DAFA - A Lightweight DES Augmented Finite Automaton Cryptosystem

Sarshad Abubaker and Kui Wu

Department of Computer Science
University of Victoria, Victoria BC V8P5C2, Canada
{sarshad,wkui}@uvic.ca

Abstract. Unlike most cryptosystems which rely on number theoretic problems, cryptosystems based on the invertibility of finite automata are lightweight in nature and can be implemented easily using simple logical operations, thus affording fast encryption and decryption. In this paper, we propose and implement a new variant of finite automaton cryptosystem, which we call DES-Augmented Finite Automaton (DAFA) cryptosystem. DAFA uses the key generation algorithm of the Data Encryption Standard (DES) to dynamically generate linear and non-linear finite automata on the fly using a 128-bit key. Compared to existing finite automaton cryptosystems, DAFA provides stronger security yet has similar encryption/decryption speeds. DAFA is also faster than popular single key cryptosystems such as Advanced Encryption Standard (AES). The test results on desktop and mobile phones with respect to the running speed and security properties are very promising.

Keywords: Cryptography, Finite Automata, Symmetric key, Probabilistic encryption.

1 Introduction

Smartphones and other portable devices are rapidly changing people's daily lives. More and more sensitive information such as bank accounts, birthdays and health care details are now carried over these devices, which still lag behind desktop PC's in terms of computational capability. Cryptosystems to protect the sensitive information on these devices must be computationally lightweight, or otherwise normal applications would be severely crippled when the main horsepower of the devices is spent on executing security-related primitives.

Most cryptosystems used today rely on problems based on number theory. In this paper, we explore a new type of single key cryptosystem based on the invertibility of finite automata (FA) [15,18]. These cryptosystems have relatively small key sizes and are lightweight in nature. They can be implemented easily in hardware or software using simple logical operations, thus affording fast encryption and decryption [15]. The difficulty in inverting non-linear finite automata and factoring matrix polynomials accounts for the security of these systems.

A.D. Keromytis and R. Di Pietro (Eds.): SecureComm 2012, LNICST 106, pp. 1–18, 2013.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2013

An FA cryptosystem can be implemented as either a public-key system or a single-key system [15]. In the public-key cryptosystem domain, various FA cryptosystems, termed as FAPKC0, FAPKC1, FAPKC2, FAPKC93, FAPKC3 and FAPKC4 [18,15], have been proposed. Some successful attacks have been reported on certain types of FA public-key cryptosystems [2,3,5,6]. However, in the single-key cryptosystem domain, we have not seen any successful attacks on the FA cryptosystems [15].

In this paper, we focus on the single-key FA cryptosystem and further enhance its security while maintaining its fast running speed. We make the following contributions:

1. We design a DES-Augmented Finite Automaton (DAFA) cryptosystem, using DES to dynamically generate linear and non-linear finite automata on the fly. While the core encryption and decryption operations are similar to those used in FAPKC3 [11], DAFA is based on a 128-bit key and the finite automata are generated using a special modification of the key generation algorithm used in DES [12].
2. We implement DAFA over smart phones and thoroughly test its performance. Test results indicate that the statistical properties measured on the ciphertext using DAFA are satisfactory and in the same range as the properties of Advanced Encryption Standard (AES) [9]. We also demonstrate that DAFA is very competitive in terms of speed of operation.

The paper is organized as follows. We begin with a very brief introduction of the basic concepts of FA cryptosystems in Section 2. We present details of the DAFA cryptosystem in Section 3 and test its statistical features and running speed in Sections 4 and 5. We discuss some related work in Section 6 and conclude the paper in Section 7.

2 Background in FA Cryptosystems

We start with the basic definitions [15].

Definition 1. *We define an FA as a five tuple $M = \langle X, Y, S, \delta, \lambda \rangle$, where X denotes the set of all input alphabets, Y denotes the set of all output alphabets, S denotes the set of all states of the finite automaton, δ is the state transition function $\delta : S \times X \rightarrow S$, and λ is the output function $\lambda : S \times X \rightarrow Y$.*

In the context of FA cryptosystems, if we use an FA, M, to encrypt plaintext to ciphertext, we need another FA, M', to recover the plaintext. M' is called the inverse FA of M and its construction is based on the invertibility theory of FA [15].

Definition 2. *FA $M = \langle X, Y, S, \delta, \lambda \rangle$ is said to be (weakly) invertible with delay τ if for any input string $x_0, x_i, \ldots, x_\tau$ and $s \in S$, x_0 can be uniquely determined by the state s and the output string $\lambda(s, x_0.......x_\tau)$.*

Definition 3. *Given two FA $M = \langle X, Y, S, \delta, \lambda \rangle$ and $M' = \langle Y, X, S', \delta', \lambda' \rangle$, states $s \in S$ and $s' \in S'$ are called a matching pair with delay τ if:*

$$\forall \alpha \in X_\omega, \exists \alpha_0 \in X_n : \lambda'(s', \lambda(s, \alpha)) = \alpha_0 \alpha,$$

where $|\alpha_0| = \tau$, X_ω denotes the set of all infinite words of alphabet X, and X_n denotes the set of all finite words of alphabet X. In other words, s' matches s with delay τ.

Definition 4. *M' is said to be a weak inverse with delay τ of M if for any $s \in S$, there exists s' in S' such that (s', s) is a matching pair with delay τ.*

As a special case of FA, we can define its state space $S = (Y_k \times X_h)$, where Y_k and X_h are sets of strings of length k and h, respectively. This type of FA is called (h, k)-order memory FA:

Definition 5. *$M = \langle X, Y, (Y_k \times X_h), \delta, \lambda \rangle$ is said to be an (h, k)-order memory FA, if there is a single-valued mapping ϕ from $Y_k \times X_{h+1}$ to Y, such that*

$$\begin{aligned}
&y(i) = \phi(y_{i-1}, \dots y_{i-k}, x_i, \dots x_{i-h}), i = 0, 1, \dots \\
&\delta(\langle y_{-1}, \dots, y_{-k}, x_{-1}, \dots, x_{-h} \rangle, x_0) = \langle y_0, \dots, y_{-k+1}, x_0, \dots, x_{-h+1} \rangle \\
&\lambda(\langle y_{-1}, \dots, y_{-k}, x_{-1}, \dots, x_{-h} \rangle, x_0) = y_0 \\
&y_0 = \phi(y_{-1}, \dots, y_{-k}, x_0, x_{-1}, \dots, x_{-h})
\end{aligned}$$

What this means is that M needs k previous outputs and h previous inputs to generate the current output. As a special case, if the mapping ϕ is from X_{h+1} to Y, M is said to be an h-order input memory finite automaton.

Example 1. Assume that X and Y are input and output sets of 8-bit characters, respectively. An example (linear) $(1, 2)$-order FA, M, is represented as follows:

$$y(i) = \begin{bmatrix} 0&1&1&1&0&0&0&1 \\ 0&1&1&0&0&1&1&1 \\ 0&0&0&0&1&0&1&1 \\ 1&1&0&1&1&1&1&0 \\ 0&0&0&0&1&0&1&0 \\ 0&1&0&1&0&1&0&0 \\ 1&1&1&0&1&0&0&1 \\ 0&0&1&1&0&1&0&0 \end{bmatrix} y(i-1) + \begin{bmatrix} 0&0&0&0&0&1&1&1 \\ 0&0&0&1&1&1&1&0 \\ 0&0&0&0&1&1&0&0 \\ 0&0&1&1&0&0&0&1 \\ 0&1&1&0&0&0&0&1 \\ 0&1&1&0&0&1&1&1 \\ 1&0&1&1&1&1&0&1 \\ 0&1&0&1&0&0&1&1 \end{bmatrix} y(i-2)$$

$$+ \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix} x(i) + \begin{bmatrix} 0&1&0&1&1&1&0&1 \\ 0&0&1&1&0&0&0&0 \\ 0&0&0&1&1&0&0&0 \\ 0&1&0&1&1&0&0&1 \\ 0&0&1&1&1&1&1&0 \\ 0&1&0&0&0&1&1&1 \\ 0&0&0&1&0&1&0&0 \\ 0&0&0&1&0&1&1&1 \end{bmatrix} x(i-1), i = 0, 1, 2, \dots.$$

The inverse FA of M with delay 1, M', is represented as:

$$x(i) = \begin{bmatrix} 0&0&1&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix} x(i-1) + \begin{bmatrix} 0&0&0&0&0&0&0&0 \\ 1&0&0&1&0&1&1&1 \\ 1&1&0&1&0&0&1&0 \\ 0&1&0&1&1&1&0&0 \\ 0&1&1&1&1&1&0&0 \\ 1&0&0&1&0&0&0&0 \\ 1&0&1&0&0&1&0&0 \\ 0&1&1&0&1&0&0&1 \end{bmatrix} y(i+1) + \begin{bmatrix} 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&1&1&0 \\ 0&0&1&0&0&0&0&1 \\ 1&1&1&0&0&1&1&1 \\ 1&1&1&0&1&1&0&0 \\ 1&0&1&0&1&1&1&1 \\ 0&0&1&0&1&1&1&0 \\ 0&1&0&1&0&0&1&0 \end{bmatrix} y(i)$$

$$+ \begin{bmatrix} 0&1&1&0&0&1&1&1 \\ 1&0&1&1&1&1&1&1 \\ 1&0&0&1&0&1&0&1 \\ 0&0&1&0&1&0&0&1 \\ 0&0&1&0&0&1&0&1 \\ 0&0&1&1&0&1&1&0 \\ 0&1&1&0&1&1&0&0 \\ 0&0&1&0&0&0&0&0 \end{bmatrix} y(i-1) + \begin{bmatrix} 0&0&0&1&1&1&1&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix} y(i-2), i = 0, 1, \ldots.$$

Assume that the input string is $x(0)x(1) =$"AB", i.e., $x(0) = 0X41 = (01000001)^T$ and $x(1) = 0X42 = (01000010)^T$. Assume that the values in the initial state are set as $x(-1) = y(-2) = y(-1) = (00000000)^T$. Since M' is the inverse of M with delay 1, we append an arbitrary character, say $x(2) = 0X0 = (00000000)^T$, to the input string. We can then use M to generate output string (i.e., ciphertext) $y(0) = 0X00, y(1) = 0X01, y(2) = 0X7B$, and we can use M' to recover the input string $x(0)x(1) =$"AB".

The above example is for illustration purpose only. Obviously, in practice, an FA is much more complex and could be linear or non-linear depending on how it is constructed. In a non-linear finite automaton, the degree of the polynomial that constitutes the FA is greater than one. Due to space limit, please refer to [15] for the details on the construction of linear/non-linear FA and the combination of several FA.

3 DES-Augmented Finite Automaton (DAFA) Cryptosystem

3.1 Basic Idea

In the section we present a new version of the single-key FA cryptosystems. Our idea is to apply the key generation algorithm of the popular and widely-used Data Encryption Standard (DES) [12] to the key generation process of FA cryptosystems. The high-level block diagram of DAFA cryptosystem is illustrated in Fig 1. In particular, DAFA operates on 64-byte plaintext blocks, and uses μ pairs of linear and nonlinear FA for encryption and decryption, where μ is a system parameter given by users. It includes three main functional components, namely (a) key processing, (b) generation of automata and starting states, and (c) encryption and decryption, which we will introduce in the sequel.

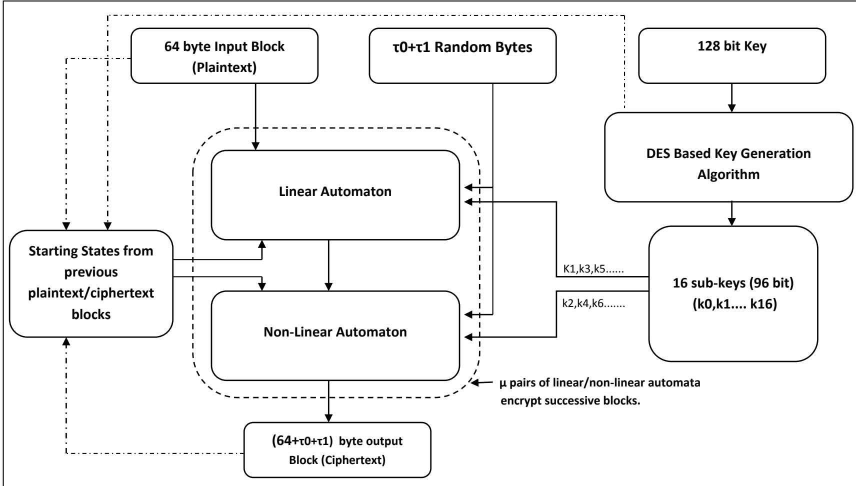


Fig. 1. High level block diagram of DAFA cryptosystem

3.2 Key Processing

We first need to describe the special treatment of the shift and permutation tables. DAFA uses various permutation tables for its operation, similar to the original DES cryptosystem. The permutation tables are randomly chosen. However, we test the tables to ensure that the permuted output is evenly spread across the entire input, and no two bits of the output are derived from the same bit of the input. Care has also been taken to ensure that there are no similar or repeating patterns among any two permutation tables. For the shift table SH-1, the sum total of all left shifts for the sixteen subkeys is 56 to ensure that at the end of the shifting process, the subkeys represent all bits of the main key and that changing even one bit of the main key will significantly affect all sixteen subkeys. An example PC-1 permutation table and an example SH-1 shift table are shown in Table 1 and Table 2, respectively.

Table 1. The PC-1 Permutation Table

57	49	41	33	25	17	9	71	105	108	72	93	78	120
1	58	50	42	34	26	18	75	86	92	104	107	83	111
10	2	59	51	43	35	27	65	102	87	99	69	95	3
19	11	127	60	52	44	36	77	116	94	118	122	74	124
63	55	47	39	31	23	15	89	98	66	112	88	81	126
7	62	54	46	38	30	22	106	113	110	119	115	79	6
14	128	61	53	45	37	29	73	90	84	97	101	114	123
21	13	5	28	20	12	4	85	67	100	80	125	70	91

DAFA is based on a 128-bit (main) key. This key is processed using a key generation algorithm similar to DES. This algorithm creates 16 subkeys, each of which are 96 bits in length and are created using the 128-bit (main) key. These subkeys

Table 2. The SH-1 Shift Table

Key Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Left Shifts	2	2	4	4	4	4	4	4	2	4	4	4	4	4	4	2

are then used to create the finite automata during encryption/decryption. The required starting states are also derived from the subkeys. The steps for creating the 16 subkeys are as follows:

- Step 1: The 128-bit key is initially permuted and shortened to 112 bits, according to the PC-1 permutation table. For example, using Table 1, the first bit of the new 112-bit key is the 57th-bit in the 128-bit key, and the second bit of the new 112-bit key is the 49th-bit in the 128-bit key, and so on till the 91st bit of the original key becomes the 112th bit of the permuted key.
- Step 2: The 112-bit key so formed is now split up into left and right halves, each 56 bits long. We denote these halves as L_0 and R_0 respectively. We now form 16 blocks L_n and R_n for $n = 1, 2, 3, \ldots, 16$. More specifically, L_i and R_i are obtained by left shifting L_{i-1} and $R_{i-1}, i = 1, 2, 3, \ldots, 16$, respectively, according to the shift table SH-1. By left shift, we mean that we move each bit one place to the left, and the first bit is cycled to the end of the block. For instance, according to the first row of the example shift table (SH-1) shown in Table 2, L_1 and R_1 are obtained by left shifting twice of L_0 and R_0, respectively. In this way, we get 16 pairs of subkeys each 56 bits long.
- Step 3: We now concatenate the L_i and R_i pairs ($i = 1, 2, \ldots, 16$) to form 16 subkeys which are each 112 bits long. This 112-bit key is now permuted according to another permutation table PC-2 (e.g., as shown in Table 3). The example in Table 3 permutes each key to a 96-bit key. The bit numbers 9, 18, 22, 25, 35, 38, 43, 54, 64, 72, 80, 83, 96, 99, 102 and 108 are discarded in this process for each of the 112-bit keys. The choice of discarded bits is random, and given that the shift table performs a complete rotation through all 56 bits of each half of the key, this choice does not expose any vulnerability which may aid in cryptanalysis of the cipher. Thus we now have sixteen 96-bit keys generated in a fashion similar to that in the DES cipher.

Table 3. The PC-2 Permutation Table

14	17	11	24	1	5	60	87	82	105	63	70
3	28	15	6	21	10	77	73	98	86	76	57
23	19	12	4	26	8	65	94	106	111	92	81
16	7	27	20	13	2	88	85	57	109	71	66
41	52	31	37	47	55	69	93	110	104	112	78
30	40	51	45	33	48	75	79	103	67	101	91
44	49	39	56	34	53	90	100	62	107	97	68
46	42	50	36	29	32	58	95	74	84	89	61

3.3 Generation of Automata and Starting State

Once the subkeys have been derived, we need to generate the automata which will be used for encryption and decryption. The starting states for these automata will also need to be generated from the subkeys. The steps involved in this process include:

- Step 1: First we need to generate μ pairs of linear and non-linear finite automata for the cryptosystem. These finite automata will be derived using the generated subkeys described above. The linear automaton is (h_0, k_0)-order memory invertible linear FA with delay τ_0, and the non-linear automaton is h_1-order input memory FA invertible with delay τ_1, where all $h_0, k_0, \tau_0, h_1, \tau_1$ are system parameters.
- Step 2: For the linear automaton, we need to generate $h_0 + k_0$ matrices as the component matrices for generating the finite automaton. We also need to generate τ_0 full rank matrices. The specifics of how this can be generated using the subkeys is as follows. For the first $h_0 + k_0$ component matrices, we use alternate subkeys K_1, K_3, K_5 and so on in a circular manner, rolling over to the beginning when we reach K_{16}. Since we need only 64 bits in order to construct an $8X8$ bit matrix, we use three permutation tables M-1, M-2 and M-3 (e.g., as shown in Tables 4, 5, 6) to derive 64 random bits from the 96-bit keys, using the similar operations as those in the PC-1 table (refer to Step 1 in Section 3.2). These three permutation tables are used in sequence in a cyclical manner. As each 64-bit represents an $8X8$ matrix, we therefore have the $h_0 + k_0$ component matrices.

Table 4. The M-1 Permutation Table

8	34	76	13	28	2	56	7
74	20	58	40	73	31	46	79
16	59	1	47	80	91	14	22
4	32	26	55	17	77	82	83
23	65	49	68	35	61	88	95
44	29	19	62	85	5	50	37
71	11	53	38	89	52	94	92
43	64	70	86	25	67	41	10

Table 5. The M-2 Permutation Table

86	65	82	90	49	72	87	13
69	14	89	85	92	4	66	95
27	64	38	80	71	26	91	83
70	3	81	68	63	50	84	94
40	48	10	1	39	78	5	75
28	19	24	25	60	51	61	67
6	46	34	44	52	33	8	59
12	32	18	58	43	7	29	17

Table 6. The M-3 Permutation Table

2	83	88	92	94	48	93	89
74	87	26	35	85	75	16	84
17	47	71	8	29	11	80	25
39	12	30	44	78	53	21	66
56	65	34	76	3	70	43	67
31	22	62	49	52	7	69	79
4	40	15	61	24	42	13	58
38	51	6	20	57	96	33	60

In order to create the τ_0 full rank matrices, a slightly different approach is adopted. The τ_0 matrices are generated using the same method as for the first $h_0 + k_0$ matrices. However, to guarantee that these matrices to be full rank, we need extra processing as follows:

- First we derive the decimal representations of the 8 component bytes that make up each of the matrices so derived and raise them mod 8. If two successive values (mod 8) are the same, then the second value is incremented by 1.
- Next we make the matrices lower triangular (for linear automaton matrices) or upper triangular (for nonlinear automaton matrices) by setting all values in the diagonal to 1 and all values below or above the diagonal to 0. This ensures that our resultant matrices are full rank.
- Finally we use the decimal values derived earlier to carry out two rounds of four row swaps and additions. For example, assume that the 8 decimal values derived are 1,7,3,6,2,0,5 and 4. For round one, we first swap rows 1 and 7 and then add row 6 to row 3. Then we carry out the inverse of this operation, i.e. we now swap the rows 3 and 6 and then add row 7 to row 1 for a total of four row adds and swaps. In round two, we perform an identical operation with the last four decimal values. We first swap rows 2 and 0 and then add row 4 to row 5. Then we carry out the inverse of this operation, i.e. we now swap the rows 5 and 4 and then add row 0 to row 2. Since only basic row swaps and additions are performed, the resultant matrix will be full rank. Using this process, we can create random, full rank matrices for use in construction of the finite automata as normal (refer to [15] for the construction of finite automata using given matrices).

- Step 3: For the nonlinear automaton, we need $h_1 + 1$ component matrices. These are generated as in Step 2, except that they use the even set of subkeys K_2, K_4 and so on in a circular manner, rolling over to the beginning when we reach K_{16}. Also, as before, we use the permutation tables (e.g., M-1, M-2 and M-3) to derive the 64 random bits from the 96-bit keys. We also need τ_1 full rank matrices which are derived in a manner similar to that for the linear automaton. These component matrices, once derived, are used to create the nonlinear finite automaton as normal [15].

– Step 4: After generation of each linear/nonlinear automaton, we derive the starting state for that particular automaton before proceeding to generate the next one. The starting state is derived from alternate subkeys immediately following the last key that was used to generate a particular finite automaton. For instance, if the first linear automaton was generated, (say) using subkeys K_1, K_3 and K_5, then the three 8 bit vectors that will be required as the starting state of this automaton are generated from the sequential keys K_7, K_9 and K_{11}. For this purpose, we use random look up tables (e.g., tables SS-1, SS-2 and SS-3 shown in Table 7). These tables are used alternately in order to provide confusion as to the selection of the 8 bits from the 96-bit subkeys. That is, if SS-1 is used on K_7 to generate the first vector, then SS-2 will be used on K_9, and SS-3 on K_{11}. This cyclical process will continue for each of the starting states required for all μ pairs of linear and nonlinear automata.

Table 7. The Starting State Permutation Tables

SS-1	1	5	9	13	95	91	87	83
SS-2	40	56	9	16	11	91	34	61
SS-3	17	29	32	46	54	65	77	85

Note that the automaton and starting state creation process is designed in a manner to increase confusion and prevent cryptanalysis. This also provides for greater diffusion in the final ciphertext once encryption is performed. Based on our implementation, it has been observed that since they are based on simple bit operations, the generation of automata and starting states takes very little time even for large values of $h_0, k_0, \tau_0, h_1, \tau_1$ and μ.

3.4 Encryption and Decryption

As shown in Fig. 1, for encryption and decryption, the plaintext is split up into 64-byte blocks. Each block is encrypted with a linear and nonlinear automata pair in succession. Since there are μ different linear/nonlinear automata pairs, these are alternately cycled by the algorithm for each successive block.

As illustrated in Section 2, each FA needs to set an initial state (e.g., the values of $x(-1), y(-1), y(-2)$ in Example 1). Clearly, the number of bytes in the initial state depends on the parameters of the FA (i.e., the values of h_0, k_0, τ_0, h_1 and τ_1 in our DAFA cryptosystem). To enhance security, when we use alternative linear/nonlinear FA pairs, we create *dynamic* initial states by allowing each block (except the first one) to use the last portion of the ciphertext in the previous block as the starting state. In this way, if a single bit of the plaintext is altered, the ciphertext undergoes a drastic change.

Example 2. Assume that μ has a value of 2. Then two linear/nonlinear automata pairs are generated by the algorithm, denoted by (L_1, NL_1) and (L_2, NL_2),

respectively. Assume that we split the plaintext into 64-byte blocks. For every eight blocks, denoted by $B_1, B_2, \ldots, B_8$, we can create the ciphertext c as:

$$c = B_1^{L_1,NL_1} B_2^{L_2,NL_2} \ldots B_7^{L_1,NL_1} B_8^{L_2,NL_2}$$

where $B_i^{L_1,NL_1}(i = 1,3,5,7)$ returns the ciphertext of B_i encrypted using L_1 and NL_1 in sequence, and $B_j^{L_2,NL_2}(j = 2,4,6,8)$ returns the ciphertext of B_j encrypted using L_2 and NL_2 in sequence (refer to Example 1 for the operation). Note that the initial states of L_1 and NL_1 when encrypting B_1 is randomly selected, but the initial states of FA when encrypting other blocks use the output of ciphertext from the previous block, i.e.,

$$B_1^{L_1,NL_1} \longrightarrow B_2^{L_2,NL_2} \longrightarrow B_3^{L_1,NL_1} \ldots \longrightarrow B_8^{L_2,NL_2},$$

where $\longrightarrow$ means setting up the initial state.

Decryption is carried out in the reverse order. Assume that the ciphertext is split up into eight 64-byte blocks $C_1, C_2, ..., C_8$. The plaintext p will be generated as follows:

$$p = C_1^{NL'_1,L'_1} C_2^{NL'_2,L'_2} \ldots C_7^{NL'_1,L'_1} C_8^{NL'_2,L'_2}$$

where NL'_i and L'_i are the inverse FA of NL_i and $L_i(i = 1,2,\ldots,8)$, respectively, $C_i^{NL'_1,L'_1}(i = 1,3,5,7)$ returns the plaintext of C_i decrypted using NL'_1 and L'_1 in sequence, and $C_j^{NL'_2,L'_2}(j = 2,4,6,8)$ returns the plaintext of C_j decrypted using NL'_2 and L'_2 in sequence (refer to Example 1 for the operation).

3.5 Features of the DAFA Cryptosystem

The DAFA cryptosystem has some nice features, including:

- It uses a 128-bit key. Unlike the traditional finite automaton cryptosystems, the key consists of a 128-bit string - not a collection of finite automata and starting states. The underlying finite automata and starting states are dynamically generated on the fly using a special modification of the key generation algorithm used in DES.
- The key space is 2^{112} bits long. Though a 128-bit key is used, 16 bits are discarded by the initial permutation, similar to DES. This security level is equivalent to that provided by triple DES, which is commonly regarded as sufficient for most applications.
- A new parameter, μ, is introduced to determine how many linear/nonlinear automaton pairs are to be generated and used for encryption/decryption purposes.
- The plaintext is split up into 64-byte blocks. Each block is encrypted by a linear and nonlinear automaton pair in succession. Further, there are μ different linear/nonlinear automaton pairs and these are alternately cycled by the algorithm for each new block. The size of each block may be user defined if required.

- Since μ can take any positive integer values, cryptanalysis on the resultant cipher is difficult since firstly, each automaton uses the encrypted values of the previous block as part of its starting state and secondly $\tau_0 + \tau_1$ random characters are added at the end of each block for encryption. This leads to probabilistic encryption results [8], as discussed in our later security analysis. The addition of two random characters to each 64-byte block of plaintext results in roughly a 3% increase in the size of the ciphertext. However, this can be reduced, if required, by increasing the size of the plaintext blocks to either 128 or 256 bytes.
- Though DAFA's key generation time is slightly larger (depending on μ) than that of the existing FA cryptosystems, the speed for encryption and decryption remains essentially the same. The security of DAFA, however, is vastly increased due to the introduction of extra randomness via the random characters appended in each block.

4 Security Analysis

The security of the FA cryptosystem has been discussed in [15]. Since DAFA consists of the same core components (i.e., the linear and nonlinear FA) used in these cryptosystems, the security of DAFA is at least as much as that afforded by these cryptosystems. Currently, there is no known attacks successful to break the single-key FA cryptosystems. In addition, the use of different automata pairs and keys along with the block-based encryption scheme introduces further randomness in the algorithm and enhances the security. We conduct extensive tests to illustrate its strong statistical properties.

4.1 Probabilistic Encryption

To be semantically secure and to avoid chosen plaintext attacks, an encryption algorithm must be probabilistic [8]. Nevertheless, most of the commonly-used single key cryptosystems such as DES and AES are deterministic in nature, i.e., given a particular plaintext and a particular key, they always encrypt to the same ciphertext. In order to achieve probabilistic encryption, DES and AES need to use other mechanisms, e.g., working with Cipher Block Chaining (CBC) [7]. In contrast, our DAFA cryptosystem integrates random padding into every block of text encrypted, resulting in a truly probabilistic symmetric encryption scheme that produces a different ciphertext each time encryption is done - even if the plaintext and the keys remain unchanged!

Two main reasons contribute to this nice feature. First, for every 64-byte block of plaintext encrypted, $\tau_0 + \tau_1$ random characters are appended at the end for encryption with the linear/nonlinear automaton pair. This is significant because it produces a *cascading* effect which alters the entire block depending on the random bytes added at the end. Second, $h_0 + k_0$ bytes need to be derived from the 128-bit key as the starting state for the first linear automaton used and $h1$ bytes need to be derived as the starting state for the first nonlinear automaton

used. For the first block of data, these starting states remain constant depending on the key used. However, for all subsequent blocks, part of the encrypted values of the random characters added at the end of the current block are used as the starting state for encrypting the next block of plaintext. Thus, the starting state for the next block is random, resulting in a significant change in the ciphertext even when the same plaintext is encrypted with the same key multiple times.

In general, $\tau_0 + \tau_1$ random bytes are added to each block of plaintext processed which also affects the subsequent blocks since their encrypted values are used as the starting states for encrypting the next block of ciphertext. This results in a completely probabilistic encryption scheme, rendering our cryptosystem semantically secure and indistinguishable under a chosen plaintext attach (IND-CPA) [8]. This means that the ciphertext hides even partial information about the plaintext. This is evident from the statistical tests carried out on the cryptosystem. It adds an element of randomness to every encryption procedure and prevents partial decryption of ciphertext by ensuring that an adversary cannot recover any portion of the plaintext without knowing the decryption key.

4.2 Multiple Keys and Alternating Automaton Types

As we have seen, the 128-bit key is processed by a sophisticated key generation algorithm in order to produce sixteen 96-bit subkeys. These subkeys are used to generate the finite automata used by the algorithm. The keys are used cyclically, and the encryption algorithm uses 2μ different keys to construct different automata, among which half of the constructed automata are linear and the other half are nonlinear. Security is considerably enhanced as a result of these alternating linear/nonlinear automaton pairs. Each pair is applied alternately on successive blocks of text with the encrypted values of one block being used as the starting state of the automaton in the next block. This introduces a high degree of complexity making cryptanalysis difficult.

4.3 Statistical Analysis

We use ENT- a pseudo random number sequence test program [20] to test DAFA cryptosystem. The statistical tests are similar to those in [14]. In order to analyze if some statistical features in the plaintext carry over into the ciphertext, it is advantageous to start with plaintext which consists of highly patterned bytes and which uses a uniform key (an example of a uniform key would be PPPPPPPPPPPPPPPP). All tests have been conducted using a mix of multiple randomly generated as well as uniform 128-bit keys.

All plaintext files are 4KB in size. Four different types of plaintext have been tested. The first type, labeled as *pt1*, consists only of repetitions of the 32-byte sequence AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH. The second type, *pt2*, consists of all zeros. The third type, *pt3*, consists of all ones, and the fourth type, *pt4*, consists of random English text.

A total of 1000 test runs were conducted on the ciphertexts created with different random and uniform keys as explained earlier. The tests were conducted

separately for two levels of security. In the first case, we use $h_0 = 1, k_0 = 2, \tau_0 = 1, h_1 = 2, \tau_1 = 2$ and μ=7. We will refer to this security level as DAFA-121227. The second case uses a security level of $h_0 = 2, k_0 = 3, \tau_0 = 2, h_1 = 3, \tau_1 = 3$ and μ=7, which we will refer to as DAFA-232337. Generally speaking, higher values result in a higher security level but a slower system, because more finite automata are used in the key generation and encryption/decryption processes. We also present the statistical results of the same tests on plaintexts encrypted using AES with a 128-bit key, under CBC mode for ready reference. We will refer to this as AES-128 in the remainder of this section. We conducted five types of test as follows.

Tests for Entropy. Entropy was first introduced by Claude Shannon [13] and is a measure of the uncertainty associated with a random variable. It refers to the expected value of the information contained in a byte of data. In other words, entropy refers to the density of the content or information contained in a file, expressed as a number of bits per character. Files which are extremely dense in information (close to 8 bits/byte) can be considered to be random. Our tests show that files encrypted with the DAFA cryptosystem, even with a lower level of security (DAFA-121227), demonstrate high average entropy (equivalent to that of AES with a 128-bit key under CBC mode) with very low levels of standard deviation. Table 8 shows the results of our entropy tests.

Table 8. Results for entropy tests using AES-128, DAFA-121227 and DAFA-232337

Plaintext Type (Size 4 KB)	Plaintext (bits/byte)	AES-128 (bits/byte)		DAFA-121227 (bits/byte)		DAFA-232337 (bits/byte)	
		Avg	S.D.	Avg	S.D.	Avg	S.D.
pt1 (AAA...HHH)	3.00200	7.95488	0.00407	7.95578	0.00915	7.95793	0.00373
pt2 (00000000...)	0	7.95479	0.00402	7.94977	0.03572	7.95770	0.00381
pt3 (11111111...)	0	7.95485	0.00414	7.94943	0.03465	7.95793	0.00358
pt4 (English Text)	4.87487	7.95477	0.00405	7.95653	0.00394	7.95818	0.00368

Chi Square (χ^2) Tests. The χ^2 test [1] with 255 degrees of freedom is a common test for measuring the randomness of data. The chi-square distribution in our tests is calculated for a stream of bytes and is expressed as an absolute number and a percentage which indicates how frequently a truly random number sequence would exceed the calculated value [1,20]. In our test, we use $\chi^2_{0.01,255}$ to test whether a given data sequence is random. We consider the test successful if the calculated χ^2 value [1] is smaller than the value of $\chi^2_{0.01,255}$.

As mentioned earlier, multiple tests were conducted with different uniform and random keys. Note that the results for the first three types of plaintext are for worst case scenarios where the plaintext is well patterned. The test result for the fourth type of plaintext is what would be the normal case scenario. As before, test results for AES-128 are also presented with DAFA-121227 and DAFA-232337 for ready reference. As is evident from Table 9, the test results for DAFA, even at the lower level of security, are comparable to AES-128.

Table 9. Results for χ^2 tests using DAFA-121227, DAFA-232337 and AES-128

Plaintext Type (Size 4 KB)	Test Runs (Unique Keys)	AES-128 % Tests Passed	DAFA-121227 % Tests Passed	DAFA-232337 % Tests Passed
pt1 (AAA...HHH)	1000	97.9	96.5	97.9
pt2 (00000000...)	1000	98.0	84.1	98.0
pt3 (11111111...)	1000	97.3	85.1	97.4
pt4 (English Text)	1000	97.2	97.2	98.6

Arithmetic Mean Tests. For the Arithmetic Mean (AM) test [20], we add the values of all the bytes in the file and divide it by the file length. If the data is random, this should be about 127.5 since there are 256 possible ASCII values that each byte of data can represent. Almost all results for our cryptosystem show values very close to 127.5. Table 10 shows the average arithmetic mean calculated from 1000 tests conducted on each type of plaintext with different keys.

Table 10. Results for AM tests using DAFA-121227, DAFA-232337 and AES-128

Plaintext Type (Size 4 KB)	Plaintext A.M	AES-128 Avg. A.M.	DAFA-121227 Avg. A.M.	DAFA-232337 Avg. A.M.
pt1 (AAA...HHH)	68.48000	127.52812	127.52098	127.51560
pt2 (00000000...)	47.99000	127.44284	127.46245	127.44099
pt3 (11111111...)	48.99000	127.49717	127.445632	127.433334
pt4 (English Text)	70.64840	127.49981	127.48059	127.57606

Tests for Monte Carlo Value for *Pi* (π). In the test for the Monte Carlo Value for *Pi* [10], successive 6-byte blocks of data are used as the source for plotting the X and Y coordinates within a square, using 24-bits for each axis. The number of points which fall within a circle inscribed in the square is used to approximate the value of *Pi*. As the number of points increases, the value will approach the correct value of *Pi* if the sequence is random [20]. Our tests in Table 11 show high accuracy in the Monte Carlo tests with a very low error percentage in the estimated value of *Pi*.

Table 11. Results for Monte Carlo tests for value of *Pi*

Plaintext Type (Size 4 KB)	Plaintext		AES-128		DAFA-121227		DAFA-232337	
	Pi	Error%	Avg. *Pi*	Error%	Avg. *Pi*	Error%	Avg. *Pi*	Error%
pt1 (AAA...HHH)	4.00	27.32	3.14010	0.0474	3.14077	0.0261	3.14064	0.0302
pt2 (00000000...)	4.00	27.32	3.13981	0.0566	3.13996	0.0518	3.14270	0.0353
pt3 (11111111...)	4.00	27.32	3.13977	0.0579	3.14192	0.0105	3.14153	0.0019
pt4 (English Text)	4.00	27.32	3.14125	0.0108	3.14188	0.0092	3.13651	0.1617

The Serial Correlation Coefficient. The degree to which neighboring bytes are related to each other are measured by the Serial Correlation Coefficient (SCC). The lower the relation, the lower the value of this measure. If the bytes

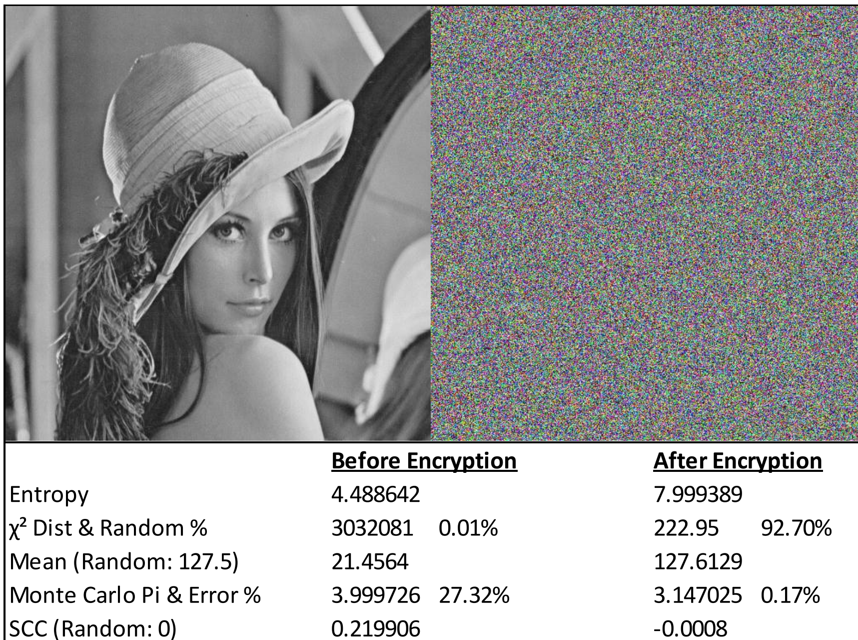
are totally uncorrelated, the SCC would be close to zero [20]. Table 12 shows the results of the SCC tests. We can see that in our results this value almost always converges close to zero for both levels of DAFA security tested. Note that the SCC for pt2 and pt3 is undefined since all values are equal [20].

Table 12. Results for SCC tests using DAFA-121227, DAFA-232337 and AES-128

Plaintext Type (Size 4 KB)	Plaintext S.C.C.	AES-128 Avg. S.C.C.	DAFA-121227 Avg. S.C.C.	DAFA-232337 Avg. S.C.C.
pt1 (AAA...HHH)	0.71926	-0.00031	-0.00015	-0.00019
pt2 (00000000...)	undefined	0.00023	0.00009	-0.00068
pt3 (11111111...)	undefined	-0.00020	0.00012	-0.00044
pt4 (English Text)	0.46831	-0.00026	-0.00074	-0.00005

In addition to the tests over text files, we also conducted the five types of tests of DAFA on a classic image, Lena. The test results are shown in Figure 2.

To summarize, all tests conducted with DAFA are satisfactory regarding the randomness in the ciphertext. As is evident from both the encrypted text and image files, there are no statistical patterns relating the original plaintext or image with the encrypted version.



	Before Encryption		After Encryption	
Entropy	4.488642		7.999389	
χ^2 Dist & Random %	3032081	0.01%	222.95	92.70%
Mean (Random: 127.5)	21.4564		127.6129	
Monte Carlo Pi & Error %	3.999726	27.32%	3.147025	0.17%
SCC (Random: 0)	0.219906		-0.0008	

Fig. 2. Results after encryption of a bitmap image

5 Encryption/Decryption Speed

We implemented DAFA in Java as a proof of concept in order to demonstrate its robust statistical properties and lightweight nature. We tested the performance of DAFA, as well as the AES cryptosystem with a 128-bit key using Cipher Block Chaining(CBC) for reference, on both an Intel Core 2 Duo 2.16 GHz desktop with 3GB RAM and the Nokia N900 Internet Tablet which has an ARM Cortex A8 600MHz processor with 256MB RAM. As shown in Table 13, DAFA greatly outperforms AES in terms of average encryption/decryption speed. The results shown in Table 13 are the average of 50 tests conducted on each file size with each test reflecting the average throughput for one complete encryption and decryption cycle. Note that since the DAFA encryption and decryption algorithms have similar operations on finite automata, the throughput for both are nearly identical.

Table 13. Average encryption and decryption throughput

Platform	Filesize (KB)	AES-128 (Kbit/sec)	DAFA-121227 (Kbit/sec)	DAFA-232337 (Kbit/sec)
Intel Core 2 Duo 2.16Ghz, 3GB RAM	4	516.13	2909.09	2133.33
	100	7476.64	12500.86	11940.30
	1024	47080.46	49652.76	47851.74
Nokia N900 600MHz, 256MB RAM	4	57.45	592.59	542.37
	100	1219.51	1523.81	1338.35
	1024	3955.63	4264.45	4016.32

Performance in Java is largely dependent on the JVM (Java Virtual Machine) implementation. The code is initially interpreted but parts are compiled at runtime using JIT (Just in Time) compilation [4] to boost performance for computationally intensive code. This is the main reason why generally we find that a Java program starts off slower when it is being interpreted and then rapidly picks up speed as the JIT compilatio n occurs. This can be seen in the performance results for both the AES as well as DAFA programs, where the throughput achieved is lower for the smaller files as compared to the larger ones.

The AES implementation in our test uses the Java JCE (Java Cryptographic Extension) library which is highly efficient and has been carefully optimized. Standard libraries in Java are largely programmed using native code which is much faster in terms of performance. In our profiling tests using the *-Xprof* option in Java, we found that roughly 25.58% of the computation time for AES encryption/decryption were handled by native method calls as compared to only 3.14% in our DAFA implementation. This is why with files of larger sizes, the relative performance of AES with respect to DAFA improves. Despite these facts, DAFA is very competitive in terms of speed even without speed and memory optimization, on both test platforms.

6 Related Work

Finite automata based public key cryptography termed FAPKC0 was introduced in [16] in 1985. Since then various public key cryptosystems based on finite automata have been proposed like FAPKC1, FAPKC2, FAPKC93, FAPKC3 and FAPKC4 [17,18,19]. In contrast to this, fewer single key cryptosystems based on finite automata have been proposed though the underlying theory behind both are similar. An excellent source for comprehensive information about both single and public key cryptosystems based on finite automata is [15]. A few attacks and suggestions on how to avoid them have been proposed in [2,3,5,6] for public key cryptosystems. However, in the single-key cryptosystem domain, we have not seen any successful attacks so far [15]. For a detailed and clear discussion about construction of finite automata required for public key cryptosystems, on which DAFA is based, readers are referred to [11]. DAFA presents an improvement on these schemes by firstly utilizing a compact key to generate finite automata on the fly and secondly by utilizing μ pairs of different linear and non-linear finite automata for encrypting successive blocks in order to increase the security afforded by the overall system.

7 Conclusion

In this paper we proposed and implemented a new variant of finite automaton cryptosystem, termed as DAFA, which uses DES to dynamically generate linear and non-linear finite automata on the fly using a 128-bit key. We conducted comprehensive statistical as well as running speed tests of DAFA on a desktop computer and on a smartphone. DAFA demonstrates strong security properties comparable to 128-bit AES, and it runs faster than AES. While our current DAFA implementation is based on Java as a proof of concept, we believe that there is large room to further improve its running speed if memory and code optimization were conducted or if implemented in assembly or C language. We expect that FA based cryptosystems, particularly the augmented variants such as DAFA, will earn credibility in the applied cryptographic world as a viable alternative to current cryptosystems and stand the tests of further cryptanalysis.

References

1. Banks, J., Carson, J.S., Nelson, B.L., Nicol, D.M.: Discrete-Event System Simulation, 5th edn. Prentice Hall (2010)
2. Bao, F., Igarashi, Y.: Break Finite Automata Public Key Cryptosystems. In: Fülöp, Z. (ed.) ICALP 1995. LNCS, vol. 944, pp. 147–158. Springer, Heidelberg (1995)
3. Bao, F., Igarashi, Y., Yu, X.: Some results on decomposition of weakly invertible finite automata. IEICE Transactions on Information and Systems E79-D, 1–7 (1996)
4. Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., Wolczko, M.: Compiling java just in time. IEEE Micro 17 (1997)

5. Dai, D.W., Wu, K., Zhang, H.G.: Cryptanalysis on a finite automaton public key cryptosystem. Science in China, Ser. A 39, 27–36 (1996)
6. Dai, Z.-D., Ye, D.F., Lam, K.-Y.: Weak Invertibility of Finite Automata and Cryptanalysis on FAPKC. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 227–241. Springer, Heidelberg (1998)
7. Ehrsam, W.F., Meyer, C.H.W., Smith, J.L., Tuchman, W.L.: Message verification and transmission error detection by block chaining. US Patent 4074066 (1976)
8. Goldwasser, S., Micali, S.: Probabilistic encryption and how to play mental poker keeping secret all partial information. In: Annual ACM Symposium on Theory of Computing (1982)
9. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall, CRC (August 2007)
10. Mathews, J.H.: Monte carlo estimate for pi. Pi Mu Epsilon Journal 5, 281–282 (1972)
11. Meskanen, T.: On finite automaton public key cryptosystems. Technical Report 408, TUCS - Turku Centre for Computer Science, Turku, Finland (2001)
12. U.S. Department of Commerce National Bureau of Standards. Data encryption standard. Federal Information Processing Standard (FIPS), Publication 46 (1977)
13. Shannon, C.E.: A mathematical theory of communication. Bell System Technical Journal 27, 379–423 (1948)
14. Hermetic Systems. Testing the me6 cryptosystem, `http://www.hermetic.ch` (accessed in December 2011)
15. Tao, R.J.: Finite Automata and Application to Cryptography. Tsinghua University Press (January 2008)
16. Tao, R.J., Chen, S.H.: A finite automaton public-key cryptosystem and digital signatures. Chinese Journal of Computers 8, 401–409 (1985) (in Chinese)
17. Tao, R.J., Chen, S.H.: Two varieties of finite automaton public key cryptosystem and digital signatures. Journal of Computer Science and Technology 1, 9–18 (1986)
18. Tao, R.J., Chen, S.H.: A variant of the public key cryptosystem fapkc3. Journal of Network and Computer Applications 20, 283–303 (1997)
19. Tao, R.J., Chen, S.H., Chen, X.M.: Fapkc3: A new finite automaton public key cryptosystem. Journal of Computer Science and Technology 12(4), 289–304 (1997)
20. Walker, J.: Fourmilab ent - a pseudorandom number sequence test program, `http://www.fourmilab.ch/random` (accessed in December 2011)