International Journal on Advanced Science Engineering Information Technology

Vol.8 (2018) No. 6
ISSN: 2088-5334

On the Fly Access Request Authentication: Two-Layer Password-Based Access Control Systems for Securing Information

Muhammed Jassem Al-Muhammed, Ahmad Daraiseh

Faculty of Information Technology, American University of Madaba, Madaba, Jordan
E-mail: m.almuhammed@aum.edu.jo; a.daraiseh@aum.edu.jo

***Abstract*— In the digital era, most of our highly sensitive documents are stored in computers. These documents are in a great threat unless protected using appropriate measures. Despite their several imperfections, passwords are becoming the de-facto mechanism for securing documents stored in local directories or on the websites. In this scheme users protect their documents using passwords. In order for such scheme to work, the passwords must be stored in the file system either in plain or hashed form so that they can be used as references when information is requested. This paper proposes innovative password-based protection system. Although the proposed system uses passwords for document protection, it proposes a completely different way of using and managing these passwords. Our system protects a stored document in terms of both the document itself and the password. Both the document's content and the password are used along with random noises to generate security code that serves as a reference when the document is requested. The security code is neither reversible nor reproducible without a full knowledge of the password and the content of the document. The users of our system keep their passwords and provide them only when they first store the document and when they later request document retrieval. The passwords are never stored neither in their plain nor hashed forms. Experiments with our prototype implementation showed that our protection scheme is effective and passed important security tests.**

***Keywords*— password-based security; information security; document protection; access control; security code; passcodes.**

I. Introduction

Almost all of our sensitive documents are stored digitally in computers or other storage devices. Securing this digital information is a very challenging problem due to the ever-advancing tools and techniques used by intruders to gain access to the repositories of this information. Researchers have proposed many ways to protect information from unauthorized access [1]–[6]. Passwords are widely used mechanisms to prevent unauthorized access to information [7]–[12]. In such a scheme, users protect their sensitive documents using passwords. These passwords are stored in the system and used for permitting or denying access to the password-protected information. To protect the passwords themselves, current security systems store these passwords in hashed forms rather than in plain forms. The security of the password is therefore solely protected by the irreversibility of these hash values. If intruders can reverse the hash value, the original password is recovered and the access to the information is gained.

This paper advocates using a new approach for protecting passwords and controlling access to password-secured information. We call this new approach on-the-fly password-based authentication. This type of authentication in which the system does not require the storage of user passwords may be the best way to protect passwords. It goes a long way toward allowing authenticating access requests without requiring any password to be stored internally in any place in the system. Such an authentication mechanism extremely restricts, or even fully eliminates, the ability of intruders to perform password-related penetration to the stored information since these passwords, which are required for accessing this information, simply do not exist in the system. Unlike the current measures that keep passwords in the system but protect them via some method (e.g. hashing, salting), we believe that even if the original passwords are protected by some method, their presence in the system is likely to expose them, and consequently the information they protect, to a great risk.

This paper proposes innovative password-based security system for controlling access to the information stored in its repository. Our system uses passwords to protect the information in a very unique way. Rather than storing these passwords internally in the security system in hashed or plain forms, our system allows users to keep the passwords in their favorite secure places. The fundamental part of our security system is the security code generator, which is a process that is composed of several operations: password expansion operation, complex-irreversible mapping operation, change amplification operation, and random noise salting operation. This security code generation process utilizes user-provided passwords and documents to be protected along with a source of random noise to produce passcodes called security codes. These security codes are used by our system to authenticate future access requests.

In addition, the paper offers a random generator, which is the source for all the random noises that are required by the operations of the security code generator to produce security codes. The input of the random generator consists of the passwords and documents and therefore its output varies as these inputs change. Along with the input information, the random generator state is continuously updated based on feedbacks collected during the generation process.

The paper offers the following contributions. First, it proposes innovative security system for controlling access to stored information. Second, it proposes a security code generator to produce security codes, which provide mechanism for controlling access to protected information. Third, it proposes an effective random number generator whose input depends on user-provided passwords and the information to be protected and whose state is continuously updated using feedbacks collected during the generation process. “Secure-by-elimination” approach

II. Material and Method

This section introduces the fundamental parts of our proposed system. It specifically presents the architecture of the system and discusses in great details the functional components of the system.

A. System Architecture

This paper proposes two-layer, password-based file protection system. Fig. 1 the high-level components of our proposed system. The first layer offers access control to the second layer. This layer is responsible for authenticating and permitting each access if it can be authorized. The second layer provides access to the file system only if the access control layer permits such an access.

The access control layer maintains light-yet-fundamental up-to-date metadata about each file protected by our system. Each protected file is represented by a record in the metadata. The record consists of three pieces of information pertaining to a file: (1) the file name, (2) a hash value for the content of the file, and (3) a security code generated for this file. To build a record, the security system uses the content of the file as an input to some hash function (e.g. SHA-512) and produces a hash value. The security code generator creates a security code for a file using the hash value of the file and the user-provided password. Both the hash value and the security code are encrypted using any encryption method such Advanced Encryption Standard [13], [14] or ĸ–Lookback Text Encryption Technique [15].These three pieces of information form a record for the file and are stored in the metadata.

The security system uses its metadata for authenticating access requests. When the security system receives a request for accessing a specific file, it accesses the metadata and retrieves the hash value corresponding to this file (if exists). The hash value is decrypted and passed on as an input to security code generator. The security code generator uses both the user-provided password and the hash value to compute, in way to be made clear later, the security code. The computed security code and the stored one after being decrypted are compared. If the provided password correct, the generated security code will match the stored one and consequently the access control layer permits the access to the file system. If the access control layer does not permit the access, the request is denied. Our system does not allow users to store their passwords. Instead, the users provide their passwords on the fly when they request an access to a specific file. Keeping the passwords away from the security system has many important security advantages. First, users do not have to worry about the security of their passwords since these passwords are under their control. Second, the system is no longer a leaking point that can be hacked to learn passwords. Third, users have full control over their passwords.

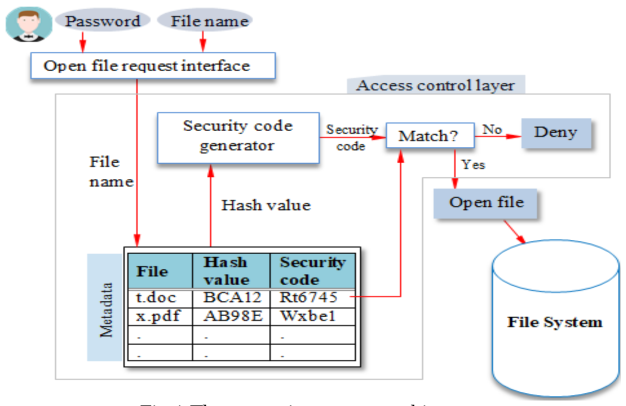


Fig.1 The protection system architecture

B. Hash Value Generation

The system requires a hash value for each stored file. There is a plethora of hash algorithms that can be used to create hash values. The hashing algorithms SHA-*x* [16]–[19], MD-5 [20], and SWIFFT [21] are just to name few. Our system requires no specific hash function algorithm. The only requirement is that the hash algorithm is highly sensitive to its input. That is, a tiny change to the input must cause drastic changes to the output. For the purpose of our implementation, Secure Hash Algorithm (SHA-512) is used. SHA-512 is novel hash function that uses in its computations 64-bit words. This algorithm processes a text of arbitrary length and condenses it to 512-bit string called message digest. SHA-512 has two stages of processing. The first stage involves padding ℓ bits to the message so that it becomes multiple of 512 bits. In the second stage, the SHA-512 applies six logical functions each of which operates on 64-bit words and produces 64-bit word. (The logical operations can be found elsewhere [19].)

C. Security Code Generator

The security code generator is the most fundamental component in our system. In particular, this component creates the security code, which is the security piece that is used by our security system for permitting or denying access to a specific file. Fig. 2 shows the generator’s main operations. In the following subsections, we discuss these operations and then describe how these operations work synergistically for generating the security code.

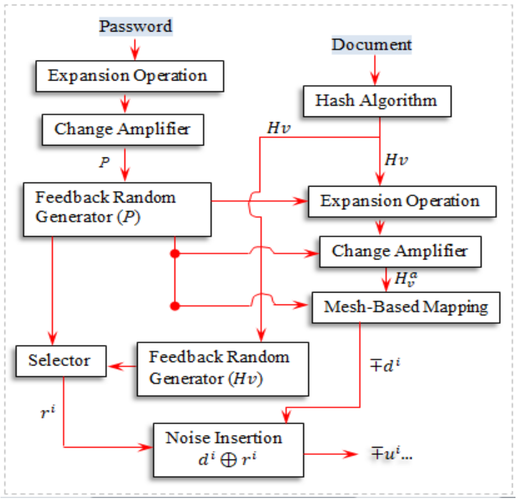


Fig. 2 The structure of the security code generator.

1) Feedback-Base Random Generator: The random generator plays very essential role in our security system. It provides sequences of random numbers as an input for all the operations of the security code generator. It particularly provides input for the mesh-based mapping technique, change amplifier operation, expansion operation, and the noise insertion operation. In order for these operations to function properly, the random generator must possess some important properties. First, the random sequences must be unpredictable. This ensures that the output of the operations has no pattern that can be exploited by adversaries. Second, it must have long period so that the generator can provide sequences of random numbers with any arbitrary length. Third, its input must depend on the user's provided credentials and the documents being secured. In addition, its sequences must be reproducible given the appropriate information.

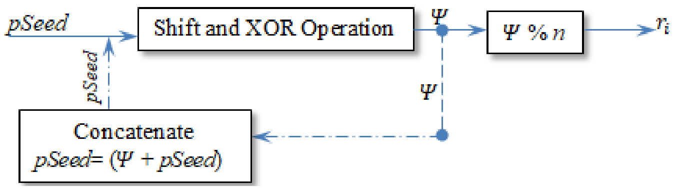
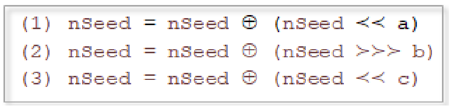


Fig.3 the main components of the feedback based random generator

This paper proposes a feedback-back random generator. Fig. 3 shows the components of the random generator. The initial input to the random generator, *pSeed*, is a sequence of *n* unicode symbols. Since the generator requires numerical seeds, we create one from the input *pSeed* using the following formula.

$$nSeed = \sum_{i=1}^{n} [val(s_i)]^3 \qquad (1)$$

Where *val* (s_i) is the integer index of s_i in the unicode encoding system and s_i is the symbol with the index *i* in *pSeed*. The input *nSeed* is passed to the shift and XOR operation. This operation adds random noises to its input by applying the logic in Fig. 4 [22].



```
(1) nSeed = nSeed ⊕ (nSeed << a)
(2) nSeed = nSeed ⊕ (nSeed >>> b)
(3) nSeed = nSeed ⊕ (nSeed << c)
```

Fig.4 Shift and XOR operations

The logic in Fig. 4 defines three operations that apply to the integer input *nSeed*. These operations are XOR operation "⊕", left arithmetic bit shift "<<", and right logic bit shift ">>>". The operation "<<" left shifts the bits of *nSeed* a number of positions equals to *a* in (1) and to *c* in (3) while both preserving the sign bit of the number. The operation ">>>" right shifts the bits of *nSeed* by *b* positions without preserving the sign bit. The shift amounts *a*, *b*, and *c* are specified as described in [23].

The output of shift and XOR operations is the integer Ψ, which is used to generate the random number r_i using the module operation Ψ % *n* (*n* is the maximum limit of the random numbers range). The number Ψ is also used as a feedback to the input of the generator. The feedback Ψ is concatenated to the left of the previous seed to create a new seed, which is used for generating the next random number. Due to the continuous feedback, the *pSeed* keeps growing and the formula (1) may result in an overflow. When the overflow occurs, the random generator reduces the size of the *pSeed* to the leftmost 64 symbols and uses these symbols as a new seed. Unlike other random generators that use the previously generated value as a seed for generating the next random number, our generator creates each new random number using a new seed. This has a very important impact on the predictability. The current random number has no relation to the next random numbers. More precisely, each random number is independent of the other random numbers in the sequence.

2) Change Amplifier: One of the most important security requirements in our approach is that changes in passwords or hash values irrespective of their magnitude (one bit or more) must cause tremendous changes to the corresponding security code. This high sensitivity to input's changes makes the relationship between the generated security code and both the passwords and hash values from which the security code is generated very complicated and untraceable. The objective of our proposed change amplifier is to detect changes in passwords and hash values, magnify these changes, and propagate them to every symbol in the output. Fig. 5 shows the three operations of the amplifier.

The amplifier executes first forward pass, which performs XOR operation on its input symbols to propagate the change left to right. The forward pass produces its output as follows. The first symbol in the input is the first symbol in the output without change. The rest of the output symbols are computed by XORing the most recently computed symbol with the current input symbol.

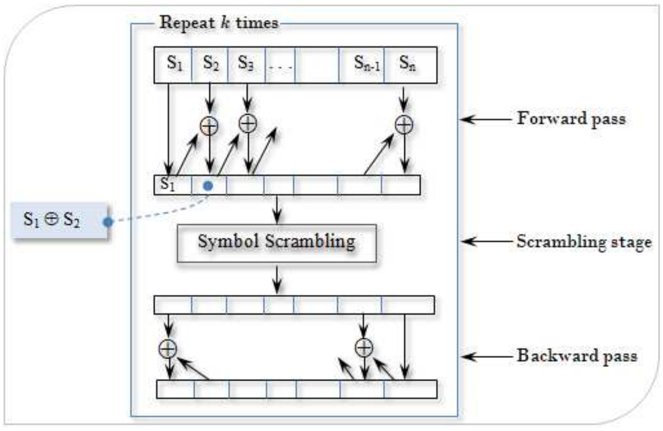


Fig. 5 The three operations of the change amplifier

The amplifier executes next the symbol-wise scrambling operation. This operation makes sharp changes to its input symbols. It performs fine-grained scrambling to the bits of each symbol, possibly yielding a new symbol. Specifically, the scrambling operation performs different number of left shifting to each symbol in the input. The number of shifts is a positive random number that is less than the number of bits that represent the symbol. The feedback random generator provides the required random numbers for the functionality of the scrambling function.

The amplifier finally executes the backward pass. This operation has the same functionality as the forward pass except that it starts from the end of the input. Therefore, the last symbol in the input is copied to the last position in the output with no change. The rest of the symbols are computed by XORing the most recently computed symbol with the current input symbol.

These three operations may be repeated *k* times for better change propagation. Typically, *k*=3 is very adequate for largely amplifying and propagating the changes so that every symbol in the input is impacted.

The amplifier plays a very vital role in our system. In order for the security code to disclose no information about the password and the hash value that may help in predicting them, it must be the case that tiny changes to the input (passwords, hash values) necessarily cause drastic change to the output (security code). Therefore, the main objective of the amplifier is to ensure that changes to the inputs entail large changes to the output.

TABLE I
THE IMPACT OF THE CHANGE AMPLIFIER TECHNIQUE ON THE INPUT'S CHANGES

Input	**Output (Hex)**															
aaaaaaaaaaaaaaaa	D9	1C	08	B	A8	F7	80	51	94	A1	6D	19	4C	EF	5B	7C
baaaaaaaaaaaaaaa	6D	01	4F	83	70	C1	37	F8	57	5E	83	0B	61	34	70	86
aaaaaaaaaaaaaaa**b**	49	64	6D	EF	A2	B	85	1C	E6	7C	49	79	7F	32	13	52
aaaaa**b**aaaaaaaaaa	5D	B	C2	2A	37	9D	D3	91	01	DA	FE	DA	61	8A	46	C7
cccccccccccccccc	4E	9C	5F	F5	05	5F	D8	63	5F	50	7D	D7	8D	5F	E4	4E
cccccccc**bbbbbbbb**	BE	FA	07	06	87	2E	4D	65	B5	D0	B6	45	1C	29	AB	54
The apple on tree	8C	64	C0	53	44	8E	96	08	2B	B0	31	83	23	BA	86	62
She apple on tree	B4	BE	EF	39	67	E7	15	BF	43	2B	2E	7D	77	A3	EE	82
Cat in the hat.!	D0	85	D2	8F	AF	08	51	DB	4C	DE	19	FD	98	D1	28	F5
Cat in the hat.**?**	B5	C8	53	94	60	E3	5E	5C	43	F0	45	4F	16	8D	06	96

Table I shows the response of the change amplifier to its input's changes. The first column shows a sample of different inputs to the amplifier and the second column shows its output (in hexadecimal). As the table shows, the operation responded to changes in its input by producing remarkably different output. Consider, for instance, the input in the first and the second rows, which differ in only single bit (the first "a" changed to "b"). It is clear that this tiny change resulted in a totally different output. The position of the change makes also significant difference. Consider, for instance, the inputs in rows 2, 3, and 4, which all differ in the position of the different single bit. It is clear that the output greatly varies depending on the position of the bit change. The rest of the rows lead to similar conclusions.

3) Expansion Operation

The expansion operation expands its input (especially short passwords) to any desired length. Fig. 6 illustrates the logic of our expansion operation. The expansion operation includes a substitution operation and three structure modification operations. These operations work synergistically to introduce deep alterations to the input (*Key*) and yield a larger version of the key. The substitution operation replaces every symbol in the input *Key* with a new symbol using the AES's substitution operation [13]. The AES substitution operation is simply a table lookup operation that utilizes 16×16 matrix of byte values called S-Box. The S-BOX is composed of all the possible combinations of 8 bits ($2^8 = 16 \times 16 = 256$). Mapping each symbol into a new one is performed as follows. The left half bits of the input symbol is used to index a particular row of the S-BOX and the right half bits to index a column. The content of the corresponding cell is retrieved as the result of the mapping. For instance, if the symbol is "a" (in binary "01100001"), then the left half bits "0110" ("6" in decimal) and the right half bits "0001" ("1" in decimal). The symbol at the cell (6, 1) is retrieved as a replacement for "a". The result of the substitution operation for the input *Key* is the string *S* whose length equals to the length of *Key*. Both *S* and *Key* are concatenated to yield a larger key (*Nkey*).

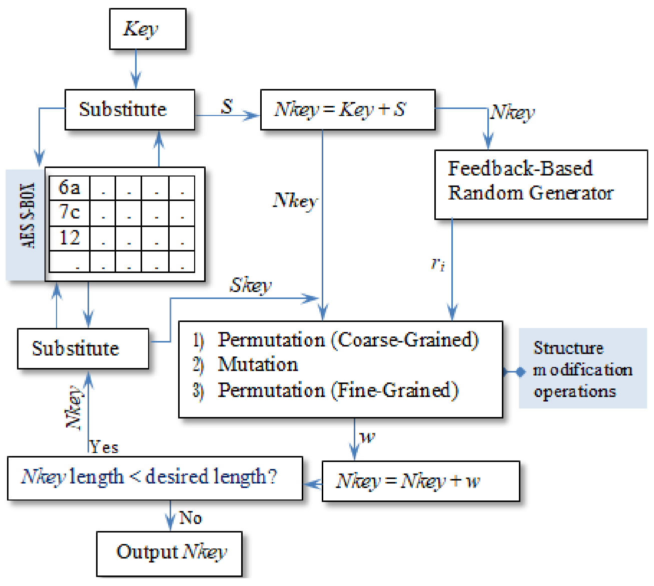

Key
Substitute
S
Nkey = Key + S
Nkey
Feedback-Based Random Generator
AES S-BOX
6a
7c
12
Nkey
r_i
Skey
Substitute
1) Permutation (Coarse-Grained)
2) Mutation
3) Permutation (Fine-Grained)
Structure modification operations
Nkey
Yes
w
Nkey length < desired length?
Nkey = Nkey + w
No
Output Nkey

Fig. 6 The expansion operation main components

The new key (*Nkey*) is passed to the feedback random generator as a seed and also to the structure modification operations for further expansion. The random number generator provides sequences of random numbers for the structure modification operations. The structure modification operations include bit mutation operation and two permutation operations: coarse-grained and fine-grained operations. These three operations operate on the input (*Nkey*) in the order specified in Fig. 6. The coarse-grained permutation splits the bit string representation of *Nkey* into groups G_1, G_2, …, G_s each of which is four bits. These groups are randomly reordered by swapping the group G_i with the group G_{r_i}, where the index r_i is a random number obtained from our feedback based random generator. The permutated bit string is passed to the bit mutation operation, which performs bit flipping with a probability of *p*. The bit flipping is performed as follows. Let *n* be the length the bit string. We obtain a sequence of *n*+1 random numbers from our feedback generator. The sequence of random numbers is confined to the interval [0, 1] by dividing each random by the maximum random number generated by our generator. The probability *p* is the first random number in the confined sequence. The rest of *n* random numbers in the sequence are used for flipping bits. Each bit in the string is flipped only if the value of the corresponding random number is less than or equal to *p*. For instance, if p =0.25 and the bit string is "011001" and the random sequence is "0.98, 0.24, 0.6, 0.01, 0.7, 0.5", the 2nd and 4th bits are flipped to yield the new bit string "001101". Finally, the fine-grained permutation operation performs exactly the same functionality as the coarse-grained, but with groups of two bits (rather than of four bits). The output of the expansion method is a string *w*, which is a deeply manipulated version of *Nkey*. The string *w* is concatenated with the original *Nkey* to create a larger *Nkey*. If the desired length of *Nkey* has not yet been reached, the expansion process is repeated. The *Nkey*'s symbols are mapped to new ones using the substitution operation. The resulting new string *Skey* is passed to the structure alteration operations for further processing. The resulting string *w* is concatenated with the original *Nkey*. Table 2 shows a sample of keys and their 32-symbol expanded version (*Nkey*).

TABLE II
EXAMPLE OF THE OUTPUT (NKEY) OF OUR METHOD

Key	**32-symbol Nkey (in Hex)**
jHYP_56	77,0,1f,ed,8a,90,6b,f5,63,c0,55,7e,60,7f,88,75,1d,45,7b,4,b2,53,41,f3,5f,4c,48,d,5a,ed,25,93
2A^s7W	26,ec,6a,73,b8,39,f7,ce,2,8f,6c,12,25,c1,cc,11,2c,1,a8,95,9d,de,ab,f8,db,8d,92,1b,43,7,eb,a7
H@m	63,7c,e9,fb,10,1e,90,d1,1a,3,d8,ee,e4,a1,c4,84,e,fe,e8,9b,8,a8,41,80,c5,d8,c6,aa,1b,3c,d8,5a
c@wmrt5	f,1,e6,eb,9,4f,90,76,7c,8e,e9,1,84,60,55,9,a,fe,c3,46,2a,96,54,b4,ba,71,8b,b2,5d,3c,b7,e7
>y_se$8	37,4e,8a,73,e3,5,c5,9a,2f,7e,8f,11,6b,a6,9b,e1,b5,2c,7b,53,9d,d3,b6,f2,8c,6c,4d,a6,5c,11,c5,7a

4) Two-Dimensional Mesh Based Mapping: This section introduces our proposed mesh-based mapping. We introduce the mesh concept in the mesh operations, and define the mapping using the mesh.

- Two-Dimensional Mesh. The two-dimensional mesh is conceptually an $N \times N$ array. Fig. 7 shows an example of a mesh. We designate the two dimensions as the horizontal and vertical dimensions. We list *N* unicode symbols in each dimension. The indexes of the symbols in each dimension are non-negative integers 0, 1 ... *N*–1. For instance, the position of *b* in the horizontal dimension is 4 and the position of *t* in the vertical dimension is 6. Each cell in the mesh is a point (*x*, *y*), where *x* and *y* are the vertical and horizontal positions of a symbol respectively. The move from point P_i to point P_j has a horizontal or vertical distance depending on whether the direction of the move is along the horizontal or vertical dimension. Regardless of the direction of the move, we can calculate the distance between two points P_i and P_j as the absolute value of the difference between their indexes. Therefore, the *horizontal distance* between P_i and P_j is the difference between their indexes on the horizontal dimension. For instance, the horizontal distance between Q_2 (0, 7) and P_3 (0, 4) is `abs` (7 – 4) = 3. Similarly, the *vertical distance* from P_i to P_j is the absolute value of the difference between their vertical indexes. Each move within the mesh starts from a point and ends at another. We therefore capture this piece of information by the move direction. We annotate the direction of a move with respect to a point P_k by the flag "–" if the move is to a point with a lower index, and by the flag "+" if the move is to a point with a higher index regardless whether the move is along the horizontal or vertical dimension. For example, the move from Q_2 to P_3 is annotated by "–" and from Q_2 to *P*4 is annotated by "+".

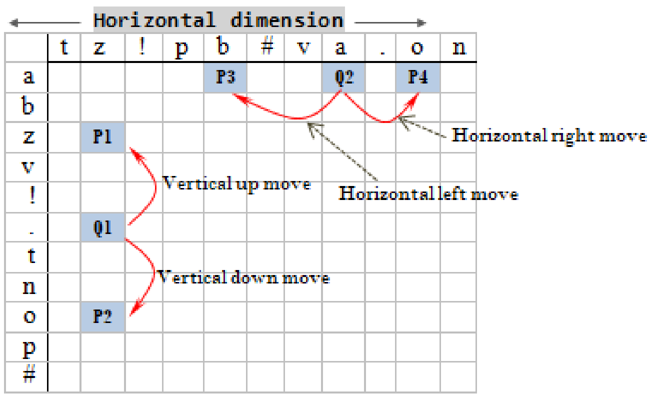


Fig. 7 An example of a mesh.

After introducing the distance and the move direction, we can now define a directive as follows. If the distance of the move from *P* to *Q* is *d* along the vertical or horizontal dimension, we annotate this by "*–d*" or "*+d*" depending on the position of *Q* with respect to *P*. We call "$\pm d$" a *directive*.

- Mesh Operations. We associate two operations with the mesh. The first operation is called shuffle (R), which randomly reorders the symbols of each dimension using a sequence of random numbers R. The second operation is the mapping operation map (t, r), which maps a symbol t to the mesh using the random number r and produces a directive "$\pm d$" as a result of this mapping. The `shuffle` (*R*) operation randomly changes the orders of the symbols in a specific dimension. To reorder the symbols, the shuffling operation places the symbols of each dimension in an *L*×*M* array and makes use of a sequence of random numbers *R* (provided by our random generator) to shift the rows and the columns of this array. Each row *i* is left shifted some positions equal to the random number $R_i \in R$. Likewise, each column *j* is down shifted some positions equal to the random number *CJ*∈ *R*. After the shuffling is performed, the symbols are read row-wise and placed in the dimension of the mesh. Since the symbols of each dimension are reordered using a different sequence of random numbers, the order of the symbols on the horizontal dimension is necessarily different from the order of the symbols on the vertical dimension. Fig. 8 illustrates the reordering operation for the dimension symbols "ABCDEFGHIJKLMNOP." The mapping operation `map` (*t*, *r*) takes a symbol *t* and a random number *r* as an input and returns a directive as an output. The random number *r* has dual objectives. First, the mapping operation uses *r* to select the mapping dimension for mapping the input symbol *t*. For this paper, the map operation selects the vertical dimension if the random number *r* is even and selects the horizontal dimension otherwise. Second, the mapping operation XORes the random number *r* with the resulting distance *d* to generate the distance part of the directive.

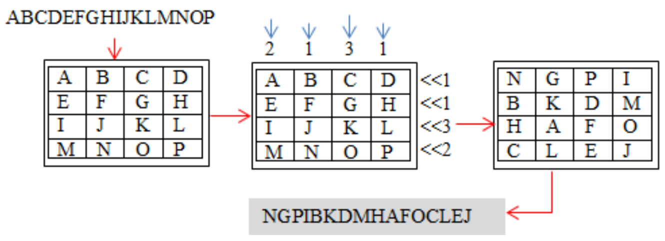


Fig. 8 An example of random reordering of the dimension symbols

- Mesh-Based Symbol Mapping. Based on the definition of the directive and the associated operations, we define mapping a symbol t to the directive "±d" as follows. The mapping operation `map` (t, r) begins from some starting point (x, y) within the mesh's boundaries and uses the random number r to determine the mapping dimension. After determining the mapping dimension, the mapping operation moves along this dimension to the position of the symbol t. Let us suppose that the amount of the move is v. The distance of the move d is the result of XORing v with the random number r (that is, d=v ⊕ r). The distance of the move and its direction ("±") concerning the starting point are compiled into a directive "±d" for that symbol. For instance, if the amount of the move is 10, the direction of the move is to the lower indexes (flag "–"), and the current random number is 45, the mapping outputs the directive "–39" (39 = 10 ⊕ 45).

5) *Tuple Mapping:* We utilize the mesh mapping to map an n-place tuple of symbols to a single directive. Let $T^1, T^2, \dots, T^k$ be n-place tuples. First, the tuple mapping operation obtains a random point (h, v) within the boundary of the mesh using our random generator. The mapping operation reads the first symbol of the first tuple T^1 and maps it to mesh starting from the random point (h, v), yielding a directive $\pm d_1^1$. The starting point is updated based on the position of the recently mapped symbol in the mesh. Likewise, the second symbol of the tuple is mapped starting from the recently updated starting point to yield the directive $\pm d_2^1$. Mapping all the symbols of the tuple yields a sequence of n directives, say, $+d_1^1, -d_2^1, \dots, +d_n^1$. For any subsequent tuple T^i, the mapping operation maps the symbols of the tuple starting from the most recently updated starting point.

The final result of mapping a tuple T^i is computed as follows. The distances d_j^i of the directives $-d_1^i$ $-d_2^i, \dots, +d_n^i$ are all XORed to produce the value *S*. The flags that represent the directions ("+" or "–") are multiplied, and the outcome of the multiplication is the sign for the final directive. For instance, assume that mapping a 3-place tuple produced the directives +10, –6, +30. The final result of the mapping for this tuple would be – (10 ⊕ 6 ⊕ 30 = 18) = –18.

Fig. 9 shows an example of mapping the tuple <e, a> to the mesh. We assume that the mesh has only a few symbols to simplify the example. Assuming the random numbers are

17, 32. First, the tuple mapping considers the first symbol "e" in the tuple. It starts with the randomly selected starting point, says (3, 5). Since the random number 17 is odd, the mapping selects the horizontal dimension as a mapping dimension and moves to the position of "e" in this dimension. The distance of the move is three cells to the lower index, and therefore the mapping produces the directive "–18" as the result of the mapping "e." The starting point is now updated to (3, 2). The next symbol in the tuple is "a," and the next random number is 32. Since the random number 32 is even, the mapping selects the vertical dimension as a mapping direction. It moves to the position of "a" in the vertical dimension. The distance of the move is two cells to the higher index. Therefore it produces the directive "+34" as the result of mapping the symbol "a" to the mesh. The final result of mapping the tuple to the mesh is the result of XORing the two distances "18" and "34" and multiplying the two signs "–" and "+." The result is –48 (18 $\oplus$ 34 = 48, and "–" $\times$ "+" = "–").

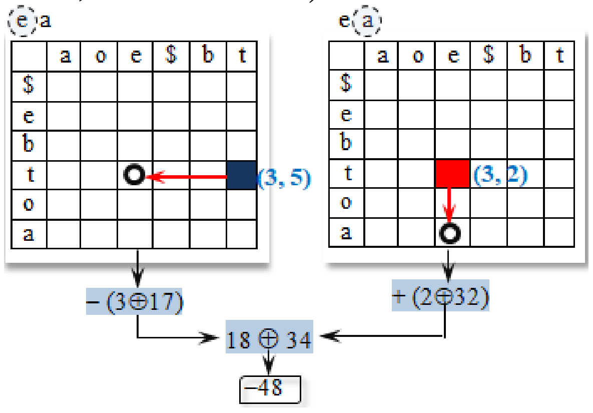


Fig. 9 A tuple mapping example.

6) Security Code Generation: After describing all the operations of the security code generator in Fig. 2, we propose our technique for generating the security code. The security code generation creates a security code of m directives from a document $\mathcal{D}$ and a user-provided password PW. Each directive is created by mapping an n-place tuple to the mesh. The generation process proceeds as follows (see Fig. 2). First, the system extends the user-provided password PW to k symbols using the expansion operation (Subsection 4.3). In our approach, we put no restriction on the length of the user-provided password. That is, the users can provide passwords of any length (one symbol or more). We should emphasize, however, that although our system is capable of expanding the user-provided password to any arbitrary length, users must be aware that short passwords certainly involve high-security risk.

Next, the system hashes the content of the document $\mathcal{D}$ using any appropriate hash algorithm and produces the hash value H_v. (We used SHA-512 in our proof of concept prototype.) The security code generation operation expands the hash value H_v into multiple of n using our expansion operation. Namely, the generation process expands the hash value H_v into $m * n$ symbols. The hash value expansion has dual goals. First, large hash values allow for using tuples with a large size, which causes the results of the tuple mapping to highly diverse. Second, the resulting hash value depends not only on the document's content but also on the effective expansion operation. In this case, the content of the document is more effectively hidden. The expanded hash value is passed into the change amplifier so that any change in the hash value propagates to every symbol in the expanded hash value. The output of the change amplification operation is the hash value H_v^a.

The system then splits H_v^a into n-place tuples $T^1, T^2, \dots, T^m$. The tuple T^1 is a subsequence of H_v^a that consists of the symbols from index 1 to index n; tuple T^2 consists of the symbols from n+1 to 2n, and so on. These tuples are mapped to the mesh as discussed in subsection *E*. The resulting sequence of m directives is further secured by salting its directives with random noise that comes from the random generator. To maximize the confusion, the random noise is generated and added to each directive as follows. First, the system prepares an additional instance of our feedback-based random generator by seeding this instance with the original hash value H_v. We confine the output of the additional instance of the random generator to be within the interval [0, 1] so that we can use its output as probability values and use it as a selector. Specifically, for each random number r_i generated by the expanded password-seeded random generator, we generate a random number q_i from the hash-value-seeded random generator and select r_i only if q_i is less than a pre-specified value p. The selected random number r_i is XORed with the distance value of the current input directive. The value p is specified randomly by taking the first non-zero random number generated by the hash-value-seeded generator.

TABLE III

A SAMPLE OF THE SECURITY CODE GENERATOR OUTPUT

Password (expanded): e3fab9b8b2921443d118665be1836b66b05d9f95a1976d2f7126dc74a2a343	
Hash Value	457C3D142EDA5EC307435C1BB569D253716D4EC2F74CDA00F265DE5F1A01EC45276BD 8DC5F75703A053EDECCB4756DA4A0DBF38190AEBDB86B50F0DF2A013E16
Security Code	+132-162+206+115-232-27-134+41+143+252-235-33+217-90+229+211+187-94+80+241-80-126-160+171-156-108+79+71+6+18+61-186
Encrypted Security Code	AC621A4D852D03C65E7E8AB31C491F9E6C73FEE6E47DD10618FA56E14E28F9347A7101 D3F8CDAC127D94FA8E6F34476F77AF8656E41AEBE61059B40083387F76A6B6A224E974 2E94EF18C9007419A85F0C550D588F4017C5FF6601546D46D344C1F1C14A3A95975836E4 0B5C84F814924605002A37C16EA0D8C8B1BBD6358B81
Hash Value	45**8**C3D142EDA5EC307435C1BB569D253716D4EC2F74CDA00F265DE5F1A01EC45276BD 8DC5F75703A053EDECCB4756DA4A0DBF38190AEBDB86B50F0DF2A013E16
Security Code	+122-212+67-247+175+80+198+222+148-55+123+173+129+143-190+166-49+43+141-31+22+9-203+20+182+164+50-138-10+1-157-45
Encrypted Security Code	B635850083EE9366182E07674900F47F8BB7924B30BA919341061FA3F64033FA95C3F3B68 F98FFC93D1D4CC3C1BC1DAE8D3C9183691F69BF65CCB8A96F96C4F88F6A75B565D536 412028F6E2940F1E66A238C1A8B87062A2293FD3A8AA0313C987C7B340C1B6C6B789031 92E5D8C9B8D2E725A7D3EA67314878357F51FC331BC
Hash Value	457C3D142EDA5EC307435C1BB569D253716D4EC2F74CDA00F265DE5F1A01EC45276BD 8DC5F75703A053EDECCB4756DA4A0DBF38190AEBDB86B50F0DF2A**1**13E16
Security Code	+86+253-224+154+103+59-59+124-247-0-100-65+190+226+206+144+208+92-32+93-84-212-222-31-148+93+97+92-214-209-212-156
Encrypted Security Code	2BEDC3EE5ADB97138B1A995C0EEAEF2B197A4BC15DFD3E6FA73814B148C28B520169 5C5A8290CADEB27CAADDB6E46C0D5786B549821A717625309B7055626754C009559AE DB6C0EB26520965070AE92B8BC4F9593020FBC99D42BBFB010383D4C42A48D057878E D9D5F53646A98FAF42474E19FC7DDFB5E2BC5B90644D2D752506

Table III shows the result of three runs of our method. The table shows the expanded key and three hash values along with their plain and encrypted security codes. The hash values differ in only one bit. It is important to note how our method highly magnifies this minor change in the hash values and propagates it to all of the symbols of the security code. Consider row 2 and row 3 for instance. The hash values differ in only one bit (7 changed to 8). The different symbol is underlined and **boldfaced**. With a simple comparison, it is clear that the generated security codes (whether plain or encrypted) are largely different. The same conclusion can be drawn by comparing rows 2 and 4 or rows 3 and 4.

III. RESULTS AND DISCUSSION

We evaluate our proposed security system in this section. We analyze the performance of both the random generator and the security code generator as the two fundamental components on which all our system is built. For the random generator, we analyse the randomness properties of the output sequences and the correlation between the output sequences that are created using different initial inputs. For the security code generator, we analyse the impact of the changes to passwords and hash values on the security code. Effective security code generator should respond to the changes regardless of their magnitude or position in the input by making drastic changes to the output.

A. Random Generator Performance Analysis

We tested the performance of our random generator using two tests: randomness test and correlation test. The randomness test is used to check whether (1) the output of the generator deviates from randomness and (2) whether different inputs affect the randomness properties of the output. The correlation test checks if different inputs to the generator would ever produce sequences that are correlated.

Because all the inputs to the random generator are passwords and hash values (Unicode strings), the input strings to the generator will be logically Unicode strings. To simulate the different password lengths that may be provided users, we used input strings with 3, 5, 6, and 8 Unicode symbols. The input strings are prepared using three sources. The first set of inputs are created using a password generator (online tool [24]). The second set consists of inputs with different lengths obtained from users (our students). The third set is created from the strings of the previous two sets by making a minor modification to randomly selected symbols of their strings. For each randomly selected string, we randomly selected one, two, and three symbols of the string and randomly changed a bit in the selected symbols. Therefore each string in the previous two sets produces three more mutated strings, and these three mutated strings differ slightly from the original one (in one, two, and three bits). The total number of strings in the three sets is 6,604.

We expanded each string in the three sets to 32 symbols using our expansion operation. We then fed this expanded string as an input to the random generator, which produced a sequence of length 25,000 numbers. (25,000 random numbers are far more that we need in our system). All the numbers in the sequences are integers from 0 to 255. We then subjected these sequences to two important statistical tests: randomness test and correlation test.

1) Randomness Test: Randomness test checks if the generated sequences deviate from randomness. We used three tests from the battery of randomness tests recommended by the National Institute for Standards and

Technology (NIST) [25][26]. These tests are **Runs Test**, **Frequency Test (Monobit)**, and **Discrete Fourier Test** (**Spectral**). These three tests require their input to be in binary rather than decimal numbers. We, therefore, transformed the sequences to binary representation. Since every number in the sequence is between 0 and 255, the binary representation of each random number is 8 bits. These 8 bits are obtained by finding the binary equivalent of the number and padding zeroes to the most significant part of the binary representation if it is fewer than 8 bits. For instance, if the number is "12", its binary representation is "00001100". We applied these three tests to sequences of different lengths. The shorter sequences are prefixes of the larger sequence 25,000. (E.g. we selected the first 1,000 random numbers to form a subsequence of 1,000).

TABLE IV
RANDOMNESS TEST RESULTS

Sequence Length	Runs Test			Monobit Test			Spectral Test		
	Min p-val.	Aver. p-val.	Max p-val.	Min p-val.	Aver. p-val.	Max p-val.	Min p-val.	Aver. p-val.	Max p-val.
1,000	0.50	0.67	0.91	0.33	0.39	0.44	**1E–12***	**0.0057***	0.28
5,000	0.23	0.58	0.81	0.41	0.53	0.64	**4.1E–6***	0.081	0.14
10,000	0.66	0.82	0.88	0.18	0.41	0.63	0.08	0.19	0.45
15,000	0.70	0.92	0.99	0.51	0.57	0.62	0.11	0.22	0.35
20,000	0.69	0.83	0.89	0.41	0.71	0.84	0.21	0.36	0.49
25,000	0.58	0.88	0.96	0.48	0.69	0.77	0.32	0.508	0.79
* Means the corresponding test is significant; that is the corresponding sequences deviate from randomness									

Table IV shows the results of the randomness tests. The leftmost column represents the size of the sequence (how many numbers). All the sequences with sizes of fewer than 25,000 are prefixes of the large sequence (25,000). The boldfaced p-values mean that the corresponding test is significant (the sequence is not random). As Table IV shows, all of the sequences passed the **Runs** and the **Monobit** randomness tests since the corresponding p-values (minimum, average) are greater than our threshold 0.05. All the sequences of length10,000 or longer passed the **Spectral** test. Table IV shows that some sequences did not pass the **Spectral** test. 23 (out of 6,604) sequences of size 5,000 numbers did not pass **Spectral** test. Furthermore, 1,503 sequences of size 1,000 did not also pass **Spectral** test.

We did not observe significant differences in the randomness between the sequences generated using long input strings (6, 8 symbols) and those generated using shorter strings (5, 3 symbols). We should emphasize, however, that although the length of the string does not affect the randomness properties of the sequences, from security standpoint short strings, if used as passwords, involve high-security risks.

2) Correlation Test: The correlation test examines the impact of changing input strings on the relationship between random sequences. In particular, we want to test if changes to the input strings (major or minor) create a correlation between the number sequences. We used the sequences generated in the previous subsection (A.1) to study the correlation. The sizes of the sequences are 5,000, 10,000, 20,000, and 25,000. We computed the Pearson correlation between pairs of sequences with the same size and their corresponding p-values using MINITAB statistical software package [27].

TABLE V
RESULTS OF THE CORRELATION TEST

Sequence	Correlation				P-value		
	Average	St.Dev	Min	Max	Min	Average	Max
5000	0.01060	0.00723	0.002	0.021	0.124	0.702	0.886
10000	0.00177	0.007001	–0.0016	0.0452	0.178	0.591	0.803
20000	0.000456	0.005889	–0.016	0.01300	0.0630	0.6879	0.989
25000	–0.00225	0.00681	–0.017	0.00700	0.092	0.5922	0.989

Table V shows the average value of the correlation, the standard deviation, the minimum, and maximum. The same table also shows the average, minimum, and maximum p-values. All the correlations values are close to zero. All of the p-values are greater than the specified significance level (0.05). According to the p-values, the correlation values are not statistically different from zero. That is, according to the tested sequences, changing the input strings does not create a correlation between the sequences. This result is very significant because it supports the randomness of the output of our random generator regardless of the input string. It also means that neither the length of the input string nor its difference from others results in correlation among the sequences (the outputs of the random generator).

B. *Security Code Generator Performance*

We evaluate in this section the performance of the security code generator. We measure the performance regarding the sensitivity of the security code generator to the input's change. To be effective, changes to the generator's input must cause large changes to its output regardless of the input change's magnitude (single bit or more). This aspect is called the *avalanche effect*. If the avalanche effect is poor (changing bits in the input does not entail significant changes to the output), the intruders may make predictions about the input (especially passwords). The performance evaluation involves three important experiments that give clear indications about the performance. These experiments include.

- Measuring the impact of changing the passwords on the performance while neutralizing the effect of the file content.
- Measuring the impact of changing the file content on the performance while neutralizing the effect of the password.
- Measuring the impact of freely changing the file content and the passwords on the performance.

These experiments aim at covering all the possible scenarios that affect the generated security code. That is because the only two inputs that can vary and affect the security generation process is the password and file content. Therefore, our experiments vary the values of passwords and file contents and measure the impact of this variation on the performance.

1) *Password Change Impact:*

This section discusses the impact of changing passwords on the produced security code. For this purpose, we fix the file's content and change only the passwords. To reasonably cover the possible scenarios, we prepared the passwords as follows. We asked our colleagues and students at the university to provide us with passwords of length 3, 5, 6, 8 symbols. We received a total of 280 passwords. The second group consists of 10,000 passwords, which are obtained from an online password generator. The third group is created by making random changes to some bits of the passwords in the previous two groups. We used the computer built-in random generator to select bits from a password and flip them randomly. These random changes affect one bit, two bits, three bits, four bits, five bits, six bits, ten bits, twenty bits, and thirty bits. Observe by making a minor bit change; we measure the sensitivity of the security code generator to these minor changes. The effective generator would translate these minor changes in the input into a remarkably large change to the output. The total of all passwords is 15,230 different passwords.

TABLE VI
PASSWORD VARIATIONS AND THEIR CORRESPONDING SECURITY CODES

Changes	**Passwords**	**The corresponding security code**
0 bit	f5}<t@J]	+132-162+206+115-232-27-134+41+143+252-235-33+217-90+229+211+187 -94 +80+241-80-126-160+171-156-108+79+71+6+18+61-186
1 bit	f5}<t@**I**]	-150+157+224+136+27+143+44-227-247+43-143-218-240-55+128+29+86 +123 +222+218-218-48+42+155-161+177-171-131+24+99-128-49
1 bit	**g**5}<t@J]	-192-160+108+181-250-201+124+0+69+235+210+111-144+7+72-218+79 -220-86-220+216-171-174-5+141-237+58-27-227-15-12-192
2 bits	**g**5}<**s**@J]	-51-90-143-95+72-228-148+131+129+169-123-37-33+194+11-30-152+129 +57+52-19-88 +237-198+199+53+94-115-26-9-214-209
3 bits	**g**5}<**s@K]**	-233+46+132+50-135+25-198-125+85-95+138+36-82-182+224-33+92-86 +190-163-121 +131-206+222-231-210+225+189+240+171-133-83
4 bits	**g6**}<**s@K]**	-45+220-12-32-75+253+16-128+34-12-8+108+113+247-20-169-45+160-244-6 +26+96-38-20-57-76+146-211-69-213+52+156
All bits	mB&j4n	+28-71-187+91-240+95-183-166-39-180-225+205-86-39+150+235-120+49 -109+159+239+137+187+229-120-5+146+27-201+33+64-21

Table VI shows an example of a password (row 1) and the same password after changing one or more bits. Observe that changing a bit or more resulted in remarkable changes to the corresponding security code. For instance, a security code that generated from the original password (row 1) is totally different from that generated from the original password with one bit changed (row 2). Additionally, the position of the change plays a significant role. Compare the security codes in rows 2 and 3. Both of them generated from passwords that differ from the original password (row 1) by only one bit. The only difference is the position of the changed bit.

We used these passwords and the hash value generated for the file's content as an input to our security code generator. (The file content is just some text copied from Wikipedia.) The generator produced one security code for each password. The length of each security code is 32 directives. These security codes are then pair-wise compared (directive comparison) and count the number of differences. We consider two directives $d_i^1 \in C^1$ and $d_i^2 \in C^2$ in two security codes C^1 and C^2 different if they differ in either the distance part or the flag (the sign + or–). For instance, the two directives +10, +31 are different because their distances are different and the two directives +22 and –22 are different because they differ in the flag.

TABLE VII
RESULT OF PAIRWISE SECURITY CODES COMPARISON (DIFFERENT PASSWORDS, SAME FILE CONTENT)

Group	Number of security codes	Total number of directives	Total of identical directives (pairwise)	Identical security codes
User provided	280	8960	0	0
Generated from Online Tool	10,000	320,000	3	0
Bit Modified	4,950	158,400	0	0

Table VII shows the results of the comparison. We adopt a very rigorous definition of the difference between security codes: two security codes are different if they differ in at least 30 directives out of the 32 directives that constitute each security code. Accordingly, none of the security codes are identical (i.e., all different) as the rightmost column in Table VII shows.

2) File Content Change Impact: This section analyses the impact of the file's content variation on the generated security code. For this objective, we fix the password (used only one user-provided password of length 8) and vary the content of the file. We prepared three groups of files whose sizes range from 1,000 to 20,000 symbols. The first group contains files whose content is copied from web sources (Wikipedia). The second group contains files whose content is randomly generated. The third group consists of files that obtained from the first group after making random changes to symbols of the files. Random changes are made by randomly selecting symbols from a file and changes one of its bits. (The random selection is done using the computer built-in random generator.) The random changes affect one symbol, two symbols, three symbols, and up to one hundred symbols. The random changes are significant because they cause minor differences between the original file content and it has mutated one and thus help measure the sensitivity of the security code generator to the inputs' changes. Effective security code generator must respond to these changes regardless of their magnitude by causing drastic changes to the output. The total number of files is 10,122. We applied the security code generator to these files while using only one password for all of them. In this way, we neutralize the effect of password variations and only focus on the impact of the file content variations.

TABLE VIII
RESULT OF PAIRWISE SECURITY CODES COMPARISON (SAME PASSWORD, DIFFERENT FILES)

Group	Number of security codes	Total number of directives	Total of identical directives (pairwise)	Identical security codes
Plain Files	3,905	124,960	6	0
Random Files	5,000	160,000	21	0
Modified files	1,217	38, 944	2	0

Table VIII shows the result of the pairwise comparison between the different security codes. We use the same rigorous definition of the difference between security codes. Considering the numbers in Table VIII, although there are very few identical directives in some security code pairs, all the security codes are different for all the groups. That, each pair of security codes differs by at least 30 directives (out of 32).

3) Password-File Change Impact: This section discusses the impact file and password change on the performance of the security code generator. To do this, we prepared 1000 different files and 1000 different passwords (all the passwords are generated using an online password generator). We used the security code generator to produce a security code for each file-password. The generated 1000 security codes are pairwise compared. All the 1000 security codes passed our definition of the difference between the security codes. That is, the generator produced different security codes for different files. These experiments indicate that our security code generator is highly sensitive to its input changes. This is evident in the experimental results in the subsections B.1, B.2, and B.3. These test cases indicate that our security code generator has a high avalanche effect. We attribute this high avalanche effect to many effective operations that constitute our security code generator. Specifically, we mention the change amplification operation, expansion operation, the random generator, and effective mapping.

C. *Performance Analytical View*

As discussed throughout the paper, our security code generator uses complicated computations to produce the security code. The computations include non-linear substitutions, complex mapping, structural modifications, hashing, change amplifications, and random noise insertions. These operations highly weaken or even cut any relationship between the generated security code and both the original passwords and the hash values. Reversing the operations and obtaining the passwords or the hash values from the security code is infeasible without knowing the password and hash values. That is because the operations used to create the security code remove any trace of the input from the output (the security code).

The input used for producing the security code comes from the passwords, documents to be protected, and from the random generator whose input depends not only on the password and document but also on the ongoing updates to its state. These updates to the state result from continuous feedback obtained from the generator's output during generating each random number. This means that the security of the documents is not merely maintained by the password, but also by the document itself and other information that involve both passwords and documents.

The security code generator has a high avalanche effect (as shown in subsection *B*). Learning passwords from the security code generator is therefore very unlikely from security standing point. Moreover, the passwords are never stored in the system neither in plain nor in hashed form. The system thus cannot be a leaking point for these passwords. Finally, security codes and hash values are all encrypted using the expanded user password as an encryption key. This adds further protection to the stored information in our system.

Before we leave this section, we make some points. The used benchmarks to test our system are reasonably large. The content of the documents are either random or copied from web resources or created by making micro changes to the documents' contents. The benchmark (passwords, documents) provide reasonable coverage for possible situations. The passwords are selected from different sources: actual users and online password generation tools. Also, to these two sets of passwords, a third one is obtained by making fine-grained random modifications to password in the two sets. The fine-grained random modification guarantees minor differences between passwords of the third set from one side and the original passwords from the other side. As our experiments showed, the security generator maintained a high avalanche effect despite the changes to the passwords, files' content, or both. This promising conclusion gives indications that the high performance of the security code generator does not change with the changes of the input.

D. Related Work

To the best of our knowledge, our protection system is unique. The closest systems to ours are those that use passwords to protect their stored information [1]–[3]. In these systems, users are asked to provide passwords to prevent unauthorized access. The provided passwords are typically stored in their plain or hashed form in a particular file and are used later to gain access to the corresponding document. Current systems never store passwords in understandable form since this would expose these passwords to a great risk of being accessed and used. Current systems instead hash the passwords and store the hashed values as references to use when access to the document is requested. In this way, even if adversaries access the passwords file, they cannot open the documents since these passwords will be rehashed and the result will not match the stored password. It is clear if the adversaries can predict a password from its hash value, they can access the document. The security in these systems hence is solely ensured by the reliability of the hashing algorithm. However, this is not the only possible attack [28]. Aadversaries may use techniques to find a string (not necessarily the same as the user-provided password) that produces the same hash value for some user-password. In this case, the security of the document is greatly jeopardized. Additionally, [29] proposed an effective technique for guessing passwords. As reported in [29], most of the passwords that are stored in password managers can be guessed with a high probability.

Unlike these systems, our system protection works differently. The security code, which is stored with the protected information, depends not only on the password per se, but also on the information to be protected. In this way, rather than having one piece of information (the password), our system ensures the security of information using both the password and document content. Furthermore, our system stores password neither in their plain nor in their hashed form. The only stored piece is the security code, which is produced through a process whose inputs are the password, the document to be protected by this password, and the random noise that is generated using the former two inputs. That is, our system ensures the security of the password by eliminating this password rather than ensuring its privacy using hashing mechanisms.

Other systems [7]–[9] use a slightly similar protection mechanism to ours. The authors [8] proposed a method called *PwdHash* that produces different password using the user password and website information. In such a mechanism, users passwords are protected because even if the hackers were able to receive the user password on a different website, this password would not be useful. Although this idea seems attractive, authors [7] showed that *PwdHash* is susceptible to the different type of attacks. Authors [9] reported several password managers that are used for user authentications. They suggested some enhancements to the security of these managers such as requiring users' actions when they type in their passwords or using some sort of secure filling.

Although there is a superficial similarity to our systems, our system differs in two ways. First, our system uses the document content to consolidate the passwords rather than using website information, which can be cloned. Second, our system adds randomness noises that further diverges the security code from the content of the document and the password.

Password security managers (vaults) have some similarities to our system. Authors in [30] propped an effective method for securing passwords by encrypting the vault using some master key. To protect against the offline hacking, the author suggested the creation of many decoy vaults associated with the real vault. In this case, even if hackers were able to decrypt the vault, they never know whether the vault is the real one or not — authors in [31] proposed honey encryption mechanism to protect password vaults. In such an approach, when hackers attempt to decrypt that vault using the wrong password, the result is plausible-looking decoy plaintexts (passwords) called honey messages. Therefore, hackers have no clue whether they obtained real passwords or faked ones. The proposed vault security approach in [32] follows the same line of protection as in [30] and [31], but intelligently produces a plausible-looking passwords vault for each master password tried to decrypt the vault.

Although these approaches provide a high protection mechanism, authors [29], [33] proposed ways to attack and guess the right password with high probability. In particular, authors [29] showed that even though these approaches purport high protection for password vaults, they have significant weaknesses that can be exploited and consequently recover the correct passwords. In contrast to these approaches, our approach works in a really different way. It uses an "eliminate-to-protect" approach to secure passwords. Therefore, if the passwords are not stored, there is nothing for hackers to recover.

IV. Conclusions

We proposed on the fly authentication scheme for controlling access requests. Although our scheme uses passwords to protect information from unauthorized access, the passwords are kept with the users and are never stored in the system. Rather than directly using passwords for permitting access requests, our system uses the security code as a reference and prompts the users to provide their passwords when they request access to some information. In this case, passwords are highly protected because (1) they are not stored in the system and (2) there is no way for predicting passwords from the security code.

We conducted many experiments to evaluate our approach. We analyzed the performance of the random generator and the security code generator as the two fundamental components in our system. The random generator output (sequences of numbers) passed several important randomness tests recommended by NIST. This output also passed the correlation tests between the sequence pairs. The output of the random generator is unpredictable and depends on the information to be secured (passwords and the data itself). The security code generator has a very high avalanche effect. As we discussed in the performance section, even tiny changes to the input of the security code generator cause tremendous changes to the output. The unpredictability of the random generator and the high avalanche of the security code generator ensure high protection for both the passwords and the documents.

Although our testing benchmarks provide reasonable coverage for all possible scenarios, more comprehensive testing is required. As future work, we will conduct more experiments to estimate our system's performance better.

References

[1] D. Silver, S. Jana, and D. Boneh, E. Chen and C. Jackson, Password Managers: Attacks and Defenses, In Proceedings of the 23rd USENIX Security Symposium (San Diego, CA) August 20–22, 2014.

[2] S-N Hsu and Y-C Hou, A Document Protection Scheme using Innocuous Messages as Camouflage, WSEAS TRANSACTIONS on Information Science and Applications, No. 4, Vol. 6, pp. 694–703, April 2009

[3] C.H. Lin and T.C. Lee, A Confused Document Encrypting Scheme and Its Implementation, Computers & Security Journal, Vol.17, No.6, pp.543–551, 1998

[4] A. Greenberg. Password Manager LastPass Got Breached Hard, June 2015. https://www.wired.com/2015/06/hack-brief-password-manager-lastpass-got-breached-hard.

[5] J. Alex Halderman , Brent Waters , Edward W. Felten, A convenient method for securely managing passwords, Proceedings of the 14th international conference on World Wide Web, May, 2005, Chiba, Japan doi:10.1145/1060745.1060815

[6] J. Bonneau. Guessing Human-Chosen Secrets. PhD dissertation, University of Cambridge, 2012

[7] D. Llewellyn-Jone and G. Ryme, Cracking PwdHash: A Brute-force Attack on Client-side Password Hashing, Proceeding of 11th International Conference on Passwords (Passwords16 Bochum), December, 2016

[8] B. Ross, C. Jackson, N. Miyake, D. Boneh, J. C. Mitchell, Stronger Password Authentication Using Browser Extensions. In 14th USENIX Security Symposium, 2005. http://crypto.stanford.edu/PwdHash/

[9] D. Silver, S. Jana, D. Boneh, E. Chen, C. Jackson, Password Managers: Attacks and Defenses, pp. 449–464. USENIX Association, 2014. https://www.usenix.org/ node/184476

[10] E. Stobert, R. Biddle, Expert Password Management, pp. 3–20. Springer International Publishing, Cham, 2016. http://dx.doi.org/10.1007/ 978-3-319-29938-9_1

[11] B. Ur, F. Alfieri, M Aung, L. Bauer, N. Christin, J. Colnago, L. Faith Cranor, H. Dixon, P. E. Naeini, H. Habib, N. Johnson, W. Melicher, Design and Evaluation of a Data-Driven Password Meter, Proceedings of the 2017 SIGCHI Conference on Human Factors in Computing Systems (CHI '17), May 2017.

[12] K-P. Yee and K. Sitake. Passpet: Convenient Password Management and Phishing Protection. In *Proceedings of the second symposium on Usable privacy and security* (SOUPS'06). ACM, New York, NY, pp. 32–43, 2006. DOI=http://dx.doi.org/10.1145/1143120.1143126

[13] J. Daemen and V. Rijmen. Advanced Encryption Standard (AES), 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf/, November 2001.

[14] W. Stallings, Cryptography and Network Security: Principles and Practice, 7th edition, Pearson publishers, 2016.

[15] M. J. Al-Muhammed, Zitar, R.A., κ–Lookback Random-Based Text Encryption Technique, Journal of King Saud University-Computer and Information Sciences, 2017. doi: https://doi.org/10.1016/j.jksuci.2017.10.002

[16] S. Gueron, S. Johnson, and J. Walker, SHA-512/256, In: Latifi, S. (ed.) Information Technology: New Generations–ITNG 2011. pp. 354–358. IEEE Computer Society, 2011.

[17] Computer Security Resource Center https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf

[18] Different versions of SHA-x, https://csrc.nist.gov.

[19] C. Dobraunig, M. Eichlseder, and F. Mendel. Analysis of SHA-512/224 and SHA-512/256. In International Conference on the Theory and Application of Cryptology and Information Security, pp. 612–630, Springer, 2014.

[20] R. Rivest, The MD5 Message Digest Algorithm, IETF RFC 1321, 1992

[21] V. Lyubashevsky, D. Micciancio, C, Peikert, and A. Rosen, SWIFFT: A Modest Proposal for FFT Hashing, 2008

[22] Pierre LEcuyer. Random Number Generation. In James E. Gentle Wolf-gang Karl Hrdle Yuichi Mori, editor, Handbook of Computational Statistics, Springer Handbooks, chapter 3, pages 35–71. Springer Berlin Heidel-berg, 2012.

[23] G. Marsaglia, Xorshift RNGs, Journal of Statistical Software, 2003

[24] https://passwordsgenerator.net/, accessed March, 17-31, 2018

[25] J. Nechvatal A. Rukhin, J. Soto and et al. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. Special publication 800-22, National Institute of Standards and Technology (NIST), 2010

[26] M. Sýs and Z. Rïha. Faster Randomness Testing with the NIST Statistical Test Suite. In Schaumont P. (eds) Chakraborty R.S., Matyas V., editor, Security, Privacy, and Applied Cryptography Engineering, volume 8804 of Lecture Notes in Computer Science, pages 272–284. Springer, Cham, 2014.

[27] Minitab 17 Statistical Software. Website, 2016. www.minitab.com.

[28] R. Hranický, P. Matoušek, O. Ryšavý, and V. Veselý. Experimental Evaluation of Password Recovery in Encrypted Documents. In: Proceedings of ICISSP 2016. Roma: SciTePress - Science and Technology Publications, pp. 299–306, 2016.

[29] M. Golla, B. Beuscher, and M. Dürmut, On the Security of Cracking-Resistant Password Vaults. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). ACM, New York, NY, USA, 1230–1241. 2016. DOI: https://doi.org/10.1145/2976749.2978416

[30] H. Bojinov, E. Bursztein, X. Boyen, and D. Boneh. Kamouflage: Loss-resistant Password Management. In European Conference on Research in Computer Security, pp. 286–302. Springer, 2010.

[31] A. Juels and T. Ristenpart. Honey Encryption: Security Beyond the Brute-Force Bound. In Advances in Cryptology–EUROCRYPT, pp. 293–310. Springer, 2014.

[32] R. Chatterjee, J. Bonneau, A. Juels, and T. Ristenpart. Cracking-Resistant Password Vaults using Natural Language Encoders. In IEEE Security and Privacy, pp. 481–498, 2015. Available at (April 2018) https://eprint.iacr.org/2015/788, as of August 16, 2016.

[33] M. Dürmuth, F. Angelstorf, C. Castelluccia, D. Perito, and A. Chaabane. OMEN: Faster Password Guessing Using an Ordered Markov Enumerator. In International Symposium on Engineering Secure Software *and Systems*, pp. 119–132. Springer, 2015.