

# New implementation of the abstract design for non-iterative hash functions with 1024 digest length NIHF-1024

**Abouchouar Abdallah, Fouzia Omary**

Department of Computer Science, Faculty of Computer Science, Mohammed V University in Rabat, Rabat, Morocco

## Article Info

### Article history:

Received Jul 18, 2021

Revised Jun 14, 2022

Accepted Jul 3, 2022

### Keywords:

Authentication

Cryptographic hash function

Domain extension

National institute of standards and technology

Security

## ABSTRACT

The non-iterative hash function design is a domain extension that uses a novel concept to fill a gap in existing cryptographic designs. This design avoids the structural issues that plagued prior paradigms, particularly those applying the Merkel-Damgård design, by presenting a novel approach. Non-iterative hash function design, on the other hand, is an abstract idea that necessitates the specification of the internal transformation in order to test and experiment it. To achieve this goal, this article describes the algorithm that implements the internal transformation based on the main characteristics of the new model. The avalanche effect's results are provided in the experiment section, as well as a preliminary validation stage of the national institute of standards and technology (NIST's) cryptographic algorithm validation program (CAVP), which employs a set of test vectors to verify algorithm accuracy and implementation errors.

*This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.*



## Corresponding Author:

Abouchouar Abdallah

Department of Computer Science, Faculty of Computer Science, Mohammed V University in Rabat

Avenue des Nations Unies, Agdal 10000, Rabat, Morocco

Email: abdollah.abouchouar@gmail.com

## 1. INTRODUCTION

Cryptographic primitives, especially hash functions, are used to secure data and privacy. They're essential for ensuring high levels of security in digital transactions, web applications, and anywhere else security is necessary. Both theoretical and proved collision attacks, now benefit from recent advancements in digital platforms and architectures. As consequence, current algorithms are in a critical position to counter serious coming threats. National institute of standards and technology (NIST) has released a request for candidate method submissions for secure hash algorithm (SHA-3) [1], a new cryptographic hash algorithm that out performs or improves on (SHA-2) [2]. At the end of this competition, NIST designated the (SHA-3) official standardized algorithm [3], also known as Keccak-based on wipe function design [4]. However, as demonstrated in previous studies [5], [6], there are still possible concerns because practically all algorithms inherit the attributes of a classic domain extension Merkle-Damgård [7], [8], which have been found to be vulnerable [9], [10]. Moreover, several surveys discussed the vulnerabilities in authentication methods [11], internet of things (IoT) systems [12]–[14], and network design [15], and called for creative solutions to address these issues.

Thus, the non-iterative hash function (NIHF) design emerges as an abstract concept that avoids construction issues and suggests an alternative to the current paradigms. However, implementing the internal transformation of NIHF is a critical step to allow experiments and assess the security level. Following this approach, the current article describes a new implementation of the NIHF design, presents the experimentations and results through basic validation test programs.

This article is organized as: in the second section, a preliminary about domain extensions and hash functions. The third section describes and discuss the implementation security level. The fourth section presents the experimentations, test results and comments on the security level. Finally, section five concludes this work and open perspectives to the next contributions.

## 2. PRELIMINARY

### 2.1. Cryptographic hash functions

Hash functions map input strings of arbitrary length to a fixed length strings output, called hash-values, message digest or fingerprint. Theoretically a cryptographic hash function must satisfy the following properties [16]: i) Ease of computation  $\rightarrow$  for a kwon function  $H$  with input  $x$ ,  $H(x)$  is easy to compute; ii) One-way function  $\rightarrow$  for each  $y=H(x)$  in range of  $H$ , using “ $y$ ” it is computationally infeasible to find  $x$  in the domain of  $H$ ; iii) Preimage resistance  $\rightarrow$  for a given digest  $y$  of  $H$ , it is infeasible to find  $x$  with  $H(x)=y$ ; iv) 2nd preimage resistance  $\rightarrow$  for a given  $x$ , it is infeasible to find  $x'$  such that  $x' \neq x$  and  $H(x')=H(x)$ ; and v) Collision resistance  $\rightarrow$  it is infeasible to find separate  $x$  and  $x'$  such that  $x' \neq x$  and  $H(x')=H(x)$ . In practice, due to the birthday theory [16] not all these properties are satisfied.

### 2.2. Construction design

The construction design, domain extension, or operation mode paradigm is used to overcome the complexity of processing a non-fixed length message and producing a fixed output length to manage the internal compression transformation of any hash function. Merkle, Damgård [7], [8] is the most well-known construction design in this sector; it was the first solution proposed independently by Damgård [7] and Merkle [8] in this subject. This structure allows for iterative behavior, which has an influence on the design of a number of popular hash algorithms.

The Merkle-Damgård construction [7], [8] operates with arbitrary-length messages subdivided into fixed-length input blocks. During the process, an initialization vector (IV) is employed. The internal compression algorithm iteratively processes the current message block as an input and the result of the previous iteration as an IV block, as shown in Figure 1.

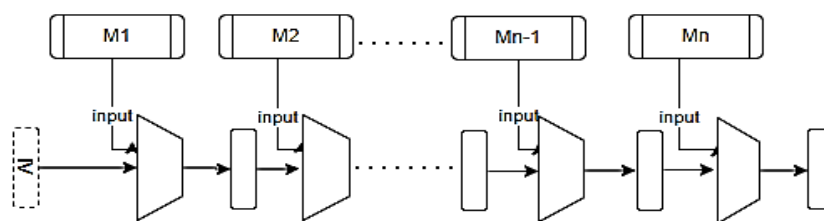


Figure 1. The Merkle-Damgård construction

Through this concept a resistant hash function can be reduced to a resistant internal compression function. An iterated construction with an internal collision resistant compression function can be extended to a collision resistant hash function [16]. Because of its efficiency, most well-known structures, such as Maurer and Tessaro [17], empirical mode decomposition (EMD) [18], random-oracle XOR (ROX) [19], and hash iterative framework (HAIFA) [20], are based on the Merkel-Damgård [7], [8], and were designed to improve and remedy the Merkle-Damgård vulnerabilities [7], [8].

## 3. DESCRIPTION OF THE NEW HASH FUNCTION

NIHF [5] is an abstract concept that requires implementation to the internal transformation, as described in the abstract. In fact, the construction properties are very important to define the behavior of the whole implemented hash function; however, the way the internal transformation is described also defines the security level and strength of the entire function. The internal implementation of the NIHF design is described in this section.

### 3.1. Applied construction design

Almost cryptographic hash functions take advantage from operational mode to simplify the way how to process a non-fixed input length and generate fixed length output, here we call back some specific

characteristics of the non-iterative domain extension [5] implemented with the new hash function: i) There is no initialization vector to handle; ii) The internal transformation has a role to extend the input blocks, not to compress them as in the classic paradigm; iii) Each input block is processed separately; iv) This model can provide a pseudo parallel processing based on separate input blocks; v) It applies a sequential XOR addition, involving all extended outputs; and vi) This construction design joins a pseudo parallel behavior to an iterative one. The Figure 2 illustrates the design characteristics:

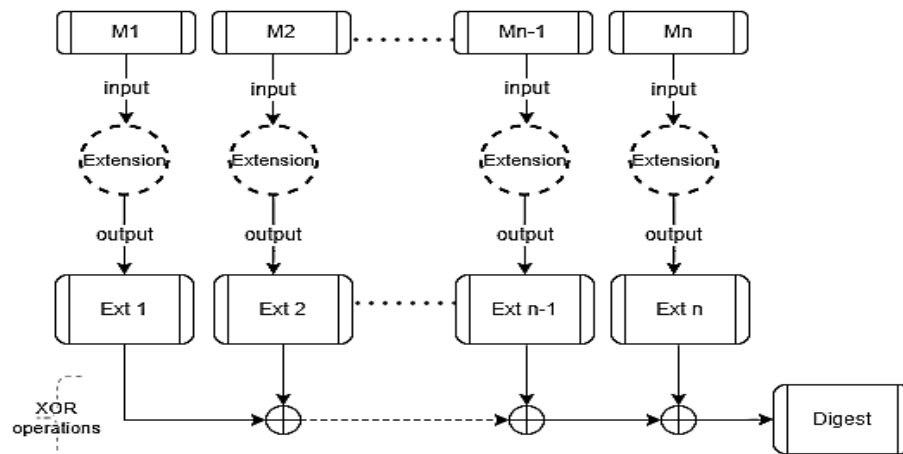


Figure 2. Non-iterative domain extension

### 3.2. Internal transformation description

The internal transformation affects the level of security expected from any cryptographic hash function. Additionally, randomness is a recommended characteristic that enhance the security level. To match with these characteristics, the proposed internal transformation is based on random block construction as well as possible following the next: i) The random input block content intervenes in the internal operations process; the initial and extended part of the output block is a set of operations that handle the random input to produce new values. ii) Avoiding use of the initialization vector increases the randomness and prevents any content manipulation. iii) By extending the input block this function prevents any local collision attack, then to ensure the compression role a sequential XOR is applied on all output blocks. As illustrated in Figure 3, this internal transformation processes a fixed size input block “Mi” and generates an extended output block “A”. Sections 3.3.2 and 3.4.2 go over the actions (transformation 1,2,3, and 4) in further detail.

### 3.3. Algorithm and implementation

A description of the algorithm can be found in this section. The following values define this hash function: the input block size is 512 bits, and the output block size is 1024 bits. (NIHF-1024), which stands for non-iterative hash function with a digest of 1024 bits, is the name of this version.

This algorithm consists of three operations. Beginning with a pre-processing phase in which the input message is padded and prepared to be subdivided into input blocks. The internal transformations can then be executed in one of two options, sequential input-blocks extension and XOR operation, or parallel input-blocks extension followed by XOR operation involving all extended input-blocks. Finally, a 1024-bit digest of the compressed block is generated.

The sequential input-blocks extension and XOR operation are implemented in this paper's experiment. For the upcoming version research, the parallel mode will be implemented. The processing mode is explained in the following subsections.

#### 3.3.1. Pre-processing phase

The purpose of the padding process is to make the overall length of the padded message a multiple of 512 bits. NIHF-1024 computes at least two blocks of 512 bits to construct the final digest. The following describes how this padding should be done, based on the (SHA-256) padding algorithm [2]: i) To the original message, append a single '1' bit; ii) append as many '0' bits to the original message as feasible, so that the sum of the padded '0' and the original message length +1+64 is a multiple of 512; and iii) append the 2-word representation of the original message length value to the padded message's last 64 bits.

### 3.3.2. Block extension process

One of the innovative main concepts that constitutes the NIHF design is processing the input blocks individually. In fact, without initialization vectors, implementing the extension transformation is a real challenge. The illustration in Figure 2 shows the proposed algorithm to overcome this obstacle. Each input block has a four-step cycle to complete, as shown in Figure 3. Each step specifies a number of iteration rounds and arithmetic operations. The following sub-sections describe these steps. The following defines some notations: i) Let  $M_i$  the current 512 bits input block which is made up of 16 Integers of 32-bits size; ii) Let  $A_1$  and  $A_2$  the first and second half of the 1024 bits output block; iii) Let  $\text{rotRight}(\text{val}, x)$ : performs a right rotation on “val” with “x” shifts; iv) Let  $\text{index}$  the current indexed Integer in the array block; v) Let  $\text{getRandomInputIndex}(x)$ : return a value from 0 to the input block length; vi) Let  $\text{getRandomOutputIndex}(x)$ : return a value from 0 to the output block length; and vi)  $\text{msgLength}$ : is the input length after the padding.

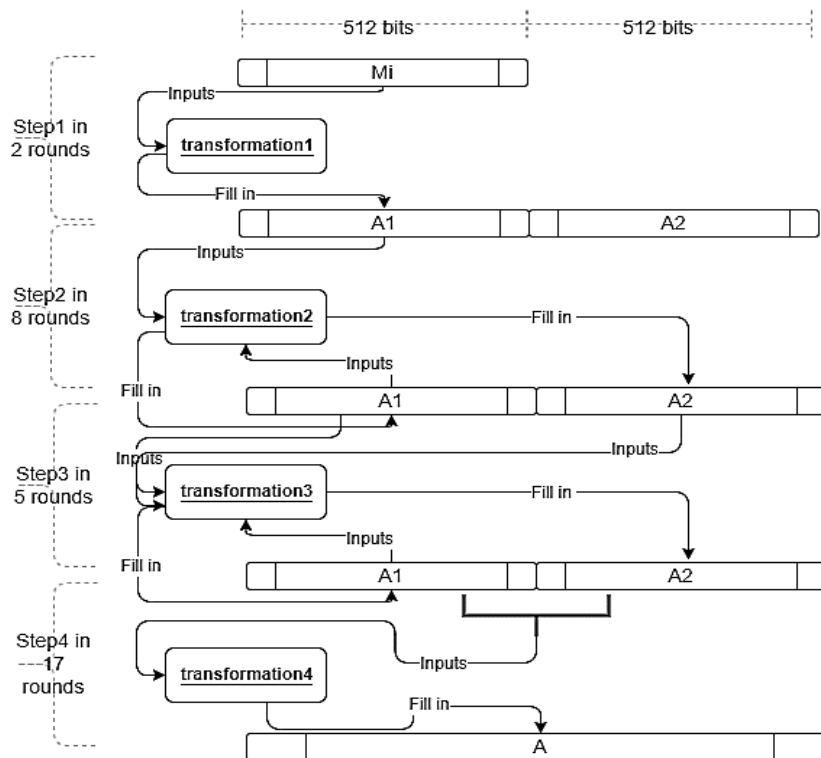


Figure 3. Internal transformation

#### a) Step1: rounds from 0 to 2

Processing ( $M_i$ ) the input block of Integers, a random index value is used to select an integer from  $M_i$ , if the integer is a zero value the (Max Integer) value is used instead, then a rotation to the right is performed with a specific number of shifts to avoid getting the same integer value. This operation is repeated for the second elected integer with a slice change in the index value, and the zero value is substituted by (Max Integer value-Magic Number), where the (Magic Number) is a binary representation that improves the exclusive addition property. The two Integers are then added together to fill the appropriate cell in ( $A_1$ ).

These operations are repeated in two rounds ( $0 \leq \text{rounds} < 2$ ) while filling up ( $A_1$ ). The purpose is to substitute any zero values for pseudo random values. Here's a quick rundown of what was said:

```
repVal1=replaceZeroValue ( $M_i$  [ $\text{getRandInputIndex}(\text{index})$ ], intMaxValue)
repVal2=replaceZeroValue ( $M_i$  [ $\text{getRandInputIndex}(\text{index}+1)$ ], intMaxValue-MagicNumber)
 $A_1$  [ $\text{index}$ ]=rotRight (repVal1, 32-index-3) + rotRight (repVal2, 32-index-2)
```

#### b) Step2: rounds from 2 to 10

The purpose of the second step is to make additional pseudo random alterations without using the input block ( $M_i$ ). So, in ( $A_1$ ), it consists of performing a rotation to the right on the current selected integer

from ( $A_1$ ), then elect another integer by a random index value in ( $A_1$ ) and apply a rotation to the right on this integer. The old integer value in (index) is substituted by the XOR operation over the two new integers.

```
val1=rotRight (A1 [index], 32-index-1)
val2=rotRight (A1 [getRandInputIndex (A1 [index])], 32-index-3)
A1[index]=val1 ^ val2
```

( $A_1$ ) integers are then transformed and used to fill ( $A_2$ ). Except rotation operation, almost the same instructions are done.

```
val1 = A1 [getRandInputIndex (index)]
val2 = A1 [(A1 [index])% (Mi.length)]
A2[index] = val1 ^ val2
```

### c) Step3: rounds from 10 to 15

In step 3, the (reverseNumber) method is introduced, which reverses the digits order in the number value, in addition to rotation, XOR addition, and pseudo random selection operations. This combination increases the output value's variability and randomness. The reverse technique, in general, generates new values rather than using predefined values as initial vectors and registries. The steps for carrying out the instructions are outlined below.

```
In (A1) :
val1 = reverseNumber(A1[index])
val2 = A1[getRandomInputIndex(msgLength-(index+1))]
A1[index] = val1 ^ val2
In (A2):
val1 = rotRight( reverseNumber(A2[getRandomOutputIndex(index)]), index)
val2 = A2 [getRandomInputIndex(index)]
A2 [index] = val1 ^ val2
```

### d) Step4: rounds from 15 to 32

The goal of the fourth step is to spread the pseudo random behavior across the entire output block (A), which increases the avalanche effect property. Here, the same instructions are performed:

```
val1 = rotRight(A[getRandomOutputIndex(index)],index )
val2 = rotRight(A[getRandomOutputIndex(index+1)],index+1)
val3 = rotRight(A[ getRandomOutputIndex(index+2)],index+2)
A[index] = val1 + val2 + val3
```

#### 3.3.3. XOR operation process

The extended output blocks generated in the previous step should be aggregated to form one final output block with a fixed length, even if the construction design does not use iterative and chained input blocks processing. In this stage, the XOR addition is used to sum the extended output blocks and to take use of the one-time pad property, which ensures a high level of complexity and ambiguity, resulting in increased randomness and system security [21].

Due to the XOR commutativity and associativity properties, the extended output blocks can be handled in two implementation mode, which can be either sequential sub-block extension and XOR operation, or parallel sub-block extension and XOR operation. While the implemented mode is sequential block extension and XOR operations, a binary XOR operation is performed over the current extended block and the preceding one. The last result output block is the final hash, or digest.

### 3.4. NIHF-1024 pseudo-code

As presented in previous paragraphes, the algorithm consists of three operations. Beginning with a pre-processing phase, the internal transformations and the XOR operation involving all extended input-blocks. This pseudo-code section describes the main function algorithm which is the sequential input-blocks extension and XOR operation mode, then details the pseudo-code of the extension operation and the XOR operation. The following is a view on the algorithms covering the important phases of NIHF-1024.

### 3.4.1. The main function: NIHF-1024 sequential extension and XOR operations

The main function manages and combines the extension and XOR addition operations, starting with the padding procedure and input block preparation. As previously stated, the implemented mode in this case study is the sequential mode, in which the two operations are processed in order. In other words, except for the first extended input block, each extended input block in a loop instruction is subjected to an XOR addition with the previously extended input block. Pseudo-code of NIHF-1024 as shown in Algorithm 1.

#### Algorithm 1. Pseudo-code of NIHF-1024

---

```

Input   : msg           //the input message in byte array
           inputBlockSize //the input byte block size parameter
           outputBlockSize //the output byte block size parameter
Output  : digest        //the final extended and xored block


---


Begin

  Let extBlock1 ← array()
  Let extBlock2 ← array()
  // Pre-processing : a padding process to the input message
  Let msgArrayBlocks ← padding (msg, inputBlockSize, outputBlockSize)
  Let paddedInputMsgLength ← msgArrayBlocks.length() * 32 //count in bits
  // Extend the initial bloc
  extBlock1 ← extendInputBlock(msgArrayBlocks[0], inputBlockSize,
                                outputBlockSize, paddedInputMsgLength)
  // A loop on the blocks from the second one, sequential extension and XOR
  operations
  for each block in msgArrayBlocks
    extBlock2 ← extendInputBlock(block, inputBlockSize,
                                  outputBlockSize, paddedInputMsgLength)
    extBlock1 ← xorOperation(extBlock1, extBlock2, inputBlockSize,
                              outputBlockSize)
  End for
  digest ← extBlock1
  return digest

```

**End**

---

### 3.4.2. Extension operation: extendInputBlock

In this implementation, the extension operation serves a consistent role; it enlarges the input blocks and gives the processing steps a random behavior. It does not rely on initial vectors IV, but rather on random input values. It performs 32 rounds of binary, logic, rotation, and substitution operations in the round ranges supplied. Each of the four round ranges is described in pseudo-code as shown in Algorithm 2.

#### Algorithm 2. Pseudo-code of extendInputBlock

---

```

Input   : block           //the current input block
           inputBlockSize   //the input block size parameter
           outputBlockSize  //the output block size parameter
           paddedInputMsgLength //the input message length after padding
Output  : extBlock        //the extended block


---


Begin

  Let integerInputBlockSize ← 16 //input Integer(32bits) block size parameter
  Let integerOutputBlockSize ← 32 //output Integer(32bits) block size parameter
  Let integerMagicNumber ← 0xAAAAAAAA //binary representation of this number
  enhance the exclusive addition
  Let integerMAXValue ← 0x7fffffff //Max int value in 32bits representation =
  0x7fffffff]
  Let integerBitSize ← 32 //number of bits used to represent an int value in
  two's complement binary form
  // in 32 rounds, apply the binary operations according to the round range
  // ranges: 1=<round>2; 2=<round>10; 10=<round>15 or 15=<round>32
  for round from 1 to 32
    if round < 2 then
      for i from 0 to inputBlockSize
        if (integerBitSize-(i+3)) % integerBitSize == 0 then
          temp1 ← | ((bloc[ |i % integerInputBlockSize| ] == 0 ?
                      integerMAXValue :
                      bloc[ |i % integerInputBlockSize| ])
                    >> (integerBitSize-(i+3)+1)) |
        Else
          temp1 ← | ((bloc[ |i % integerInputBlockSize| ] == 0 ?
                      integerMAXValue :
                      bloc[ |i % integerInputBlockSize| ])
                    >> (integerBitSize-(i+3))) |

```

---

```

End if
if (integerBitSize-(i+2)) % integerInputBlockSize == 0 then
    temp2 ← | ((bloc[ |i+1| % integerInputBlockSize ] == 0 ?
    (integerMAXValue-integerMagicNumber): bloc[ |i+1| %
    integerInputBlockSize ])
    >> (integerBitSize-(i+2))+1) |
Else
    temp2 ← | ((bloc[ |i+1| % integerInputBlockSize ] == 0 ?
    (integerMAXValue-integerMagicNumber): bloc[ |i+1| %
    integerInputBlockSize ])
    >> (integerBitSize-(i+2))) |
End if
extendArray[i] ← temp1 + temp2
End for
End if
if round >= 2 && round < 10 then
    // on the first part of the extended block, process operations (J<
    integerInputBlockSize)
    for j from 0 to integerInputBlockSize
        extendArray[j] ← (extendArray[j] >> (integerBitSize-(j+1))) ^
        ( extendArray[ | extendArray[j] | %integerInputBlockSize] >>
        (integerBitSize-(j+3)))
    End for
    //on the second part of the extended block, process operations
    (integerInputBlockSize =< i< integerOutputBlockSize)
    for i from integerInputBlockSize to integerOutputBlockSize
        extendArray[i] ← (extendArray[ |i| % integerInputBlockSize ] ) ^
        (extendArray[ |extendArray[i]| %
        integerInputBlockSize ])
    End for

    if round >= 10 && round < 15 then
        for j from 0 to integerInputBlockSize
            extendArray[j] ← reverseNumber(extendArray[j]) ^
            extendArray[ | paddedInputMsgLength - (j+1)
            | % integerInputBlockSize]
        End for
        for i from integerInputBlockSize to integerOutputBlockSize
            extendArray[i] ← (reverseNumber(extendArray[ | i %
            integerOutputBlockSize | ] ) >> i) ^
            extendArray[ | i % integerInputBlockSize | ]
        End for
    End if
    if round >= 15 && round < 32 then
        for i from 0 to integerOutputBlockSize
            extendArray[i] ← (extendArray[ | i % integerOutputBlockSize | ] >>
            i )
            + (extendArray[ | i+1 % integerOutputBlockSize | ]
            >> i+1)
            + (extendArray[ | i+2 %
            integerOutputBlockSize | ] >> i+2)
        End for
    End if
End for
return extendArray
End

```

### 3.4.3. Xor operation over extended blocks: xorOperation

The XOR operation is a binary XOR addition over two extended input blocks. This method is used to deal with input blocks extension, replacing and avoiding the classic block chaining strategy while simultaneously giving high levels of ambiguity and cryptanalysis protection. As a result, the output digest is compressed and has a set length at the end of the processing.

## 4. RESULTS AND DISCUSSION

For evaluating cryptographic hash functions, specific tests and validation programs are recommended, such as the avalanche effect [22], [23], the statistical test suite (STS) [24], and the cryptographic algorithm validation program (CAVP) [25] tools from NIST. Some of these programs, particularly the avalanche effect and the test vectors provided by CAVP-check the behavior while processing the test vectors, are discussed in this section. The remaining CVAP phases and STS will be carried out in future research initiatives. The following are the system configurations used in the execution environment: i) Implementation with JAVA

jdk7; ii) Laptop processor Intel® Core™ i5-7440HQ CPU @ 2.80 GHz 2.80 GHz; iii) Memory of 16 GB; and iv) Windows 10 (64 bits) system.

#### 4.1. Avalanche test

The avalanche test was introduced by Feistel with the data encryption standard (DES) S-Box [22]. The avalanche effect is a desired characteristic that strengthens the hash function security. The concept is “if a function is to satisfy the strict avalanche criterion (SAC), then each of its output bits should change with a probability of one half”, i.e. if any single bit input is complemented, each of the output bits changes with a probability of one half [23].

Calculating the Hamming distance for a hash function  $F$  and input ( $m$ ), i.e. measuring the difference between the outputs  $F(m)$  and  $F(m')$ , where ( $m'$ ) is an input after complementing a random single bit in the input ( $m$ ), is part of the validation test procedure. To check this property with NIHF-1024, we generate a random input message of 128 byte, then process it in a loop of 10,000 rounds, by complementing a random single bit each round and comparing the current output with the prior output. At least perform a Hamming distance comparison on 10,000 output messages.

For a 1024-bit output digest, the Hamming distance between two consecutive digests should theoretically be close to 512. The Hamming distance with the NIHF-1024 avalanche effect is near to the expected value. According to the data gathered throughout the experiments, the Hamming distance varies between 465 and 560. This metric prompted us to conclude that the generated outputs are practically independent of the inputs, which is a desirable property. Figure 4 shows the avalanche impact of NIHF-1024 in the first 5,200 rounds.

#### 4.2. Test vectors

The NIST CAVP conducts validation testing of certified cryptographic algorithms and their constituent components by using any of the accredited laboratories to test the algorithm implementations [25]. The experiments in this paper are limited to using only the test vectors provided by the CAVP to informally verify and detect pointer problems, insufficient space allocation, improper error handling, incorrect function behavior, or potential collision detection; the remainder of the CAVP procedure is not covered. The constituents of the test vectors, as well as the test results, are presented in this section.

The secure hash algorithm-3 validation system SHA-3VS-which outlines the validation testing criteria for the SHA-3 family-[26] provides the specifications and design of test vectors. The test vectors, as well as the Short messages test files, Long messages test files, and pseudo-randomly generated messages (Monte Carlo) test files, are all related to SHA-3 requirements. All are in the bits and bytes encoding format and may be found in the test vectors section [25].

Only the short and long input files will be considered in the test. The (Monte Carlo) test files have been postponed till later. The NIHF-1024 process, generates and records the resulting digest by parsing the content files and for each input message defined in. Throughout the experiment, the NIHF-1024 algorithm was updated and reviewed to remedy implementation issues until all test vectors were correctly processed. To summarize the test vectors' findings, we've included two tables, Table 1. and Table 2., that show used file name, input message count, detected error, collision detection and execution time for both Byte and Bit input files. According to the given results in Table 1 and Table 2, the NIHF-1024 passed these test vectors without error or collision detection. To be clear, the purpose of this experiment is to examine the NIHF-1024 implementation and behavior, not to compare it to the SHA-3 implementation.

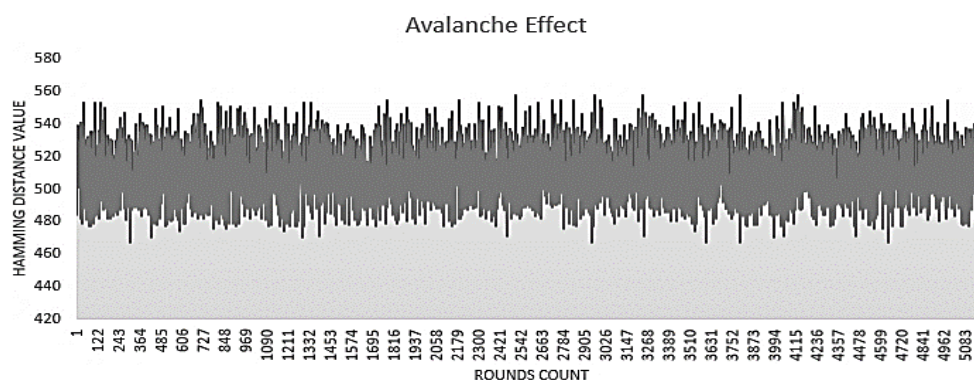


Figure 4. Avalanche test results



Table 1. Test vector input files, Byte oriented message

Byte Input file	Input message count	Error detection	Collision detection	Time (ms)
SHA-3_224LongMsg.rsp	100	0	0	509
SHA-3_224ShortMsg.rsp	145	0	0	10
SHA-3_256LongMsg.rsp	100	0	0	467
SHA-3_256ShortMsg.rsp	137	0	0	11
SHA-3_384LongMsg.rsp	100	0	0	355
SHA-3_384ShortMsg.rsp	105	0	0	6
SHA-3_512LongMsg.rsp	100	0	0	243
SHA-3_512ShortMsg.rsp	73	0	0	6

Table 2. Test vector input files, Bit oriented message

Bit Input file	Input message count	Error detection	Collision detection	Time (ms)
SHA-3_224LongMsg.rsp	100	0	0	559
SHA-3_224ShortMsg.rsp	1153	0	0	83
SHA-3_256LongMsg.rsp	100	0	0	455
SHA-3_256ShortMsg.rsp	1089	0	0	77
SHA-3_384LongMsg.rsp	100	0	0	346
SHA-3_384ShortMsg.rsp	833	0	0	53
SHA-3_512LongMsg.rsp	100	0	0	243
SHA-3_512ShortMsg.rsp	577	0	0	27

## 5. CONCLUSION AND PERSPECTIVES

To recap, we've stated that it's safe to look for a new concept in cryptographic constructions in order to avoid future security problems. Thus, the pillar of this work was to think outside the box about how to innovate a new cryptographic design, first by resolving structural issues with a new paradigm, then proposing a new implementation to the new concept -which is the main subject of this paper, and finally challenging the potential of this new solution. By the way, we detailed and discussed the algorithm implementation in the previous sections, which is based on random and independent processing and so adheres to the paradigm properties. We set an objective to challenge the implementation of NIHF-1024 with several validation test programs, such as the avalanche effect and NIST vector tests, at this stage of the continuous work. The test results were promising, prompting us to try more validation programs and explore new options in parallel or distributed architectures with new variants of NIHF-1024 in order to uncover its potential. Now, we have a perspective to challenge more experimentations and more environment configurations in a continuous progression to achieve a considerable status for this new approach of cryptographic hash functions.




## REFERENCES

- [1] NIST, "Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family," *Federal Register Notices*, vol. 72, no. 212, pp. 62212–62220, Nov. 2007.
- [2] Q. H. Dang, "Secure Hash Standard," Gaithersburg, MD, Jul. 2015. doi: 10.6028/NIST.FIPS.180-4.
- [3] M. J. Dworkin, "SHA-3 standard: permutation-based hash and extendable-output functions," Gaithersburg, MD, Jul. 2015. doi: 10.6028/NIST.FIPS.202.
- [4] G. Bertoni, J. Daemen, and G. Van Assche, "The KECCAK Reference. Submission to NIST (Round 3)", 2011. Accessed:2022. [Online]. Available: <http://keccak.nokeon.org/Keccak-reference-3.0.pdf>.
- [5] A. Abouchouar, F. Omary, and K. Achkoun, "New concept for cryptographic construction design based on noniterative behavior," *IAES International Journal of Artificial Intelligence (IJ-AI)*, vol. 9, no. 2, pp. 229–235, Jun. 2020, doi: 10.11591/ijai.v9.i2.pp229-235.
- [6] A. T. Hashim, A. M. Hasan, and H. M. Abbas, "Design and implementation of proposed 320 bit RC6-cascaded encryption/decryption cores on altera FPGA," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 10, no. 6, pp. 6370–6379, Dec. 2020, doi: 10.11591/ijece.v10i6.pp6370-6379.
- [7] I. B. Damgård, "A design principle for hash functions," in *Advances in Cryptology — CRYPTO '89 Proceedings*, Springer New York, 1990, pp. 416–427.
- [8] R. C. Merkle, "One way hash functions and DES," in *Advances in Cryptology — CRYPTO '89 Proceedings*, Springer New York, 1990, pp. 428–446.
- [9] Wang, X., Yin, Y.L., Yu, H. (2005). Finding Collisions in the Full SHA-1. In: Shoup, V. (eds) *Advances in Cryptology – CRYPTO 2005*. CRYPTO 2005. Lecture Notes in Computer Science, vol 3621. Springer, Berlin, Heidelberg. [https://doi.org/10.1007/11535218\\_2](https://doi.org/10.1007/11535218_2).
- [10] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1," in *Advances in Cryptology – CRYPTO 2017*, Springer International Publishing, 2017, pp. 570–596.
- [11] T. Mehraj, M. A. Sheheryar, S. A. Lone, and A. H. Mir, "A critical insight into the identity authentication systems on smartphones," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 13, no. 3, pp. 982–989, Mar. 2019, doi: 10.11591/ijeecs.v13.i3.pp982-989.
- [12] M. Imdad, D. W. Jacob, H. Mahdin, Z. Baharum, S. M. Shaharudin, and M. S. Azmi, "Internet of things: security requirements, attacks and counter measures," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 18, no. 3, pp. 1520–1530, Jun. 2020, doi: 10.11591/ijeecs.v18.i3.pp1520-1530.
- [13] R. Chetan and R. Shahabaddkar, "A comprehensive survey on exiting solution approaches towards security and privacy requirements of IoT," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 8, no. 4, pp. 2319–2326, Aug. 2018, doi: 10.11591/ijece.v8i4.pp2319-2326.




- [14] A. H. Aly, A. Ghalwash, M. M. Nasr, and A. A. A.-E. Hafez, "Formal security analysis of lightweight authenticated key agreement protocol for IoT in cloud computing," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 24, no. 1, pp. 621–636, Oct. 2021, doi: 10.11591/ijeecs.v24.i1.pp621-636.
- [15] V. M. S. and M. C. Patil, "Reviewing effectivity in security approaches towards strengthening internet architecture," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 9, no. 5, pp. 3862–3871, Oct. 2019, doi: 10.11591/ijece.v9i5.pp3862-3871.
- [16] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC Press, 1996.
- [17] U. Maurer and S. Tessaro, "Domain extension of public random functions: Beyond the birthday barrier," in *Advances in Cryptology - CRYPTO 2007*, Springer Berlin Heidelberg, 2007, pp. 187–204.
- [18] M. Bellare and T. Ristenpart, "Multi-property-preserving hash domain extension and the EMD transform," *Advances in Cryptology - ASIACRYPT 2006, Lecture Notes in Computer Science*, vol. 4284, pp. 299–314, 2006, doi: 10.1007/11935230\_20.
- [19] E. Andreeva, G. Neven, B. Preneel, and T. Shrimpton, "Seven-property-preserving iterated hashing: ROX," in *Advances in Cryptology - ASIACRYPT 2007*, Springer Berlin Heidelberg, 2007, pp. 130–146.
- [20] E. Biham and O. Dunkelman, "A framework for iterative hash functions - HAIFA," *IACR Cryptology ePrint Archive*, p. 278, 2007.
- [21] A. I. Salih, A. M. Alabaichi, and A. Y. Tuama, "Enhancing advance encryption standard security based on dual dynamic XOR table and MixColumns transformation," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 19, no. 3, pp. 1574–1581, Sep. 2020, doi: 10.11591/ijeecs.v19.i3.pp1574-1581.
- [22] H. Feistel, "Cryptography and computer privacy," *Scientific American, a division of Nature America, Inc.*, vol. 228, no. 5, pp. 15–23, May 1973, doi: 10.1038/scientificamerican0573-15.
- [23] R. Forrié, "The strict avalanche criterion: Spectral properties of boolean functions and an extended definition," in *Advances in Cryptology - CRYPTO '88, 8th Annual International Cryptology Conference*, Springer New York, 1990, pp. 450–468.
- [24] L. E. Bassham *et al.*, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," National Institute of Standards and Technology, 2010. doi: 10.6028/nist.sp.800-22r1a.
- [25] P. D. O'Reilly, K. Rigopoulos, G. Witte, and L. Feldman, "2017 annual report: NIST/ITL cybersecurity program," Gaithersburg, MD, Sep. 2018. doi: 10.6028/NIST.SP.800-203.
- [26] L. E. B. III and T. A. Hall, "The secure hash algorithm validation system (SHA VS)," 2004.

## BIOGRAPHIES OF AUTHORS



**Abdallah Abouchouar**    is a Ph.D. student in computer science at Mohammed V University in Rabat, where he works as a researcher in the Intelligent Processing & Security of Systems (IPSS) lab. He specializes in cryptography and security systems, with an emphasis on cryptographic hash function concepts and implementations. His research topic's goal is to propose and develop a novel cryptographic hash function concept. He can be contacted at email: abdallah.abouchouar@gmail.com.



**Fouzia Omary**    is a Professor of Higher Education at - Mohammed V University in Rabat. Having obtained her master's degree in applied mathematics and her DEA in computer science, she joined the Department of Mathematics of the Faculty of Sciences of Rabat, as a teacher. Three years later, she obtained her 3rd cycle Doctorate in Computer Science in the field of compilation to obtain the position of Assistant Professor In Computer Science. She evolved her professional career and became a Professor of Higher Education by obtaining the State Doctorate in Computer Science in the context of Cryptography in 2006. She also obtained the certificate "Blockchain Developer Mastery Award (2018)" issued by IBM. From 2012 to 2016 she was Director of the LRI research structure. And from 2016 she is Head of the "Intelligent Processing & Security of Systems" (IPSS) Research Structure. She can be contacted at email: omaryfouzia@gmail.com.