RM70 : A Lightweight Hash Function

Benardi Widhiara, Yusuf Kurniawan, and Bety Hayat Susanti, *Member, IAENG*

***Abstract*—One of the key challenges in cryptography is creating a secure yet efficient design of cryptographic algorithms. Recently, lightweight cryptography has drawn a strong interest of cryptographers, with the most recent research concentrating on constructing the design of block ciphers. In contrast, there are considerably fewer publicly released proposals for the design of lightweight hash functions, indicating that this field is still far from being thoroughly explored. In this work, we present a new lightweight hash function based on sponge construction with ARX-SPN structure permutation. We evaluated the proposed algorithm on a number of parameters, including randomness, security (focusing on the algorithm's preimage resistance, second preimage resistance, and collision resistance properties), and performance in terms of time and memory consumption.**

***Index Terms*—cryptography, hash function, IoT, lightweight, sponge construction.**

I. Introduction

THE development of technology has rapidly grown over the past decades. This can be seen with the shift of human civilization to Industry 4.0. On top of that, with the introduction of the Metaverse, it is entirely possible for human civilization to enter that era in the upcoming years. That surely has an impact on the increased demand for secure applications, considering almost all applications currently existing are still using insecure channels such as the internet to exchange data. The services of confidentiality, integrity, and authentication may provide security guarantees for applications, such as RFID and sensor networks. The said services can be achieved by utilizing cryptographic primitives, including the secret-key cipher and hash function.

The hash function has a pivotal role in enforcing security on applications. It can be used to check message integrity and/or enable verification processes. Many seemingly trivial yet critical daily circumstancing, including online banking transactions, digital currency, OTP generation, password protection, as well as lock pattern on mobile phones will not operate as smoothly and as securely without hash function. These features usually utilize dedicated ciphers that require large memory, high-power consumption, and high implementation cost. This is certainly an issue considering the currently emerging technologies, like small computing devices and IoT, run in a constrained environment, deeming that large consumption costs are no longer an option.

Lightweight cryptography is accounted as the answer to the issue because it is designed to ensure the efficiency of an algorithm (i.e., only requiring low memory, power consumption, and cost) without ignoring its security. Therefore, a well calculated trade-off between efficiency and security is inevitable. Many lightweight ciphers have been proposed by researchers and analysts in the last decades. Several proposed lightweight stream ciphers are Grain v1 [1], MICKEY v2 [2], and Trivium [3]. Moreover, lightweight block ciphers such as PRESENT [4], CLEFIA [5], SIMON and SPECK [6], SIMECK [7], DLBCA [8], and SLIM [9] have also come to the surface. Further, works have been done on several lightweight hash functions as part of the lightweight cipher. In [10], the hashing mode operation that uses lightweight block cipher PRESENT with Hirose and Davies-Meyer construction is well described, while in [11], they proposed a lightweight hash function with Merkle-Damgard builder construction. In addition, the use sponge construction as the builder of a lightweight hash functions PHOTON, QUARK, SPONGENT, and NEEVA are thoroughly explored in [12], [13], [14], and [15] respectively.

Though there have been many proposed lightweight ciphers, there is still no standard algorithm that can be used globally [15]. In this paper, we proposed a secure sponge-based lightweight hash function algorithm with an ARX-SPN structure permutation named RM70, which is more efficient in terms of processing time than the most efficient sponge-based algorithm, SPONGENT and requires considerably low memory consumption. This paper is organized as follows: Section II describes the fundamental theory underlining the ideas we are presenting in this paper. The design of the proposed algorithm is described in Section III. Sections IV, V, and VI present the randomness evaluation, security evaluation, and performance evaluation of the RM70 algorithm, respectively. This paper ends with a conclusion which is stated in Section VII.

II. Fundamental Theory

A. Lightweight Cryptography

Lightweight cryptography is one of the fairly new branches of cryptography, resulting from the massive development of cryptographic primitives that can be implemented on various devices with constrained power, such as the RFID and Wireless Sensor Network ([16], [17]). Generally, lightweight cryptography is divided into two categories: symmetric and asymmetric [17]. Over the last decade, a lot of research has yielded symmetric lightweight cryptography. The algorithms PRESENT [4], CLEFIA [5], SIMON and SPECK [6],

Manuscript received March 27th, 2022; revised November 22th, 2022.

This research is supported by the Research and Development Unit of Institut Teknologi Bandung, West Java, Indonesia and Politeknik Siber dan Sandi Negara, Bogor, West Java, Indonesia.

B. Widhiara is a postgraduate student in the School of Electrical Engineering and Informatics, Institut Teknologi Bandung, Indonesia. e-mail (23220093@std.stei.itb.ac.id, benardi.widhiara@bssn.go.id)

Y. Kurniawan is an Assistant Professor of School of Electrical Engineering and Informatics, Institut Teknologi Bandung, Indonesia. e-mail (yusufk@stei.itb.ac.id)

B. H. Susanti is an Assistant Professor of Cryptography Engineering Department, Politeknik Siber dan Sandi Negara, Bogor, Indonesia. e-mail (bety.hayat@poltekssn.ac.id, bety.hayat@bssn.go.id)

SIMECK [7], DLBCA [8], and SLIM [9] are all lightweight block ciphers that have emerged steadfastly over the years. Research on lightweight stream ciphers also yielded some algorithms such as Grain v1 [1], MICKEY v2 [2], and Trivium [3]. Furthermore, many have also proposed lightweight hash function algorithms such as, ARMADILLO [11], PHOTON [12], QUARK [13], SPONGENT [14], and NEEVA [15].

B. Sponge Construction

The sponge construction is a simple iteration construction that processes input to certain length variable into an output with arbitrary length, which is based on a transformation or permutation function with fixed length in a state with fixed number of bits [14],[18]. In its structure, sponge construction has several parameters such as rate r, capacity c, and output n. The width of a sponge construction is called state s with $s = r + c$. Sponge construction processed in three phases as follows:

1) Initialization Phase
   The padding process and the division of messages into message blocks are carried out in this phase. Padding is only performed on messages that are not multiples of the rate because the input message can be of any length. The padded message must be a multiple of rate in length, so the message can be divided into message blocks.
2) Absorbing Phase
   The first message block is XOR-ed with r most significant bits of the initial state. Afterward, the state is processed into the permutation function of the sponge construction. This process is carried out continuously until the last message block. The output of this phase will be the input for the squeezing phase.
3) Squeezing Phase
   The hash value generation occurs in the squeezing phase by taking the first r-bit of the state as partial output, interleaved with the application of the permutation function f until an n-bit output is generated. The n-bit output is obtained by merging all partial hash values.

III. The Design of RM70

RM70 is a lightweight hash function with a sponge construction builder. This algorithm processes arbitrary input and produces output with an 88-bit length. The parameters used in this algorithm include 96-bit state s with an 8-bit rate r and an 88-bit capacity c. In order to gain a clear picture of the RM70 algorithm, the general scheme is presented in Fig.1.

RM70 processes message input into a hash value in three phases:

1) Initialization Phase
   The initialization phase consists of the padding process and dividing the message input into message blocks. Message input, X can be of arbitrary length and the padded message should be a multiple of 8-bit. That is why padding on messages which are not a multiple of 8-bit is required. The padding process is done by adding one bit 1 followed by k bit 0 at the end of the message. Before being processed in the absorbing phase, the padded message will be divided into message blocks of 8-bit length.
2) Absorbing Phase
   This phase starts by XOR-ing the first block message to the 8 most significant bits of the initial state (i.e., a 96-bit state of all zeros) and then the permutation function f is applied to the state. This process is done repeatedly until the last message block. The permutation function f is the twenty times composition of permutation g which can be formulated as $f = g_1 \circ g_2 \circ \ ... \ \circ g_{20}$. The function g is an ARX-SPN structure permutation consisting of a 2^8-modulus sum layer, substitution box layer, mix word layer, and rotation bit layer.
3) Squeezing Phase
   The first 8-bit of the state is taken as the partial hash value, interleaved with the application of the permutation function f. This process is done repeatedly until 88-bit output hash value is generated. The final hash value H is obtained by merging all partial hash values generated previously. The final hash value can be formulated by $H(X) = h_1 \parallel h_2 \parallel \cdots \parallel h_{10} \parallel h_{11}$.

A. Permutation Function

The permutation function f of RM70 algorithm consists of function g which operates twenty times. The function g comprises several layers such as 2^8-modulus sum layer, substitution box layer, mix word layer, and rotation bit layer. The permutation function f of RM70 is presented on Fig. 2.

1) 2^8-Modulus Sum Layer

The 2^8-modulus sum layer (as shown on Fig. 3) is the first layer in function g. This layer will operate 8-bit of state s with 8-bit of a round number rN. Thus, to process a state with the size of 96-bits, there will be twelve 2^8-modulus sum operations that run simultaneously. Each iteration of function g uses different round number value. Hence, for one permutation function f, there will be twenty different round number values. Table I presents the round number value for each iteration of function g.

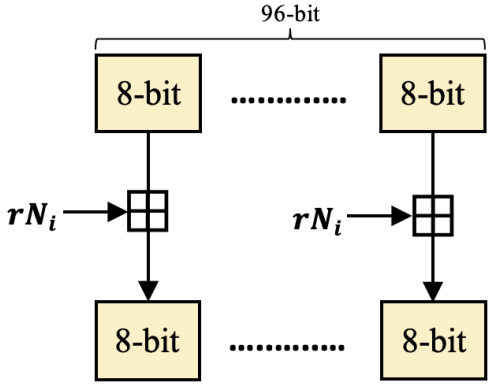

Fig. 3. 2^8-modulus sum layer

2) Substitution Box Layer
   The substitution box (sB) is a function that maps input bits into determined output bits. The RM70 algorithm uses substitution box AES [19] which maps 8-bit input into 8-bit output. Therefore, twelve substitution box operations will be conducted simultaneously to process a 96-bit state s. The substitution box used in RM70 is presented on Fig. 4.

TABLE I
ROUND NUMBER VALUE

Iteration	Round Number *rN* (in hexadecimal)
1	8940fc56e0df0f7d80821fe7
2	51e64cb2dc26db6210d108a0
3	b97f2fcb3e987b4def3c7b1e
4	40d11dd80a72cd0da6650aae
5	ef9ec7a9bc506b0cda83aa81
6	42bde00aa5ce0d217aa1d504
7	ac220a881cf0249a9c22ce60
8	f7038a6bb68576220c316506
9	f6b87607d0b40ca4b90e5594
10	f6855096f6eb2a40f8b97371
11	855bd9516201c7fe464e6fa1
12	ccf178dae6ecbba10ad15143
13	dd7d0baa1a8696f99065cc36
14	ad7d188e37e3804182193c78
15	7821f9cf5c15e99cf1bea557
16	4e1808b5b002e30221208020
17	fb22a00e9a75eabcc4576674
18	4ef5ed7b84d56f49419555fe
19	4405bdb750344a2926a35878
20	b4dabdf3db27cd49122ca1dd

3) Mix Word Layer

The 96-bit state *s* will be divided into three groups of words *gw* of 32-bits so there will be gw_1, gw_2, and gw_3. Each group-word consists of four words *w* of 8-bit which is denoted by $w_{(i,j)}$ with $i = 1, 2, 3, 4$ and $j = 1, 2, 3$. Each word in group-word will be XOR-ed one to another. The second word will be XOR-ed with the first word $(w_{1,j} \oplus w_{2,j})$, the third word will be XOR-ed with the first word $(w_{1,j} \oplus w_{3,j})$, and the fourth word will be XOR-ed with the first word $(w_{1,j} \oplus w_{4,j})$. Mix word layer (mW) is shown in Fig. 5.

4) Rotation Bit Layer

After obtaining the output of the mix word layer, the 8 most significant bits of it is rotated to the left. This process sums up the rotation bit layer, as shown in Fig. 6.

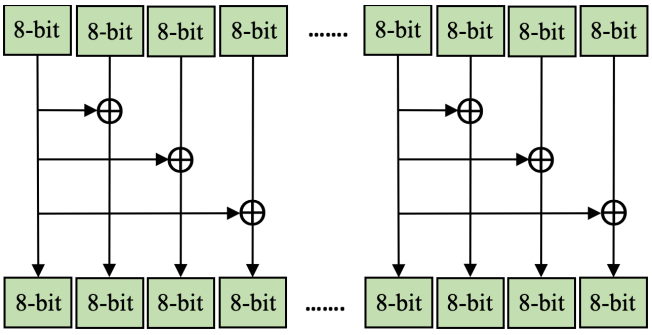

Fig. 5. Mix word layer

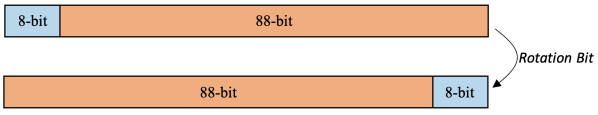

Fig. 6. Rotation bit operation

B. Design Rationale

The design of the RM70 algorithm has weighed various considerations, from the selection of building construction, the selection of function type, and the operations composing the function of the sponge construction. The sponge construction is chosen to be the building construction of the RM70 algorithm because of these following reasons:

1) Sponge construction is a simple construction and has a good level of security compared to other hash function constructions such as the Merkle-Damgard, the Hirose, and the Davies-Meyer constructions [20].
2) Sponge construction with one permutation or transformation function can be used to generate hash values of any length [20].
3) Sponge construction has a flexible nature where various levels of security of an algorithm can be achieved by adjusting to the parameters rate r, capacity c, and output n [13].
4) The computation on sponge construction does not require memory to store message blocks and feedforward as the case of the Davies-Meyer and the Hirose constructions [13],[14].
5) Sponge construction can be used to achieve the full security level of hash function [13], [21].
6) Sponge construction can be used for various functions such as a deterministic pseudorandom number generator, message authentication codes (MAC), salted hashing, plain hashing, and stream encryption [20],[22].

Sponge construction has at least one function, be it a permutation or transformation function. The iterated permutation function was chosen as a function in the sponge construction algorithm RM70 with the consideration that the iterated permutation function is difficult for attackers to exploit in cryptographic attacks. In addition, the computation of the iterated permutation function does not require memory to store messages [20].

The permutation function f which is built from the Addition-Rotation-XOR (ARX) and Substitution Permutation Network (SPN) is intended to achieve efficiency in processing but still possesses the security of the function. The ARX structure is known as an efficient and fast structure, but its security is not so well fortified compared to other structures such as SPN or Feistel [23]. The permutation function *f* is also designed to process 8-bit input with the aim of speeding up the processing.

The 2^8-modulus sum operation between state *s* and the round number *rN* is intended to change the pattern of the input bits so that no bits with a constant pattern are found to achieve confusion property. Similar to the 2^8-modulus sum operation, substitution box is used to achieve good confusion property. Other than that, the AES substitution box is used because it has adequate security and resistant toward cryptanalysis attack such as differential cryptanalysis, linear cryptanalysis, impossible differential cryptanalysis, and others [19]. Furthermore, the low correlation between the input and the output bits of the AES substitution box is another good reason as to why we chose it.

Other than confusion property, an algorithm must also satisfy the diffusion property. To achieve this, a mix word operation is performed which processes the input in word groups as shown in section III.A. However, the output of this layer has a weakness for which the first word of each word group has a fixed value. To overcome this weakness, rotation bit operation is therefore conducted. This operation will rotate the 8 most significant bits of mix word layer output. Moreover, all components of the permutation function f use basic operation that will make it easier to write the source code both in software and hardware applications.

C. Test Vector

We performed a test vector for every intermediate step of RM70 hashing process. A 28-bit message (X = 0x1234567) will be hashed to generate 88-bit hash value. Table II shows the test vector of the RM70 algorithm.

TABLE II
TEST VECTOR OF RM70

Initialization Phase		
X = 0x1234567 $X + PAD(X)$ = 0x12345678 X_1 = 0x12, X_2 = 0x34, X_3 = 56, X_4 = 78		
Absorbing Phase		
Input	**Process**	**Output**
0x120000000000000000000000	Absorb 0	0xa24be532d286cd786a1a5ca5
0x964be532d286cd786a1a5ca5	Absorb 1	0xc4d5036be13ee0422ff8f697
0x92d5036be13ee0422ff8f697	Absorb 2	0x8f0afe2ef2c3aecaf05f3c0c
0xf70afe2ef2c3aecaf05f3c0c	Absorb 3	0xc8dac54a8a1cb906b64fe64f
Squeezing Phase		
Input	**Process**	**Output**
0x**c8**dac54a8a1cb906b64fe64f	Squeezing 0	0xca3b9b3177db4ffd0a80c10f
0x**ca**3b9b3177db4ffd0a80c10f	Squeezing 1	0x41e9d515f779b492f86ddd13
0x**41**e9d515f779b492f86ddd13	Squeezing 2	0xa9f6fb02b8a306c15bbe0eec
0x**a9**f6fb02b8a306c15bbe0eec	Squeezing 3	0xbc6803a559d3c0c9c3bba207
0x**bc**6803a559d3c0c9c3bba207	Squeezing 4	0xa8aa52e9f9fd3ec3b59fbb20
0x**a8**aa52e9f9fd3ec3b59fbb20	Squeezing 5	0x2a2b8307283ec9322d3dad80
0x**2a**2b8307283ec9322d3dad80	Squeezing 6	0xca8d68428ac2e390c6d55849
0x**ca**8d68428ac2e390c6d55849	Squeezing 7	0xea31f8a07b35503fd97fa5d9
0x**ea**31f8a07b35503fd97fa5d9	Squeezing 8	0xdf8eafa1c2e04c1e911acf14
0x**df**8eafa1c2e04c1e911acf14	Squeezing 9	0x58c1d3ca3ca00f94d7bef4e6
0x**58**c1d3ca3ca00f94d7bef4e6	Squeezing 10	0x8124348a7be7855284c31cc4
Hash Value		
$RM70(X)$ = **0xc8ca41a9bca82acaeadf58**		

IV. RANDOMNESS EVALUATION

Randomness is one of the generic properties of cryptographic primitives, including hash functions [24]. A hash function is expected to behave in the manner of random map. Besides randomness, the hash function also needs to satisfy other generic cryptographic properties such as strict avalanche criterion (SAC) and collision resistance. Hence, evaluation of the output of a hash function is compulsory. One way to evaluate the randomness of a hash function output is with a method proposed in [24].

The cryptographic randomness testing explored in [24] is a statistical randomness test evaluating a function by investigating its cryptographic properties. In this paper, we conducted three cryptographic randomness tests, namely the Strict Avalanche Criterion (SAC) Test, the Collision Test, and the Coverage Test. Each test that we performed utilized 2^{20} independent input samples of 32-bits with a significance level parameter $\alpha = 0.01$. Each test is carried out five times to avoid biased results.

A. SAC Test

The SAC test aims to evaluate whether an algorithm has satisfied the SAC property. The nature of the SAC in question is that a change of one input bit, will cause a change in the output bits with a probability of 0.5. In other words, the SAC test will evaluate an algorithm whether there is a correlation between the input bits and the output bits.

Two approaches are used to conduct an SAC test to the RM70. The first approach uses the *chi square goodness of fit* test and the second approach uses the *expected interval*. The second approach is used to determine whether or not the results of the evaluation of the first approach are coincidences, considering that the evaluation of the correlation between the input bits and the output bits is carried out thoroughly. Either the first or second approach will evaluate the SAC matrix formed during the testing process.

Table III shows that the resulting *p-value* from all five tests with the first approach is bigger than its significance level. It can therefore be concluded that the algorithm has passed the SAC test with the chi-square goodness of fit test approach. Further evaluation is carried out using the expectation interval approach. This evaluation expects each entry in the formed SAC matrix to be in the interval of [519,279, 529,297]. The results presented in Table IV show that from the first to the fifth test, there are 2,816 SAC matrix entries that are in the interval and none of them are outside the interval. Thus, the proposed algorithm passes the SAC test through the second approach.

The evaluation of the proposed algorithm for the SAC test shows that the algorithm passed both approaches of the SAC test. In other words, the algorithm has a random mapping that satisfies cryptographic properties of the strict avalanche criterion.

TABLE III
SAC TEST RESULT WITH FIRST APPROACH

SAC Test	*Chi-square* value	*p*-value
I	4.1694	0.3836
II	3.2473	0.5173
III	1.6235	0.8046
IV	6.0274	0.1971
V	6.0145	0.1981

TABLE IV
SAC TEST RESULT WITH SECOND APPROACH

SAC Test	Inside interval	Outside interval
I	2,816	0
II	2,816	0
III	2,816	0
IV	2,816	0
V	2,816	0

B. Collision Test

The purpose of a collision test is to evaluate the randomness of a function by considering the nature of the collision resistance. This collision test focuses on the number of collisions that occur on certain output bits (near collision). In other words, the collision test evaluates an input of size n through the function f and evaluates the number of collisions b found in the t-bit output.

The parameters $n = 2^{12}$ and $m = 2^{16}$ are used in conducting the test, where n is the number of messages modified by the message sample and m is the number of output bits observed for collisions. In other words, the message sample will be modified to the 12 most significant bits and collision observations will be performed on the 16 most significant bits of the resulting output.

The result of the five times collision test stated in Table V shows that none of the five tests has a p-value less than 0.01. It can therefore be concluded that the RM70 algorithm passed the collision test and has a random mapping based on the collision resistance property.

TABLE V
COLLISION TEST RESULT OF RM70

Collision Test	*Chi-square* value	*p*-value
I	6.1869	0.1856
II	8.2074	0.0843
III	6.1097	0.1911
IV	6.6557	0.1552
V	12.2189	0.0158

C. Coverage Test

The coverage test is used to evaluate the randomness of a function by observing the size of the output set formed from the input set (coverage). The hash function is said to have a random map if its coverage reaches 63%. The coverage test carried out on the proposed algorithm was performed by observing the coverage of the 12 most significant bits output formed. We observed the output generated from every modified message of the 12 most significant bits of each message sample. The results of the observations are presented in Table VI, from which we can infer that all the *p*-values of the coverage tests are greater than 0.01. This shows that the proposed algorithm has coverage of at least 63% and passed the coverage test. Thus, we can safely claim that the proposed algorithm is an algorithm with a one-way function that has a random mapping.

TABLE VI
COVERAGE TEST RESULT OF RM70

Coverage Test	*Chi-square* value	*p*-value
I	4.6160	0.3290
II	7.3611	0.1179
III	5.3991	0.2487
IV	12.6324	0.0132
V	8.4196	0.0774

V. SECURITY EVALUATION

We conducted several security evaluations on the proposed algorithm including second preimage resistance, collision resistance, and preimage resistance properties. These evaluations are meant to determine whether or not the proposed algorithm has a decent cryptographic security.

A. Theoretical Security

In an iterated variable-output length hash function (sponge-based hash function), the required resistance against attacks is expressed relative to a single-parameter called capacity [20]. An n-bit sponge-based hash with capacity c and rate r, can obtain $\min(2^{n/2}, 2^{c/2})$ of collision resistance bound and $\min(2^n, 2^{c/2})$ of second preimage resistance and preimage resistance bounds [12]. The design of RM70 uses half of the actual state size, resulting in the reduction of second preimage security bound to $2^{c/2}$ and preimage security bound to 2^{n-r}. The bound reduction is not an issue, considering the fact that full second preimage security is not a mandatory requirement in many embedded scenarios implementing lightweight hash functions. Moreover, the reduction of the preimage security bound to 2^{n-r} does not significantly affect the preimage resistance of the algorithm, considering the small value of r. Table VII shows the security bounds of RM70 and other sponge-based lightweight hash functions with similar output size.

TABLE VII
SECURITY BOUND

Algorithm	Preimage Resistance	Second Preimage Resistance	Collision Resistance
RM70	2^{80}	2^{44}	2^{44}
SPONGENT-88/80/8 [14]	2^{80}	2^{40}	2^{40}
PHOTON-80/20/16 [12]	2^{64}	2^{40}	2^{40}

B. Second Preimage Resistance

Second preimage resistance can be interpreted by a condition which is hard to find a different message with the same hash value from a given message [25]. The evaluation of the second preimage resistance property of RM70 is performed by applying the second preimage attack method proposed by [26]. We developed the attack method by changing the position of the modified message block. In [26], they used the first-second pair of message blocks to conduct a second preimage attack. In the method that we developed, we used the first-third, first-fourth, second-third, second-fourth, and third-fourth message block pairs. Thus, there are six different scenarios of the attack. All these attack scenarios were performed on 32-bits length of 10,000 independent sample messages. In general, second preimage attacks are carried out as mentioned in Algorithm 1.

After carrying out six different scenarios of second preimage attack against the proposed algorithm, we obtained the results as stated in Table VIII. The data shown in Table VIII bring us the information that no collision was found from the second preimage attack with a message modification of $2^{(n+1)/2}$ which was carried out on the proposed algorithm in attack scenarios 1, 2, 3, 4, 5, or 6. Henceforth, it can be claimed that the proposed algorithm satisfies the property of second preimage resistance.

Algorithm 1. Second preimage attack

INPUT : x_1, x_2, f
OUTPUT : x_1', x_2'
1. $S_1 \rightarrow \emptyset$
2. For $i \leftarrow 0$ to $2^{(n+1)/2}$ do
   Choose $x_1^i \notin \{x_1, x_1^0, \dots, x_1^{i-1}\}$ at random and compute $y_1^i \leftarrow f(x_1^i, x_2)$
   $S_1 \leftarrow S_1 \cup \{(x_1^i, y_1^i)\}$
   End looping
3. For $i \leftarrow 0$ to $2^{(n+1)/2}$ do
   Choose $x_2^i \notin \{x_2, x_2^0, \dots, x_2^{i-1}\}$ at random and compute $y_2^i \leftarrow f(x_1, x_2^i)$
   If $y_2^i = y_1^k$ where y_1^k is the second component of an element of S_1
   Return (x_1^k, x_2^i)
   End Looping

TABLE VIII
RESULT OF SECOND PREIMAGE ATTACK

Scenario	Blocks modified	Total Collision
S1	1 and 2	0
S2	1 and 3	0
S3	1 and 4	0
S4	2 and 3	0
S5	2 and 4	0
S6	3 and 4	0

C. Collision Resistance

Collision resistance is a condition of the hash function such that it is very hard to find two different messages with the same hash value [27]. To evaluate the collision resistance properties of the RM70 algorithm, we used Yuval's birthday attack [26]. As the name implies, this attack uses the birthday paradox approach. In general, a collision attack is carried out as mentioned in Algorithm 2.

In the attack, we modified the 20 first bits of the message

resulting in 2^{20} (1,048,576) modifications. Furthermore, the attack was carried out five times with five different pairs of message samples. The pair of message samples used in the attack is called legitimate message and fraudulent message. The five pairs of message samples are presented in Table IX.

After five collision attacks were applied, we obtained the results of the attacks as listed in Table X. The data from the attack were then analyzed to determine whether the collision attack was successfully carried out against the RM70 algorithm. From Table X, we know that there is no collision from the first, second, third, fourth, and fifth attacks.

A collision attack fails to perform against an algorithm when no collision is found. From the evaluation conducted, it can be concluded that the RM70 algorithm meets the collision resistance property with $O(2^{20})$.

Algorithm 2. Collision attack

INPUT : x_1, x_2, h
OUTPUT : x_1', x_2'

1. Generate $t = 2^{n/2}$ minor modifications x_1' of x_1.
2. Hashed every modified message x_1' and store the hash value $h(x_1')$ with the corresponding modified message.
3. Generate $t = 2^{n/2}$ minor modifications x_2' of x_2 and hashed every modified message x_2' to get hash value $h(x_2')$.
4. Check every hash value $h(x_2')$ to $h(x_1')$. Collision is found when $h(x_2') = h(h(x_1')$.

TABLE IX
MESSAGE SAMPLES OF COLLISION ATTACK

Collision Attack	Legitimate Message (in hexadecimal)	Fraudulent Message (in hexadecimal)
I	111111111111111111111	f30a75caaef2371bbb9ad9
II	2222222222222222222222	e6370ca76d364986caa6a6
III	333333333333333333333	5938a4e0c0db6296f656fc
IV	444444444444444444444	3b7fce2ebd894147ea11ca
V	5555555555555555555555	15ba3b790033a27e6234ed

TABLE X
COLLISION ATTACK OF RM70

Collision Attack	Total Collision
I	0
II	0
III	0
IV	0
V	0

D. Preimage Resistance

The meet-in-the-middle attack approach is used to evaluate the preimage resistance property. Basically, this attack tries to find the middle value by calculating backwards and forwards. The meet-in-the-middle attack is carried out in two stages, namely pre-computation (looking for the value of d_1) and matching (looking for preimage). The attack scheme on the RM70 algorithm can be seen in Fig. 7.

In carrying out a meet-in-the-middle attack, the attacker already knows the original message X, hash value $H(X) = h_1 \parallel h_2 \parallel \cdots \parallel h_i$, and the number of iterations in the squeezing phase (n/r iterations). This attack was carried out as follows:

1) Pre-computation Step

It is known that $f^{-1}(h_{i+1}, v_{i+1}) = (h_i, v_i)$ for $i = 1,2, \ldots, n/r$ in squeezing phase. With the knowledge of the value of h_i with size r bit, the probability of finding the value is ½. Thus, the search for the value (h_1, v_1) in the squeezing phase is performed by counting backwards using $2^{((n/r)-1).r} = 2^{n-r}$ v_i value which is different with $i = 1,2, \ldots, n/r$, with n is the length of the hash value, and r is the rate.

2) Matching Step

This stage is carried out in the absorbing phase after the value (h_i, v_i) from the pre-computation step. The matching step is done in the following way:

a. Choose k parameter so that $k \cdot r \geq c/2$ with c is the capacity and r the rate.
b. Generate $2^{c/2}$ messages which allows for $x_{k+2}, x_{k+3}, \ldots, x_{2k+1}$ and do the backward calculation with (h_1, v_1). Store both values as the elements of the set G_1.
c. Generate $2^{c/2}$ which allows for $x_1, x_2, \ldots, x_k$ and do the forward calculation with an initial state value $s_0 = 0^r \parallel 0^c$.
d. Compare each value from the forward calculation process with all elements in G_1 to find the same c least significant bit. The calculated value is the capacity c from x_{k+1}.
e. In x_k and x_{k+2} with the same c least significant bit, perform the XOR operation between the r most significant bit x_k with x_{k+2} to obtain the value of x_{k+1}.

Based on the attacks carried out, it is known that the precomputation step has a complexity of 2^{n-r} to obtain d_1 while the matching step has a memory complexity of $2^{c/2}$ and a time complexity of $2^{c/2}$. Furthermore, it can be inferred that the meet-in-the-middle attack has a time complexity of $\max(2^{n-r}, 2^{c/2})$ and a memory complexity of $2^{c/2}$.

VI. PERFORMANCE EVALUATION

Performance evaluation is conducted by comparing and analyzing the processing time and memory consumption of the proposed algorithm to other similar algorithms in software and hardware applications. SPONGENT-88 [14], a sponge-based algorithm with great time and memory efficiency, is chosen as a comparison in this evaluation. In order to gain a fair comparison, the performance evaluation of both algorithms is carried out under the same conditions and environments.

A. Software-Based Evaluation

There are two approaches used in this software-based evaluation. The first approach is done by measuring the algorithm's processing time with several variations of message size once, while the second approach is done by measuring the algorithm's processing time of a determined message with several iteration variations. These two approaches are carried out several times to gain a less biased result.

The results of the software-based evaluation of RM70, either with the first or the second approach show that RM70 takes less processing time compared to its competitor, SPONGENT-88. Based on the result of the evaluation using the first approach stated in Table XI, RM70 has a processing time efficiency up to 32.82% compared to SPONGENT-88. In addition, RM70 can get up to 93.03% of processing time efficiency based on the second approach evaluation as presented in Table XII.

TABLE XI
RESULT OF RM70 EVALUATION ON SOFTWARE WITH THE FIRST APPROACH

Data (in KB)	Time Needed (in second)	
	RM70	SPONGENT-88
32	0.0323	0.5189
64	0.0531	0.0777
128	0.1072	0.1527
256	0.2084	0.2915
512	0.3902	1.1771

TABLE XII
RESULT OF RM70 EVALUATION ON SOFTWARE WITH THE SECOND APPROACH

Iterations	Time Needed (in second)	
	RM70	SPONGENT-88
1,000	0.0539	0.7716
2,000	0.1038	1.5091
3,000	0.1554	2.2274
4,000	0.2037	2.9359
5,000	0.2537	3.6162

B. Hardware-Based Evaluation

The hardware-based evaluation of the proposed algorithm focuses on the processing time and memory consumption of the algorithm implemented on a small computing device, like the Arduino. The Arduino Uno R3 with its 8-bit processor is considered capable of representing small computing devices implementing a lot of lightweight cryptography algorithms. In conducting this evaluation, a 96-bit message was chosen as the input to the algorithms.

As per shown in Table XIII, RM70 has a better processing time than its competitor, SPONGENT-88, with an efficiency of 97.86%. In terms of memory consumption, RM70 requires 1% more flash memory and 25% more SRAM compared to SPONGENT-88. The amount of memory consumed in hardware is greatly influenced by the efficiency of the source code.

TABLE XIII
RESULT OF RM70 EVALUATION ON HARDWARE

Aspect	RM70	SPONGENT-88
Time	40 milliseconds	1,872 milliseconds
Flash memory	3,362 bytes (10% of max capacity)	3,074 bytes (9% of max capacity
SRAM	1,248 bytes (60% of SRAM's capacity)	732 bytes (35% of SRAM's capacity)

VII. CONCLUSION

In this research, we proposed a sponge-based lightweight hash function named RM70, which uses an ARX-SPN based permutation. This algorithm processes arbitrary input and produces output with an 88-bit length. We evaluated the proposed algorithm on a number of parameters, including randomness, security, and performance. The evaluation of randomness was performed by evaluating the randomness of the algorithm's permutation using cryptographic randomness testing (strict avalanche criterion test, collision test, and coverage test). We also conducted security evaluation in terms of preimage resistance, second preimage resistance, and collision resistance. We also performed software-based and hardware-based performance evaluations by comparing the processing time and memory consumption of the RM70 to its competitor, the SPONGENT-88. From all the evaluations that have been carried out, it is shown that the proposed algorithm has a random mapping and meets the preimage resistance, the second preimage resistance, and the collision resistance properties. In addition, the algorithm also has better processing time compared to SPONGENT-88, as well as lower memory consumption. This algorithm emerges to be a good option for a lightweight hash function, which can be used in many applications.

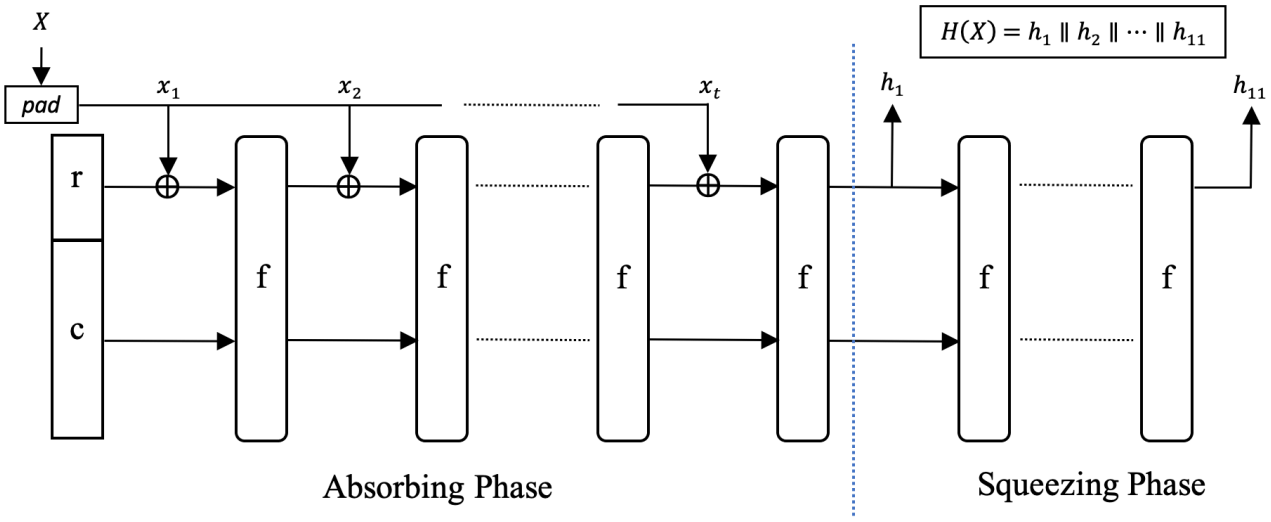

Fig. 1. General scheme of RM70 algorithm

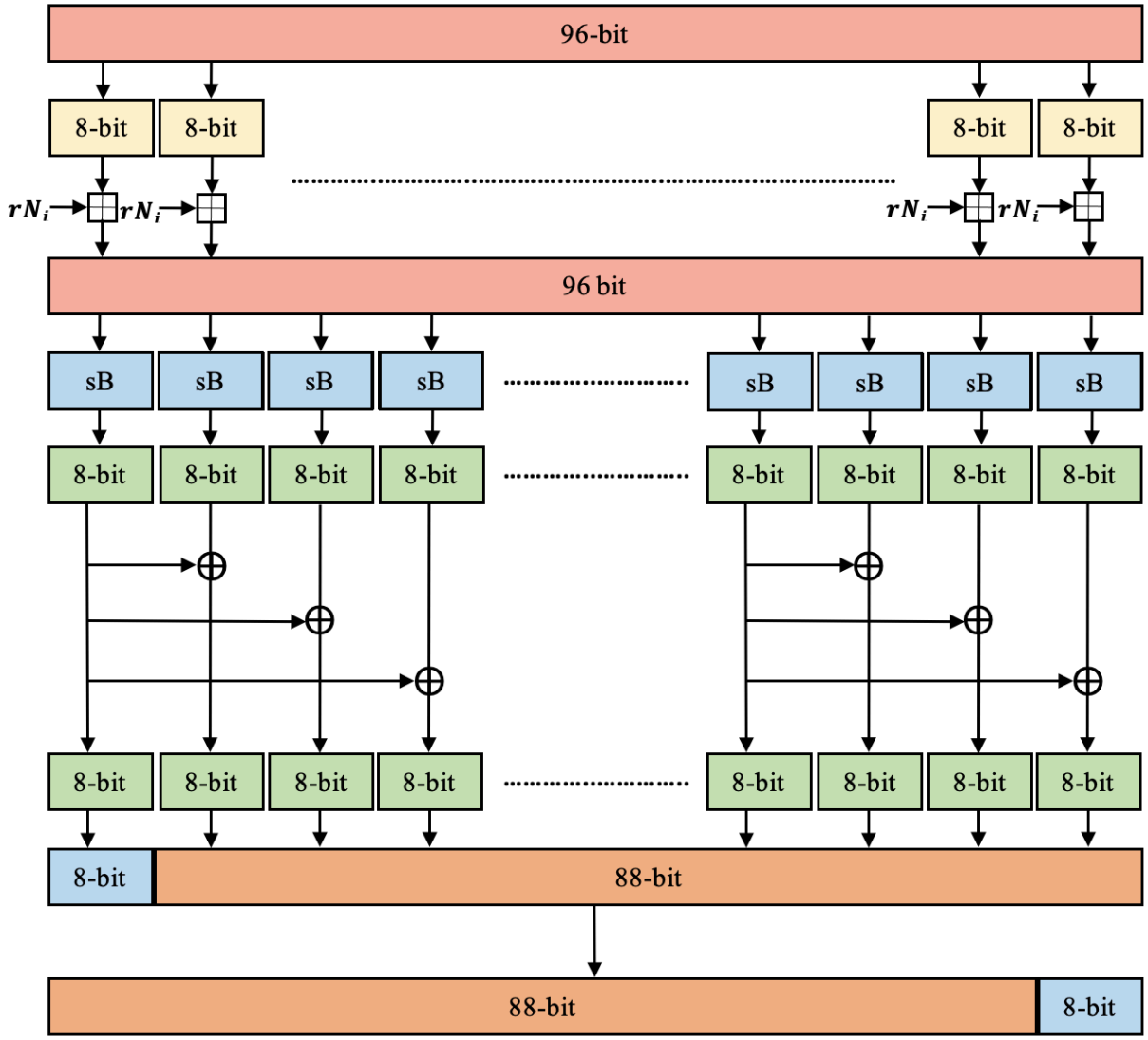

Fig. 2. Permutation function *f*

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Fig. 4. Substitution box

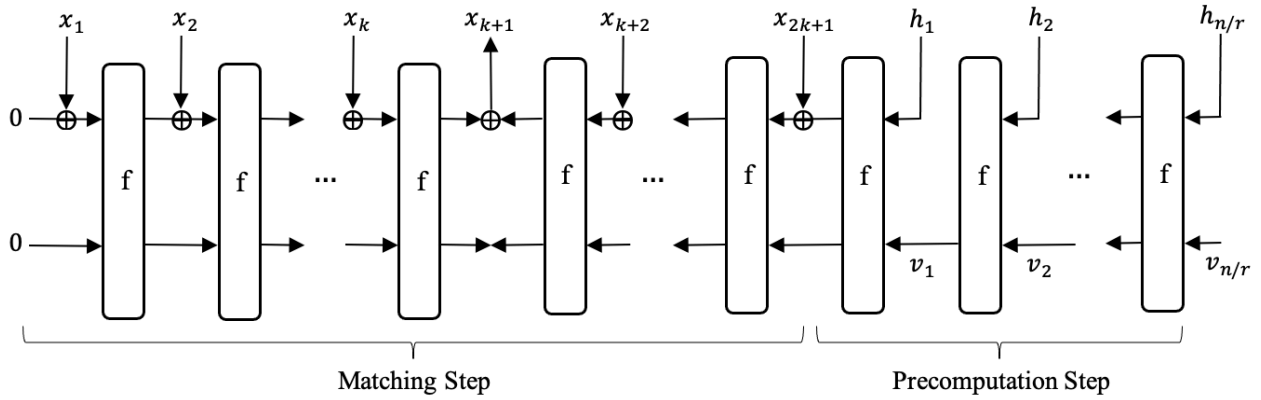

Fig. 7. Meet-in-the-middle attack scheme of RM70

REFERENCES

[1] M. Hell, T. Johansson, and W. Meier, "Grain - A Stream Cipher for Constrained Environments," *International Journal of Wireless and Mobile Computing*, vol. 2, pp. 86–93, 2007.

[2] S. Babbage and D. Matthew, "The stream cipher MICKEY 2.0," *ECRYPT Stream Cipher Project Report*, 2006. Available: http://www.ecrypt.eu.org/stream/p3ciphers/mickey/mickey_p3.pdf

[3] Y. Tian, G. Chen, and J. Li, "On the Design of Trivium," *Cryptology ePrint Archive* (Online), Paper 2009/431, 2009. Available: https://eprint.iacr.org/2009/431

[4] A. Bogdanov *et al.*, "PRESENT: An Ultra-Lightweight Block Cipher," *Cryptographic Hardware and Embedded System - CHES 2007*, pp. 450-466, 2007.

[5] T. Shirai, K. Shibutani, T. Akishita, S. Moriai, and T. Iwata, "The 128-bit Blockcipher CLEFIA (Extended Abstract)," *Fast Software Encryption – FSE 2007*, pp. 181-195, 2007.

[6] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The Simon and Speck families of lightweight block ciphers," *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1-6, 2015.

[7] G. Yang, B. Zhu, V. Suder, M. D. Aagaard, and G. Gong, "The Simeck Family of Lightweight Block Ciphers," *Cryptographic Hardware and Embedded System - CHES 2015,* pp. 307-329, 2015.

[8] S. Salim and M. Aldabbagh, "Design 32-bit Lightweight Block Cipher Algorithm (DLBCA)," *International Journal of Computer Applications*, vol. 166, no. 8, pp. 17-20, 2017.

[9] B. Aboushosha, R. A. Ramadan, A. D. Dwivedi, A. El-Sayed, and M. M. Dessouky, "SLIM: A lightweight block cipher for internet of health things," *IEEE Access*, vol. 8, pp. 203747–203757, 2020.

[10] A. Bogdanov, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, and Y. Seurin, "Hash functions and RFID tags: mind the gap," *Cryptographic Hardware and Embedded System - CHES 2008,* pp. 283-299, 2008.

[11] S. Badel *et al.*, "ARMADILLO: a Multi-Purpose Cryptographic Primitive Dedicated to Hardware," *Cryptographic Hardware and Embedded System - CHES 2010*, pp. 398-412, 2010.

[12] J. Guo, T. Peyrin, and A. Poschmann, "The PHOTON Family of Lightweight Hash Functions," *Advances in Cryptology – CRYPTO 2011*, pp. 222-239, 2011.

[13] J. P. Aumasson, L. Henzen, W. Meier, and M. Naya-Plasencia, "Quark: A lightweight hash," *Journal of Cryptology*, vol. 26, no. 2, pp. 313–339, 2013.

[14] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varici, and I. Verbauwhede, "SPONGENT: The design space of lightweight cryptographic hashing," *IEEE Transactions on Computers*, vol. 62, no. 10, pp. 2041–2053, 2013.

[15] K. Bussi, D. Dey, M. Kumar, and B. K. Dass, "Neeva: A Lightweight Hash Function," *Cryptology ePrint Archive,* Paper 2016/042, 2016. Available: https://eprint.iacr.org/2016/042

[16] S. B. Sadkhan and A. O. Salman, "A survey of lightweight-cryptography status and future challenges," *2018 International Conference on Advance of Sustainable Engineering and its Application (ICASEA)*, pp. 105-108, 2018.

[17] A. Shah and M. Engineer, "A Survey of Lightweight Cryptographic Algorithms for IoT-Based Applications," in *Advances in Intelligent Systems and Computing*, vol. 851, pp. 283–293, 2019.

[18] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche, "Cryptographic sponge functions," 2011. Available: https://keccak.team/files/CSF-0.1.pdf

[19] J. Daemen *et al.*, "The Design of Rijndael: AES-The Advanced Encryption Standard," 2001.

[20] G. Bertoni, J. Daemen, M. Peeters, and G. van Assche, "Keccak sponge function family main document," 2009. Available: https://keccak.team/obsolete/Keccak-main-2.1.pdf

[21] D. N. Gupta and R. Kumar, "Sponge based Lightweight Cryptographic Hash Functions for IoT Applications," in *2021 International Conference on Intelligent Technologies (CONIT)*, pp. 1–5, 2021.

[22] M. Borowski, "The sponge construction as a source of secure cryptographic primitives," *2013 Military Communications and Information Systems Conference - MCC*, pp. 1–5, 2013. Available: https://www.researchgate.net/publication/261051558

[23] V. A. Thakor, M. A. Razzaque, and M. R. A. Khandaker, "Lightweight Cryptography for IoT: A State-of-the-Art," 2020.

[24] F. Sulak, "Statistical Analysis of Block Ciphers and Hash Functions," Ph.D. dissertation, Dept. Cryptography, Middle East Technical University, 2011.

[25] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography. 1997.

[26] K. Jia, X. Wang, Z. Yuan, and G. Xu, "Distinguishing Attack and Second-Preimage Attack on the CBC-like MACs," *Cryptology and Network Security – CANS 2009*, pp. 349-361, 2009.

[27] B. H. Susanti, M. H. N. Ilahi, Amiruddin, S. S. Carita, "Finding Collisions in Block Cipher-based Iterative Hash Function Schemes Using Iterative Differential," *IAENG International Journal of Computer Science*, vol. 48, no. 3, pp. 634-645, 2021.