5FBFB62E128923681829C4D0C7B
B442EEF2D763BFD429

Round 2 =

F127024F7C5137AC3AF1AD1C1743D96EF2980795F52F7B15FF596DF80360487F9F5405650A0D8306
60758BFAB1C21C09

Round 3 =

07786F57E5EC63E5A87CFCEA8767CFB357C0B18CD5B04ABABA6E142BC40BF4709B91EC6D3F0
BD12EBF864870C5B5A5C6

Round 4 =

3079761FF5E9A15CA2CA964B3737A9D41F908BD363D35A86AF2C953D7C7A40ED773E67133E03C
83DC9FA723D80555DD5

Round 5 =

C64420B026CA1143EEAB5D37A317DB19D40D5DECFFF11418DF0CCFBD62ED52E74D25B57348B
42138A3415836736231AE

Round 6 =

02C33E0298E5B09D7877B018F9D68DB149655D1A8395588882D5957D06BD527FEADFB41773211F
75287B306E30E5D0A3

The resulting 384-bit Message Digest is

MD =

02C33E0298E5B09D7877B018F9D68DB149655D1A8395588882D5957D06BD527FEADFB41773211F
75287B306E30E5D0A3

Appendix C: Khichidi-1 (512 bits) Examples

A1. Let the message be the 24-bit ASCII string ‘abc’, which is equivalent to the binary string
01100001 01100010 01100011.

Round 1 =
11A6986D16A3FDB669C7D84C0DC5498495D5E774F6C18574F80C22837235371919F8B7EA96F7087
F8A4D9EB5EBF3B80A5D2413AC0B195CE0FE43151464DDA2F3
Round 2 =
6E289E3AAC3A684472A913D441BF0340F0DB8430DCB8138742E325D79C4495384814A63F0933EC
13BF34580F486876DAB497981FDB4081A9D3BD441A5BA91A86
Round 3 =
C66024866B75CAEF77B0869957E4DABA83D66AEC6A8F5AC9B74A96287329E1E7E41885D4BC657
C4277F79DFD48D80A350496635BE3F4B12F3688897D19087CE5
Round 4 =
CC737D914DD47B38AE2A494676A26CF0BA26875758DA8F3C597D979726B37E40903855C4057337
4B57BDD4A89E8AF98A2F3FD4464E32D0CD80799DA78EF0492B
Round 5 =
BB13BB90C805B864EBC5326C1B631633B147CF74F83F63D4AD87E2BEFC90EC9F865F79EE4053F
F516C71741654EAB3F3511D02BDF17C0EE4CAA9B812FC0E49AA
Round 6 =
7246E7420B4C164C6FBB4CA38F7AC70094C383C3AEEE8F4E4C46CEA341406FF0EEDBC80C8A1C
D6267360D8E38A0979A73A6D1EBFA7D380E563C885E93EBDE249

The resulting 512-bit Message Digest is

MD =
7246E7420B4C164C6FBB4CA38F7AC70094C383C3AEEE8F4E4C46CEA341406FF0EEDBC80C8A1C
D6267360D8E38A0979A73A6D1EBFA7D380E563C885E93EBDE249

A2. Khichidi-1 (384-bits) ("The quick brown fox jumps over the lazy dog")

Round 1 =
1E26C9D732A98929D3C3961D2A5B9C1A46382EAC100641A00916BCE77828D69EA413EC5FAB3D
1BC2A5D343DCE34FD544843F7E041EF45204292965D7043A3E45
Round 2 =

F7507DAEEEF674C528E6E16365BB515D7455FE4D6A82FE1A4208B3D14CDA618E809B872A4F416
1D0A74FE0079E0CA06E5A90A61BFCBB10CE34CF2C61E6696374

Round 3 =

E507F7E251747472E806C2E1EDE4A2F4B24A511D349C81B9876AA17730EFE53172D5B39931895A7
15A9102B8759EA14B7DDC01E7A2E22BBA57DD7025D18A02F1

Round 4 =

36BCD12A8023A38096B9F6DD3B4502EACBB83D82DFD1C22EF8C01ECD14580133609C99EB4D99
A4EB7636C2D99C40DEFF8B6B17C0CD85801C70C0D031D5AB1FA7

Round 5 =

6ABE645E1AE874CB6185400B1F0095B1895996D96BF6CE0E0FF27E49FEAFB86F98AFA373BCC7E
267B598A23CE7B5D4D97ABC6DA1C38E3F4A31C6617EF1EA70EA

Round 6 =

CB516489F37B2D9C26ADFD9E99B06E855AC72903FD416DFD0216C5732228E503D961C853B2E80
B70BDB6C5178AAF377BB8B755AC8F06BCBC3B31E04849DEB59B

The resulting 512-bit message digest is

MD =

CB516489F37B2D9C26ADFD9E99B06E855AC72903FD416DFD0216C5732228E503D961C853B2E80
B70BDB6C5178AAF377BB8B755AC8F06BCBC3B31E04849DEB59B

Even a small change in the message will (with overwhelming probability) result in a completely different hash; for example, changing d to c:

Khichidi-1 (512-bits) ("The quick brown fox jumps over the lazy cog")

Round 1 =

45A1D51967532D1D76874DADB13DD55A62DF825CC7F7A884422E290429433FC7C8636F427333EE
9B99AB3D95C8C2A29ABF65616F225E7632E94FBD980958B953

Round 2 =

5DE4D6416EDF0F84A89FC386F107F607CF39BDEE2B374DAFD35A4852B7CBA0CA2170CC376FF3
9AF3D52583683927CB6C60045F09C83BA5896BA4BF4693700551

Round 3 =

BB4396B915DE4ED1B6F4BFC554644A9B4DE111BD57F58498C12BA6EC0C65C3DFCE37BF63DFC
DCFF755C7808ECECB78489321AA3A009580094CDD5AF2F131ED12

Round 4 =

0697FCD5CE8F8E34A946C94D1B809E78E07A35CA3BC16C6C2574F210F11E0AF2F5AF6A9D99A04
F6FF8C8A88D04542B9A5F8BCE5404A7C8B66F6C4387DAE8840A

Round 5 =

70960FB60AE6E02BE0D9A3DFE1509CC5B8784CDE461C9A2BD41EDC5917AF72EE08766FD9033B

26B8F4E40C2D0FFF7D0E2CC77308836D68E88ED8FEF2A48AFE80

Round 6 =

E4EFEE4354B2A82360946FF3931E891F5968CA9609B19E66AFE9B1B5DCFF19C957BA1B14DE6B2

C03A02DA7BB4C8752415AAD3F31F755AC86DE8FEA74A619A174

The resulting 512-bit message digest is

MD =

E4EFEE4354B2A82360946FF3931E891F5968CA9609B19E66AFE9B1B5DCFF19C957BA1B14DE6B2

C03A02DA7BB4C8752415AAD3F31F755AC86DE8FEA74A619A174

A3. The hash value of Null string / Zero-length message (“”) is

Round 1 =

2CE0F04589A334A68BC5FBFB62E128923681829C4D0C7BB442EEF2D763BFD429E84B4CBA4E919

2ACB6644803D2FFB819D5207EBA92359F7F300A6432429B1D58

Round 2 =

15EF0B9A1BE74FE0093B0A89BA08103E39BE7C2BB8670869E765C73C95640CF610A38284047FED

903FFA33158C7FD643BA272B0D3DADCDEF8997AAED025B0151

Round 3 =

F5F7689D7860CD26BD11BB7590A880528D445316F610DFC2F7BE287C14E50FD8521E1E47A70CEF

C5C1FED2E73AF2D0838A4DFE7C02E127D9CBCCCF70BA81393A

Round 4 =

E32C64D5A04EC1193727033304DFD23AA99AD444D6A8FD78C7B8A8C707BC29324B05EBF6785A

3FD2E4FA3A51432BD34892D94B7024C84E31417FACF1963FEEC9

Round 5 =

2B37807F39DA7CBC57B4A285C8637CC249C014BCF2743520E5C714F9EE12C322269F4B04C7CDA

24BF63FBE5B3EC48E19912962EA04C2B9526FD46112BF485495

Round 6 =

9C12F8F5E2DC3A315F958297FFDFDC81914FCDED093E8EAD7D02E9832B9EBC06BDBE009C0555

30F510E7504FFED2ADFBF9734CFA4950C0949C2BAAA88E30D41A

The resulting 512-bit Message Digest is

MD =

9C12F8F5E2DC3A315F958297FFDFDC81914FCDED093E8EAD7D02E9832B9EBC06BDBE009C0555

30F510E7504FFED2ADFBF9734CFA4950C0949C2BAAA88E30D41A

Appendix E: Entropy tests

The following statistical tests [8] provide random properties for Khichidi-1 algorithm.

E1. For the message = abc, Khichidi-1 (224-bits) gives the hash output EEFFAC89F540FC49BE3C06339EFE255782ADC610F8C28C053C853C59 which is passed into Entropy tests. One of the statistical tests is Monte Carlo, which is helpful in determining random properties.

- Entropy = 3.813095 bits per byte.
- Optimum compression would reduce the size of this 58 byte file by 52 percent.
- Chi square distribution for 58 samples is 1142.55, and randomly would exceed this value 0.01 percent of the times.
- Arithmetic mean value of data bytes is 57.3103 (127.5 = random).
- Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).
- Serial correlation coefficient is 0.286293 (totally uncorrelated = 0.0)

E2. For the message = abc, Khichidi-1 (256-bits) gives the hash output 99C9B9744FC81C819B0A1813CD8940FED69DB5B14CFA07F0BA591DFC3C02E87 2 which is passed into Entropy tests. One of the statistical tests is Monte Carlo, which is helpful in determining random properties.

- Entropy = 3.906370 bits per byte.
- Optimum compression would reduce the size of this 66 byte file by 51 percent.
- Chi square distribution for 66 samples is 1175.21, and randomly would exceed this value 0.01 percent of the times.
- Arithmetic mean value of data bytes is 57.2576 (127.5 = random).
- Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).
- Serial correlation coefficient is 0.298133 (totally uncorrelated = 0.0).

E3. For the message = abc, Khichidi-1 (384-bits) gives the hash output 8E9F3D720491B3799249B550A0F5AF47D15570D92004BCF423224C7D82242722BE

9740467748086FB06C73921B292E78, which is passed into Entropy tests. One of the statistical tests is Monte Carlo, which is helpful in determining random properties.

- Entropy = 3.849476 bits per byte.
- Optimum compression would reduce the size of this 98 byte file by 51 percent.
- Chi square distribution for 98 samples is 1902.98, and randomly would exceed this value 0.01 percent of the times.
- Arithmetic mean value of data bytes is 55.1429 (127.5 = random).
- Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).
- Serial correlation coefficient is 0.227110 (totally uncorrelated = 0.0).

E4. For the message = abc, Khichidi-1 (512-bits) gives the hash output 7246E7420B4C164C6FBB4CA38F7AC70094C383C3AEEE8F4E4C46CEA341406FF0 EEDBC80C8A1CD6267360D8E38A0979A73A6D1EBFA7D380E563C885E93EBDE2 49, which is passed into Entropy tests. One of the statistical tests is Monte Carlo, which is helpful in determining random properties.

- Entropy = 3.922122 bits per byte.
- Optimum compression would reduce the size of this 130 byte file by 50 percent.
- Chi square distribution for 130 samples is 2240.95, and randomly would exceed this value 0.01 percent of the times.
- Arithmetic mean value of data bytes is 58.0231 (127.5 = random).
- Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).
- Serial correlation coefficient is 0.185394 (totally uncorrelated = 0.0).

Acknowledgment

To complete the project work, the inventor Dr. N. Vijayarangan is thankful to Mr. Sitaram Chamarty, Dr. Rajagopal Srinivasan, Dr. A. Sarangarajan, Mr. R. Ramkumar, Mr. Chalamala Srinivasa Rao, Mrs. Priya V. Som, Ms. Divya Vyas and Ms. Mallika Choudhuri (of TCS) for their contributions in source code reviewing, testing, documentation, copyright and patent issues. Finally, the inventor is thankful to Dr. M. Vidyasagar, TCS, Head of Innovation labs, Hyderabad for his encouragement to this project work and also Mr. K. Ananth Krishnan, Chief Technology Officer, TCS for his approval to NIST hash competition