Extended Protocol Using Keyless Encryption Based On Memristors

Yuxuan Zhu
School of Informatics Computing and Cyber Systems
Northern Arizona University
Flagstaff, Arizona, United States
yz298@nau.edu

Bertrand Cambou
School of Informatics Computing and Cyber Systems
Northern Arizona University
Flagstaff, Arizona, United States
Bertrand.cambou@nau.edu

David Hely
Grenoble INP
University Grenoble Alpes
F-26000 Valence, France
David.hely@grenoble-inp.fr

Sareh Assiri
School of Informatics Computing and Cyber Systems
Northern Arizona University
Flagstaff, Arizona, United States
sa2363@nau.edu

***Abstract*—The growing interest for keyless encryption calls for new cryptographic solutions with real keyless schemes. This paper introduces an extended protocol using keyless encryption, which is hash-based and generic in cryptography. The sender side and the receiver side will be contained in the protocol. The sender will encrypt a plaintext and then send the cipher to the receiver side, and the cipher used in the protocol will be based on memristor arrays. We will use values of blocks of the plaintext to sort the cipher, which will improve the difficulty of being deciphered. Then, the receiver will receive the cipher and use it to decrypt the plaintext. The method of implementation is thoroughly detailed in this paper, and the security of the protocol is evaluated by testing random plaintexts thousands of times.**

***Keywords*—Security and privacy, Keyless cryptography, Security protocol for encryption, Security evaluation**

I. Introduction

For a long time, most of traditional cryptography uses keys to encrypt the messages. But in recent years, the interest of encryption without using the keys is growing very fast. The reason for choosing keyless encryption topic is that the key generation, key distribution, and key storage in key cryptography are incredibly complex. Also, there are many issues with the keys that can help a hacker to extract the key such as the attack based on differential power analysis. This kind of attack is practical and non-invasive; the information will be leaked through hackers analyzing power consumption to extract secret keys from a wide range of devices [1]. Also, there is another reason that can motivate researchers to pay attention to the keyless encryption, which is to make protection for the network of internet of things (IoTs). Because the IoTs have a limitation of power and memory, the long-secret keys and some strong cryptographic schemes are hard to be implemented [2]-[3]. The keyless encryption has been considered one of the best cryptographic method for protecting IoTs [4].

A keyless protocol was invented at Northern Arizona University (NAU) cyber security lab and named as "Memristors to Design Keyless Encrypting Devices." This protocol is based on keyless encrypting devices with arrays of memristors, which convert the message to encrypt into the modulation of the currents driving the cells at low power. It has used the multifactor of security such as the random number generator, password, hash functions and memristor arrays as shown in Figure 1 [5].

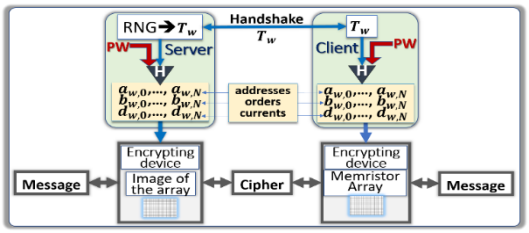

Fig. 1. Keyless encryption schemes with memristor [5]

Figure 1 shows all the protocol steps, which are the following: first, the handshake is generated between the sender side and the receiver side. Second, each party will XOR the password (PW) with a random number (RN), and the result of the XOR will be fed to the hash function to get message digest (MD). The MD will be divided into three parts: addresses, orders, and currents. The third step is to do the encryption. The encryption step has two other steps: (a) the plaintext will be divided into several blocks; (b) each block will be combined with resistance value which comes from "images" of the memristors. Both address and current will point to the cell (resistance value), which will be picked to combine with the block that comes from the plaintext, whereas the orders will help to reorder the cipher.

NAU cybersecurity lab has developed Physical Unclonable

Functions (PUFs) from ReRAM array memristors. The PUF is generated via the injection of low currents in cells of memristor arrays, and it will give new variable resistances each time different low currents are injected. These different values of resistances have been exploited to design PUFs. In this paper, the extended keyless protocol based on the protocol in "Memristors to Design Keyless Encrypting Devices" will be designed the same as the architectures of Figure 1 except the orders [5]. Instead of getting the orders from the MD, they will be provided from the plaintext itself. This extended protocol is designed to perform encryption and decryption in a safe method. The paper explains how to design it from the software perspective.

II. Environmental Setup

The protocol contains the sender side and the receiver side. The sender side will encrypt a message and then send the cipher and masks to the receiver side. Then, the receiver side will use the information received to restore the same message. It is a complicated process that our team has written it in C++ language.

First, we want to use Figure 2 to introduce the basic environment for building this protocol. As shown in Figure

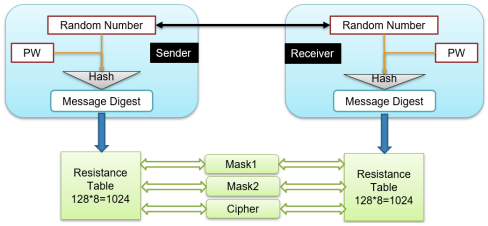

Fig. 2. Receiver side and sender side in protocol

2, both the sender side and the receiver side share the same random number (RN) which is a 64-byte long binary stream that came from our random number generator function [6]-[7]. The RN is public which means everyone can access it. At the same time, the sender side also generates a 64-byte long password (PW) which is also in binary representation, and it will be sent to the receiver side in a secure environment. "Secure" means only two sides can access it.

The cybersecurity lab at NAU provided a table of resistances, which contains experimental data generated from 128 ReRAM cells measured with negative bias between 100nA and 800nA in the 0°C to 60°C temperature range [5],[8]. This data has been saved as a CSV file and sent to our team, and both sides in the protocol will read the file and use this table as a two-dimensional array, which contains 128 rows and 8 columns. We will call it the resistance table for simplicity. In sum, the sender side and the receiver side will have the same RN, PW, and resistance table (RT). There are a few emerging themes between RN and PW. They are both 64 bytes long and in binary form. The only difference between them is that the RN is public, but the PW is private.

Then, the sender side will combine the RN and the PW using the Exclusive or (XOR) operation and will get a binary stream (64 bytes). The protocol chose XOR because other people cannot get any bit of the original message bytes. For example, every time the sender side sees a '1' in the encrypted byte, that '1' could have been generated from a '0' or a '1'. The same thing with a '0', it could come from both '0' or '1'. Therefore, not a single bit is leaked from the original message byte after using XOR logical operation [9]; this will greatly improve the level of security. After that, the sender side will use the SHA3-512 (Secure Hash Algorithm 3) function to generate a short message digest (SMD), which is 64 bytes (512 bits) long [10]. This procedure is shown in Figure 3.

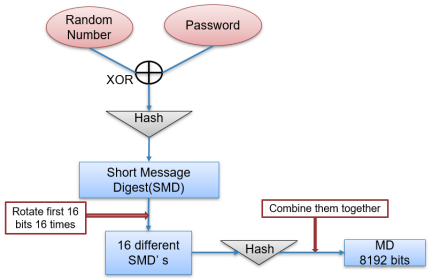

Fig. 3. How to generate SMD and MD.

After getting the SMD, the sender side will try to extend it because the length of the message digest decides how many characters can be encrypted in the protocol. To do it, the first n bits of SMD will be rotated, each time the output of rotation is going to feed the SHA3-512 Hash Function to obtain a new SMD, and finally all SMDs will be combined to get a longer message digest. In software implementation, the sender side will rotate the first 16 bits and then obtain 16 different SMDs. Finally, those 16 SMDs will create the longest message digest (MD), which will be 512 * 16 = 8192 bits.

III. Encryption

After obtaining 8192-bits MD, it's time to do the encryption in sender side. The first step in encryption architecture is shown in Figure 4.

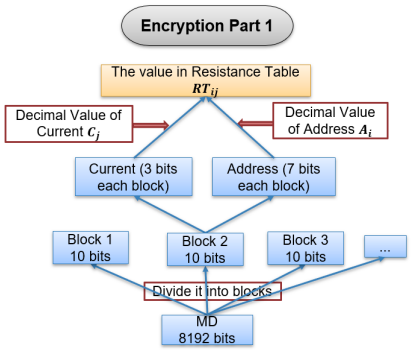

Fig. 4. Encryption Part1

At first, the MD will be divided into n blocks; each block contains "address" and "current;" the address size is 7 bits, and the current size is 3 bits. The decimal value of 7 bits will

be from 0 to 127, whereas the decimal value of 3 bits will be from 0 to 7. As a result, the decimal values of address (A_i) and decimal values of current (C_j) are used to determine the position (RT_{ij}) in the resistance table. As we mentioned earlier, the resistance table (RT) is a Two-Dimensional array and it has 128 rows and 8 columns; there are a total of 1024 data. The decimal value of address array (0-127) A_i will decide on the row in the resistance table, and the decimal value of current array (0-7) C_j will decide on the column in the table. These two indices will determine the exact resistance RT_{ij}.

Since it is a protocol for encryption and decryption, it is natural to have information that needs to be encrypted. The operation for encrypting information is shown in Figure 5.

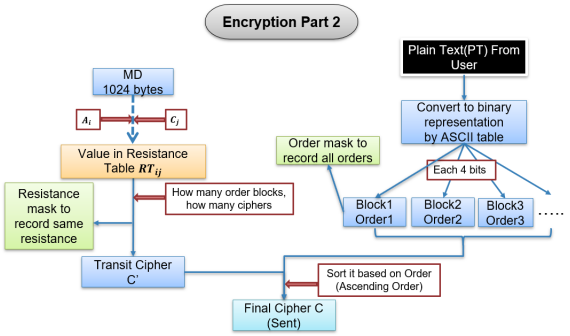

Fig. 5. Encryption Part2

As shown, the sender side will be allowed to enter a plaintext (PT) that needs to be encrypted, and the PT will be divided into blocks by using the ASCII table [11]. This is shown on the right side of Figure 5. For example, if a user enters "hello" as the PT, it will be converted into its hexadecimal representation at first which are 0x68 (h), 0x65 (e), 0x6c (l), 0x6c (l), 0x6f (o). Then it will be divided into 5x2 = 10 blocks. Each block contains 4 bits. If the hexadecimal notation 0x68 represents 1 byte (8 bits), then there are two blocks for this character, and '6' and '8' will be decimal values for each block. After getting the corresponding blocks, decimal values of each block obtained from PT will be shown in Figure 6:

The string you entered is:	h		e		l		l		o	
The hexadecimal notion of plaintext is:	06	08	06	05	06	0C	06	0C	06	0F
The decimal value for each block(4 bits) are:	6	8	6	5	6	12	6	12	6	15

Fig. 6. Decimal values of each block extracted from PT

These values are also considered to be orders in the protocol, they will be used to sort the cipher. The sender side will also use an "order mask" (OM) to record these values because they will be used in the decryption process. The sender side will create an array with 16 positions (the index is from 0 to 15) to record how many times each value appears. In the above example, number '5' appears one time, number '6' appears five times, number '8' appears one time, number '12' appears two times, and number '15' appears one time. The result of this array should look like in Table I. The values in the right column will be contained in the OM.

0	=>	0
1	=>	0
2	=>	0
3	=>	0
4	=>	0
5	=>	1
6	=>	5
7	=>	0
8	=>	1
9	=>	0
10	=>	0
11	=>	0
12	=>	2
13	=>	0
14	=>	0
15	=>	1

TABLE I
ORDER MASK, VALUES ARE ON THE RIGHT

In short, we have introduced how to get resistances from RT by using A_i and C_j. These resistances will be seen as ciphers directly, and decimal value of PT blocks is known as order, which will be used for sorting these ciphers. Also. the number of blocks extracted from PT will decide the number of ciphers. Such as in the example whose PT is "hello;" it has a total of 10 blocks, so the sender side will extract ten values from RT as well.

Then the sender side will decide which ten values in RT will be used as ciphers. Figure 5 explains this process. First, the sender side will extract ten values from the RT by using decimal values of "address" and "current," which are extracted from blocks of MD. A_0 and C_0 will give us $RT_{0,0}$, A_1 and C_1 will provide $RT_{1,1}$, and so on, until the A_9 and C_9 will give us $RT_{9,9}$. The subscripts of A and C represent which block they are using from MD; in this step, the sender reads from the first block of MD (subscript is 0). From now on, we will use R[i] to represent $RT_{i,i}$ for simplicity. The ten values R[0] to R[9] are candidates for the cipher.

In the process of getting R[i] from A_i and C_i, the protocol will create another mask which is the "resistance mask" (RM) that will be responsible for recording which position in R[i] array contains the same resistance. The RM will only consist of either 0's or 1's. '0' means the value in R[i] array corresponding to this position (0's position in RM) is unique, and '1' means the value in R[i] corresponding to this position (1's position in RM) is repeated. The protocol will avoid repeated resistances as ciphers. So, the sender side will check and delete the same value in R[i] array and push '1' to RM to label this position. To illustrate, let's use the example. If the characters in PT are "hello;" and from R[0] to R[9], the value of R[8] is the same as the value of R[0], the sender side will use '1' to represent this repeated condition in RM. Table II below illustrates this step.

When sender side finds that R[8] is equal to R[0], it will use '1' for this position in RM and will not use R[8] as a candidate cipher. After discarding R[8], the sender side will have a total of 9 candidates whose values are all unique; these

R[i]	R[0]	R[1]	R[2]	R[3]	R[4]
RM	0	0	0	0	0
R[i]	R[5]	R[6]	R[7]	R[8]	R[9]
RM	0	0	0	1	0

TABLE II
RESISTANCE MASK, AND R[8] IS THE SAME AS R[0]

nine values are the cipher to be sent by the sender side. But that is not enough; the protocol needs ten ciphers since the PT has been divided into ten blocks. As a result, the sender side will read one more value R[10] from the RT, which is got through A_{10} and C_{10}. Then the sender will check if R[10] is different from R[0] to R[9]. If so, the sender side will use R[10] as the 10^{th} cipher and use '0' to label this position in RM. The result after this operation is shown in Table III.

	R[0]	R[1]	R[2]	R[3]	R[4]	
RM	0	0	0	0	0	
	R[5]	R[6]	R[7]	R[8]	R[9]	**R[10]**
RM	0	0	0	1	0	**0**

TABLE III
READ A NEW VALUE R[10] FROM RT

In addition, the length of the RM is not fixed. In Table III, only R[8] and R[0] are repeated, so the sender side used '1' to represent that and then read a new value from the RM as a new cipher. Then the 10 values R[0] to R[10] (R[8] is dropped) will be included in our transit cipher ($C^{'}$). Table IV illustrates this example:

	R[0]	R[1]	R[2]	R[3]	R[4]
Transit Cipher $C^{'}$	$C_0^{'}$	$C_1^{'}$	$C_2^{'}$	$C_3^{'}$	$C_4^{'}$
	R[5]	R[6]	R[7]	R[9]	R[10]
Transit Cipher $C^{'}$	$C_5^{'}$	$C_6^{'}$	$C_7^{'}$	$C_8^{'}$	$C_9^{'}$

TABLE IV
BUILD TRANSIT CIPHER $C^{'}$ FROM R[0] TO R[10]

But there is the possibility that the new value R[10] is also repeated with other values from R[0] to R[9] which needs to use another '1' in the RM, and then read a new value R[11] from the RT. In this case, the length of the RM will be 12. The length of the cipher will be still 10, which means the length of the RM may change, but the length of the cipher is fixed, only depending on the number of blocks of PT. Also, the sender side has got 10 decimal values from blocks of PT; the sender side will call these values "order" (Od) because it will use these values to sort $C^{'}$. The Od and $C^{'}$ are shown in Table V. The values of Od used here are from Figure 6.

Od	6	8	6	5	6
Transit Cipher $C^{'}$	$C_0^{'}$	$C_1^{'}$	$C_2^{'}$	$C_3^{'}$	$C_4^{'}$
Od	12	6	12	6	15
Transit Cipher $C^{'}$	$C_5^{'}$	$C_6^{'}$	$C_7^{'}$	$C_8^{'}$	$C_9^{'}$

TABLE V
TABLE CONTAINS ORDER AND TRANSIT CIPHER

The next step is to sort $C^{'}$ according to Od and then get the final cipher (FC). Figure 7 illustrates it.

As shown, the protocol will sort the transit cipher in the ascending order according to the values in Od array. For example, number '5' is the smallest in Od array, and the transit cipher corresponding to '5' is $C_3^{'}$. So $C_3^{'}$ will be in the first position in the final cipher FC. Then, number '15' in Od array

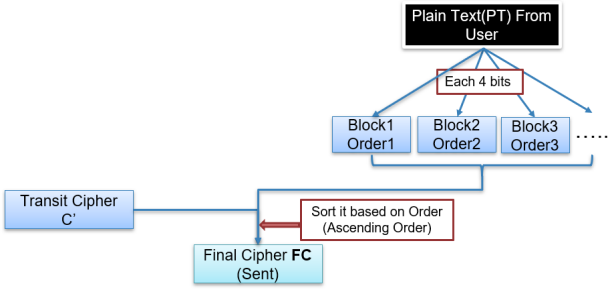

Fig. 7. How to get final cipher (FC)

is the largest value, so the value $C_9^{'}$ corresponding to it should be in the last position of FC. Finally, after sorting the $C^{'}$ based on Od array, the result will be shown in Table VI.

Od	5	6	6	6	6
Final Cipher FC	$FC_0 = C_3^{'}$	$FC_1 = C_0^{'}$	$FC_2 = C_2^{'}$	$FC_3 = C_4^{'}$	$FC_4 = C_6^{'}$
Od	6	8	12	12	15
Final Cipher FC	$FC_5 = C_8^{'}$	$FC_6 = C_1^{'}$	$FC_7 = C_5^{'}$	$FC_8 = C_7^{'}$	$FC_9 = C_9^{'}$

TABLE VI
FINAL CIPHER IN CORRECT ORDER

So far, we have got the final cipher FC, the resistance mask RM, and the order mask OM. To implement decryption on another side, they all need to be sent to that side. FC can be sent directly and safely, but it is not safe to send two masks in the same way without any protection.

The protocol will implement XOR (Exclusive or) operation on these two masks and half of MD separately in order to ensure the safe transmission of information. This step is shown in Figure 8.

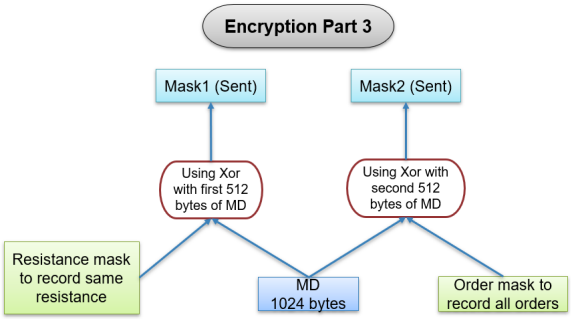

Fig. 8. Encryption Part3

The protocol uses XOR function here because XOR is an involutory function, which means if the protocol applies XOR twice, it can get the original RM and OM back during decryption [12]. For RM, the protocol will use it to perform XOR operation with the first half of MD, and OM uses the second half of MD to perform XOR operation. After that, "Mask1" and "Mask2" will be generated, which are both 512 bytes long. "Mask1" and "Mask2" are different from the two initial masks OM and RM. But RM and OM are shorter than half of MD, so when doing XOR operation, the system will automatically fill in some bytes at the end of these two masks

to let them have 512 bytes. Finally, "Mask1" and "Mask2" will be sent to another side. That is all for encryption.

IV. Decryption

Upon receiving "Mask1", "Mask2", and FC, the receiver side will use this information to restore the same PT generated in encryption. In environmental setup section, we have mentioned that the RN and PW will be known by both sides. So, the receiver side can get the same MD by doing the same operation that the sender side have done with encryption. And the "Mask1" and "Mask2" are both obtained by doing XOR operation with half of MD. XOR is an involutory function. If the protocol applies XOR twice, it will get the original thing back. Figure 9 illustrates that the receiver side can retrieve the RM and OM back by implementing XOR operation with MD again.

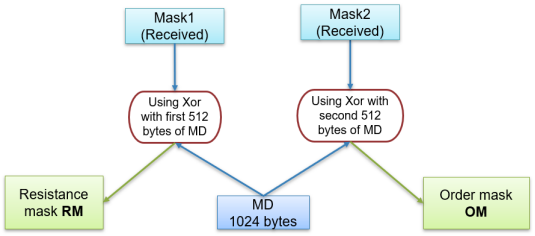

Fig. 9. Retrieving OM and RM

In Figure 9, after the XOR operation is performed on the "Mask1" and the first half of MD, the receiver side will get the RM back. The problem here is that the receiver side does not know which part in the RM is for regulating ciphers. Taking the previous example whose PT is "hello," the actual length of RM is 11, but after the XOR operation in encryption, its length becomes 512 bytes. So, the receiver side gets a 512-byte RM, and does not know which part is useful.

The receiver side will use another method to solve this problem. Because both sides have the same MD and share the same A_i and C_j, A_0 and C_0 will provide R[0] on both sides, A_1 and C_1 will provide R[1] on both sides and so on. As some unique R[i]'s will be used as $C^{'}$ directly, then FC will be generated based on sorting $C^{'}$. Figure IV shows the relationship between $C^{'}$ and FC.

$C^{'}$	$C^{'}_3$	$C^{'}_0$	$C^{'}_2$	$C^{'}_4$	$C^{'}_6$
FC	$FC_0 = C^{'}_3$	$FC_1 = C^{'}_0$	$FC_2 = C^{'}_2$	$FC_3 = C^{'}_4$	$FC_4 = C^{'}_6$
C'	$C^{'}_8$	$C^{'}_1$	$C^{'}_5$	$C^{'}_7$	$C^{'}_9$
FC	$FC_5 = C^{'}_8$	$FC_6 = C^{'}_1$	$FC_7 = C^{'}_5$	$FC_8 = C^{'}_7$	$FC_9 = C^{'}_9$

TABLE VII
Relationship between FC and $C^{'}$

As a result, if the R[i] is contained in $C^{'}$, it must also be found in FC. The receiver side will loop from A_0 and C_0 and get its corresponding R[0] from the RT, and then check if R[0] appears in FC. If so, it will continue to search until an R[m] value is not found in FC, it will stop searching, and elements from R[0] to R[m-1] must be $C^{'}$'s elements. Next, the receiver side will count how many elements are there from R[0] to R[m-1], and the result will give the receiver side the useful part in the RM. This useful part will be named ERM. The receiver will use ERM to get $C^{'}$. Using the same example whose PT is "hello," the ERM obtained above is shown in Table VIII.

0	0	0	0	0	0	0	0	1	0	0

TABLE VIII
ERM Table

And R[i] got from the RT will be shown in Table IX:

$A_0, C_0 \rightarrow R[0]$	$A_1, C_1 \rightarrow R[1]$	$A_2, C_2 \rightarrow R[2]$	$A_3, C_3 \rightarrow R[3]$	$A_4, C_4 \rightarrow R[4]$
$A_5, C_5 \rightarrow R[5]$	$A_6, C_6 \rightarrow R[6]$	$A_7, C_7 \rightarrow R[7]$	$A_8, C_8 \rightarrow R[8]$	$A_9, C_9 \rightarrow R[9]$
$A_{10}, C_{10} \rightarrow R[10]$	$A_{11}, C_{11} \rightarrow R[11]$	$A_{12}, C_{12} \rightarrow R[12]$	..	

TABLE IX
Resistances from RT

There will be a lot of resistances extracted from the RT, but since the length of the ERM is 11, the receiver side will only use 11 values from R[0] to R[10]. The value in the ERM determines which R in Table IX can be used as $C^{'}$. For example, the corresponding value of R[0] is '0' in Table VIII, so R[0] will be pushed to $C^{'}$. But if the corresponding value is '1,' that value will be discarded such as R[8]. Then other values from R[0] to R[10] will become elements of $C^{'}$. The $C^{'}$ the receiver side gets here is the same as the previous one when encrypting. Figure 10 provides a summary of the above steps.

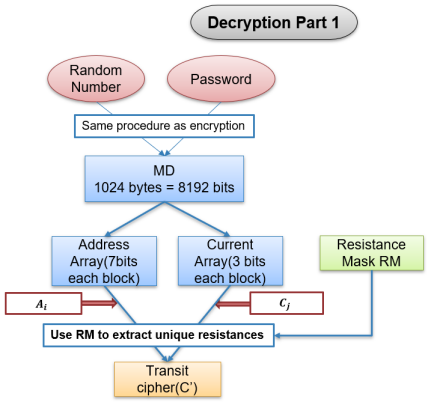

Fig. 10. Decryption Part1

After getting FC and $C^{'}$, the receiver side will get the OM by doing XOR operation on "Mask2" and second half of MD. The first 16 values in the output are useful and should be same as the right side in Table X.

In Table X, see following page, the numbers on the left side are indices, and the numbers on the right side are the content really included in the OM. For example, "5 => 1" means number '5' appears one time, "6 => 5" means number '6' appears 5 times, until "15 => 1" means number '15' appears one time. The result in Table XI turns the number of occurrences into actual numbers. Because number '6' appears five times, there will be five 6's in the table.

The receiver side thinks of it as an array with 10 elements, and we will call it RO, which means "random order." The reason is, compared with Table V, the elements in RO array

0	=>	0
1	=>	0
2	=>	0
3	=>	0
4	=>	0
5	=>	1
6	=>	5
7	=>	0
8	=>	1
9	=>	0
10	=>	0
11	=>	0
12	=>	2
13	=>	0
14	=>	0
15	=>	1

TABLE X
ORDER MASK, VALUES ARE ON THE RIGHT

5	6	6	6	6
6	8	12	12	15

TABLE XI
RANDOM ORDER ARRAY RO

are same as the elements in Od array but in a different order. If the receiver side wants to get a correct PT, it has to turn it into the correct order. The plan implemented is shown in Figure 11.

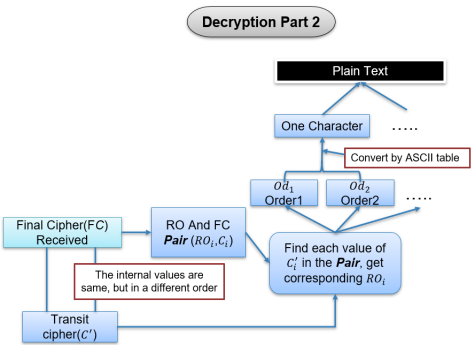

Fig. 11. Decryption Part2

The receiver side will make a pair of $C^{'}$ and RO as described in Figure 11. And then it will find each value of $C^{'}$ in this pair and get its corresponding RO_i. The example is shown in Table XII. Numbers in the "Corresponding RO in the ***Pair***" row are the same as the numbers in Od array, and they are in the same order.

(FC,RO) *Pair*	$(FC_0,5)$	$(FC_1,6)$	$(FC_2,6)$	$(FC_3,6)$	$(FC_4,6)$
Relationship between FC and $C^{'}$	$FC_0 = C_3^{'}$	$FC_1 = C_0^{'}$	$FC_2 = C_2^{'}$	$FC_3 = C_4^{'}$	$FC_4 = C_6^{'}$
(FC,RO) *Pair*	$(FC_5,6)$	$(FC_6,8)$	$(FC_7,12)$	$(FC_8,12)$	$(FC_9,15)$
Relationship between FC and $C^{'}$	$FC_5 = C_8^{'}$	$FC_6 = C_1^{'}$	$FC_7 = C_5^{'}$	$FC_8 = C_7^{'}$	$FC_9 = C_9^{'}$
Transit Cipher $C^{'}$	$C_0^{'}$	$C_1^{'}$	$C_2^{'}$	$C_3^{'}$	$C_4^{'}$
Corresponding RO in the *Pair*	6	8	6	5	6
Transit Cipher $C^{'}$	$C_5^{'}$	$C_6^{'}$	$C_7^{'}$	$C_8^{'}$	$C_9^{'}$
Corresponding RO in the *Pair*	12	6	12	6	15

TABLE XII
GET CORRECT OD ARRAY BY SEARCHING $C^{'}$ IN THE *Pair*

So far, the receiver side has got Od array back. Every two values in Od array represent one character and can be converted into their corresponding character through ASCII table. For example, the first two numbers in Od are '6' and '8,' then the receiver will combine them into hexadecimal notation, which is "0x68." The character corresponding to "0x68" in ASCII table is 'h.' Finally, the receiver side will convert the ten numbers in RO array into five characters and get the original PT - "hello." That is all for decryption.

V. SECURITY EVALUATION

It is important that other people cannot retrieve cipher values by observing cell usage in RT. So, in this section, we discuss and verify the security of the protocol by measuring cell usage. We have introduced that ciphers came from the RT directly, where the RT is a two-dimensional array with 128 rows and 8 columns, a total of 1,024 cells. The protocol uses addresses and currents which are obtained from MD to determine specific cell from RT. Then these cell values will be treated as cipher after sorting. The problem is that if we encrypt different messages, the values extracted from the RT each time are the same, which means that ciphers used are almost the same every time. Hackers can decipher information by observing cipher usage. But if every data in the RT has the opportunity to be used, the ciphers generated during encryption will be different, which will increase the difficulty of cracking our message. So, we will measure the cell usage in the form of a statistical chart below [13].

Figure 12 is the Scatter graph after doing encryption and decryption 1,000 times with 140 random characters. 140 characters is the maximum value that the protocol can implement encryption and decryption in the case where the MD length is 8,192 bits. We want to test the performance of the protocol in extreme states. There are 128 rows and 8 columns in the RT, which constitute a total of 1,024 cells. In Figure 12, the horizontal axis represents the row number, the vertical axis represents the number of times used, and the points with eight different colors represent the column number. So, Figure 12 shows how many times each cell is used. As Figure 12 demonstrates, each cell has been called approximately 520 times, which means each cell has been used a similar number of times without the extremely unsafe situation where some cells have been used tens of thousands of times, and some have been used only a few times.

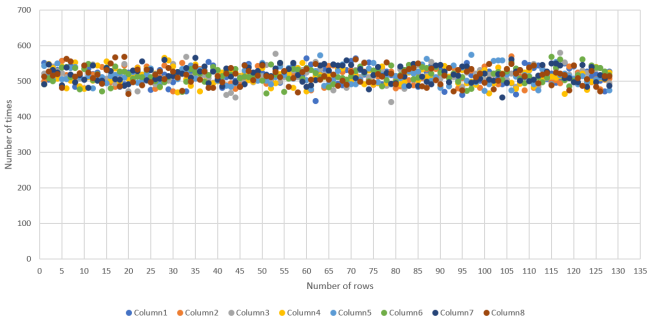

Fig. 12. Scatter graph for each cell in RT, 1,000 times, 140 characters

Figure 13 is similar to Figure 12, but it does not show the usage for each cell. It shows the usage of each row. From Figure 12, we can see that each cell is used about 520 times, and there are 8 cells in a row, so it is expected that the number of uses per row should be around 4,160. The data in Figure 13 confirms our expectations.

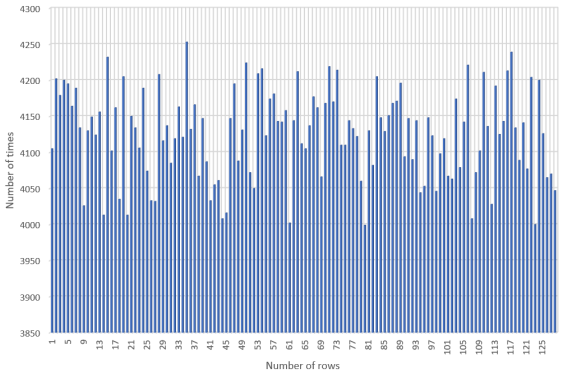

Fig. 13. Column chart for each row in RT, 1,000 times, 140 characters

In order to get more accurate results, we performed more tests. As shown in Figures 14 and 15 below, we used 140 random letters for encryption and decryption 10,000 times.

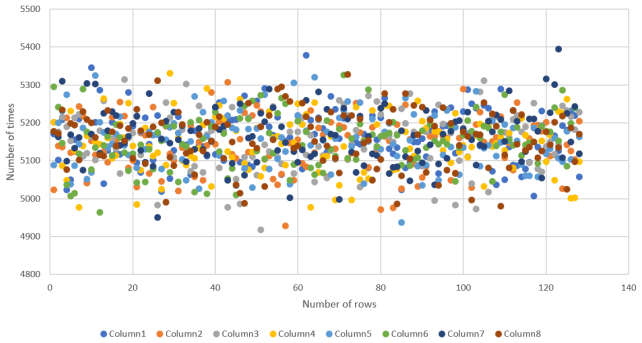

Fig. 14. Scatter graph for each cell in RT, 10,000 times, 140 characters

The data in Figure 14 has the similar meaning as in Figure 12, the only difference is the number of tests. Under 10,000 tests, we can see that most cells are used around 5,000 times, and the smallest value in the graph is still greater than 4,900. There is no case where the cell is not used or is rarely used.

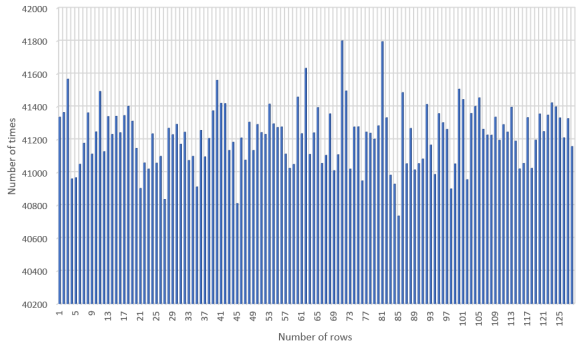

Fig. 15. Column chart for each row in RT, 10,000 times, 140 characters

Then, in Figure 15, the data reflects the number of times each row is called. Again, all of the values in the column chart are larger than 40,000. The largest value in the chart is about 41,800. The gap is within acceptable limits.

By verifying the encryption and decryption procedure of 140 characters 1,000 and 10,000 times, we found that our method has very high and similar utilization rates for different cells in the RT; even under 10,000 tests, there are no special cases. In general, we can conclude that the protocol is safe, and it is not easy for hackers to decipher the message by discovering cell usage.

VI. Conclusion

This paper proposed an extended protocol based on memristor arrays to perform encryption and decryption without using any keys. First, an MD is generated on the sender side and the receiver side using the hash function. Then, both sides import the resistance table, which is obtained from the memristor arrays. Next, the plaintext to be encrypted is divided into small blocks. Finally, the protocol uses decimal values of these blocks to sort the ciphers extracted from resistance table. In the evaluation section, this paper also provided security proof for this protocol in the case of multiple tests. Going forward, we want to implement the protocol into hardware because hardware design was not involved with our study. For example, we can build two communicating devices based on memristor arrays [5],[14]. If so, ciphers generated can only be decrypted by the same memristors stored in separate devices.

References

[1] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, Apr 2011. [Online]. Available: https://doi.org/10.1007/s13389-011-0006-y

[2] N. Baracaldo, L. A. D. Bathen, R. O. Ozugha, R. Engel, S. Tata, and H. Ludwig, "Securing data provenance in internet of things (iot) systems," in *International Conference on Service-Oriented Computing*. Springer, 2016, pp. 92–98.

[3] R. Roman, P. Najera, and J. Lopez, "Securing the internet of things," *Computer*, no. 9, pp. 51–58, 2011.

[4] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[5] B. Cambou, S. Assiri, and D. Hely, "Memristors to design keyless encrypting devices," *ACM Journal on Emerging Technologies in Computing Systems (JETC), unpublished*.

[6] B. Francis Cambou, "Design of true random numbers generators with ternary physical unclonable functions," *Advances in Science, Technology and Engineering Systems Journal*, vol. 3, pp. 15–29, 05 2018.

[7] B. Cambou, "A xor data compiler: Combined with physical unclonable function for true random number generation," 07 2017, pp. 819–827.

[8] B. Cambou and M. Orlowski, "Design of pufs with reram and ternary states," in *Proceedings of the 11th Annual Cyber and Information Security Research Conference, Oak Ridge, TN, USA*, 2016, pp. 5–7.

[9] J.-W. Han, C.-S. Park, D.-H. Ryu, and E.-S. Kim, "Optical image encryption based on xor operations," *Optical Engineering*, vol. 38, 1999.

[10] C. Boutin, "Nist releases sha-3 cryptographic hash standard. nist information technology laboratory," 2015.

[11] A. Kaushik, A. Kumar, and M. Barnela, "Block encryption standard for transfer of data," in *2010 International Conference on Networking and Information Technology*. IEEE, 2010, pp. 381–385.

[12] C. Li, S. Li, G. Alvarez, G. Chen, and K.-T. Lo, "Cryptanalysis of two chaotic encryption schemes based on circular bit shift and xor operations," *Physics Letters A*, vol. 369, no. 1-2, pp. 23–30, 2007.

[13] A. Kahate, *Cryptography and network security*. Tata McGraw-Hill Education, 2013.

[14] R. Stanley Williams, "How we found the missing memristor," in *Chaos, CNN, Memristors and Beyond: A Festschrift for Leon Chua With DVD-ROM, composed by Eleonora Bilotta*. World Scientific, 2013, pp. 483–489.