Probabilistic, Lightweight Cryptosystems based on Finite Automata

by

Sarshad Abubaker
M.Sc. Information Technology, De Montfort University, Leicester, U.K., 2009

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

M.Sc. Computer Science

in the Department of Computer Science

Probabilistic, Lightweight Cryptosystems based on Finite Automata

by

Sarshad Abubaker
M.Sc. Information Technology, De Montfort University, Leicester, U.K., 2009

Supervisory Committee

Dr. Kui Wu, Supervisor
(Department of Computer Science)

Dr. Jianping Pan, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Kui Wu, Supervisor
(Department of Computer Science)

Dr. Jianping Pan, Departmental Member
(Department of Computer Science)

ABSTRACT

Most of the cryptosystems currently used are based on number theoretic problems. We focus on cryptosystems based on finite automata (FA) which are lightweight in nature and have relatively small key sizes. The security of these systems relies on the difficulties in inverting non-linear finite automata and factoring matrix polynomials.

In symmetric or single key encryption, the secret key consists of two finite automata and their inverses. By applying the inverses of the automata to the cipher text, the plain text can be effectively calculated. In case of asymmetric or public key encryption, the public key consists of another automaton, which is the combination of the two finite automata while the private key consists of the inverse of the two individual automata. It is hard to invert the combined automaton without the knowledge of the private key automata. We propose a third variant which is based on a 128-bit key and uses a DES-based key generation algorithm.

We implement and test all three cryptosystems - the standard single key and public key cryptosystems as well as our novel DES-based FA cryptosystem. We also extensively test the finite automata cryptosystems on a standard desktop machine as well as the Nokia N900 smartphone. All statistical tests carried out on the ciphertext are satisfactory.

Contents

List of Figures

ACKNOWLEDGEMENTS

The last two years at the University of Victoria have been an intensely rewarding and educating experience for me. I feel I have made a lot of progress in my chosen line and I am thankful to my supervisor Dr. Kui Wu for giving me this wonderful opportunity and for opening a window to the world of cryptography, which fascinates me. No words can express sufficiently, the gratitude I feel for the countless hours of patient tutoring and constant correction, without which this thesis would not have been a reality today.

During the course of my studies, I worked with the PANDA Research Laboratory on their Nokia Social Networks project. During this period, I learnt mobile programming for various Nokia smartphones and thoroughly enjoyed myself. I would like to thank Dr. Jianping Pan for his constant encouragement and good wishes along the way. Had it not been for the knowledge I gained, I might not have been able to test my cryptosystem on the Nokia N900 internet tablet.

I am thankful also to my family, friends and well wishers who have been a consistent source of support and encouragement for me. Without you, I may not have been able to accomplish the journey I undertook. Thank you for tiding me over my darkest phases and gently egging me on to achieve my goal.

Last but definitely not the least, I would like to thank my parents. My father, who aroused in me the love for reading and learning at a very young age and my mother who has always been there for me through thick and thin. Thank you both for always being my guiding light. I am blessed to have recieved your love, your care and your time.

Chapter 1

Introduction

1.1 Importance of Lightweight and Secure Cryptosystems

The study of cryptography has progressed tremendously in the late 20th century. In the early days, cryptography mainly addressed the problem of secure and secret communications but in today's digital era, it is omnipresent. From message authentication, digital signatures, electronic elections, auctions and digital cash, it has found its way into almost every aspect of modern day digital applications.

Indeed, without cryptography, most of the devices available today like laptops, smartphones and PDA's would be severely crippled. Most of the security and privacy features we take for granted would be compromised. In the face of rapid advances in computing power and technology, and given the very real probability that practical quantum computing becomes a reality in the near future, older cryptographic schemes are increasingly likely to be broken and much stronger security is needed in order to keep up with the times. It makes sense then, that faster, better, more efficient and secure cryptographic schemes should be continually researched and developed. Also, with the mass scale advent of miniature computing devices like smartphones and PDA's, which albeit having limited resources are becoming part and parcel of our very existence, speed of encryption and decryption also assumes great significance. Most importantly, with the growing popularity of public social networks, a majority of which operate via these resource limited devices, the need for encrypting large amounts of data on a routine basis to protect individual privacy arises and this, more than ever, presents the need for solutions which are both secure as well as extremely

fast.

Public key cryptosystems were introduced by Diffe and Hellman [23], and since then, various different techniques have been successfully proposed to promote this brilliant concept. One of the most important uses of public key cryptosystems is to securely exchange private keys for single key cryptosystems between persons who may never have met before. This is because while public key cryptosystems solve the age old problem of having to physically meet and exchange keys (which is a necessity in case of single key encryption), they are quite slow in comparison to single key cryptosystems and hence cannot be used to encrypt large volumes of data. Also, while being faster than their public key counterparts, even existing single key schemes require further speedup as far as devices with limited resources are concerned. The objective of this thesis is to study solutions which are both secure as well as lightweight in nature.

1.2 What is Finite Automata?

While finite automata may have different meanings depending on the applications for which it is used, we shall look at it in the context of the cryptosystems in this thesis. For our purpose, finite automata refers to a finite sequential state machine which has an input and an output as well as an "internal state". It is similar to a digital system with finite "memory" [18]. When it recieves an input of finite length, it produces an output of the same length and the internal state changes according to predefined rules. The output at the current instant as well as the internal state at the next instant can be determined by the current input and internal state.

Whenever the input sequence can be retrieved by the output sequence (and the initial internal state), the system is said to be invertible and this is the property used for cryptosystems based on finite automata. In particular, a weakly invertible finite automaton with a delay τ is an automaton where any input can be determined by the state and the output given that the subsequent τ outputs are known.

Simply put, the input can be determined after τ delays or steps. While similar to usual injective functions in the sense that the input can be determined from the output, the dependence on the state information and the value of τ make it more complicated than normal functions.

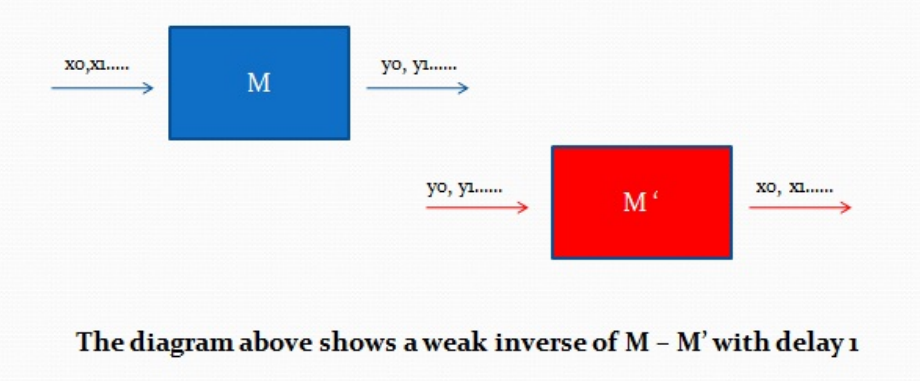


Figure 1.1: Finite automaton and its inverse

1.3 Finite Automata Based Cryptosystems

Most of the cryptosystems used today are based on number theoretic problems. Cryptosystems based on finite automata are a relatively new concept. These cryptosystems are lightweight in nature and can be implemented easily in hardware or software using simple logical operations, thus affording fast encryption and decryption as well as relatively small key sizes. Added to this is the advantage that they can be used to implement digital signatures as well as perform conventional encryption. The difficulty in inversion of finite automata and factoring matrix polynomials account for the security of these systems. Various public key cryptosystems based on finite automata have been proposed like FAPKC0, FAPKC1, FAPKC2, FAPKC93, FAPKC3 and FAPKC4 [21].

In symmetric or single key encryption, the secret key consists of two finite automata which are constructed so that their inverses are easily calculated. By applying the inverses of the automata to the cipher-text, the plain-text can be effectively calculated. In case of asymmetric or public key encryption, the public key consists of another automata which is the combination of the two finite automata while the private key consists of the inverse of the two individual automata. It is normally hard to invert the combined automata without knowledge of the private key automata.

The FAPKC3 and FAPKC4 systems are secure, simple and fast and can resist various attacks like the chosen plaintext attacks, the chosen ciphertext attacks and the exhaustive search attacks [22]. These factors account for the attractiveness of these cryptosystems and though development is still in its infancy, the results look quite promising.

1.4 Contributions of this Thesis

In this thesis, we thoroughly discuss the various finite automata based cryptosystems. We present a Java based implementation of both single and public key cryptosystems. We then propose and analyze a modified version of the finite automata based cryptosystem based on DES (Data Encryption Standard). Specifically, we make the following contributions;

- We present a thorough review of finite automata based cryptosystems and related cryptographic concepts.
- We propose a modified DES (Data Encryption Standard) based FA Cryptosystem.
- We implement a software library in Java and present the software design and implementation in detail. Implementations of the original single key and FAPKC3 [22] public key cryptosystems as well as our proposed DES-based finite automata cryptosystem are presented.
- We carry out prototype testing and analysis and show that the statistical properties measured on the ciphertext of our proposed DES-based finite automata cryptosystem are satisfactory. Our test results on the Nokia N900 Internet Tablet are also satisfactory.

Chapter 2

Background Knowledge

2.1 Symmetric and Asymmetric Cryptosystems

Cryptosystems can be divided into two major categories depending on the manner in which they are used. The first and the more traditional form is the symmetric cryptosystem [16] in which the sender and the receiver of a message both share the same secret key. This secret key had to be securely communicated between both sending and receiving parties before any encrypted communication could begin. Typically, in a symmetric cryptosystem, decryption is carried out using the same procedure as encryption but in reverse.

Symmetric ciphers may be further divided into block ciphers and stream ciphers. Block ciphers operate on blocks of data of some predetermined length. A block of data is encrypted at a time using the secret key. In contrast, stream ciphers operate on a stream of data. This stream of data is encrypted continuously without segregating it into different blocks.

Asymmetric or public key cryptography was first introduced in 1976 by Diffe and Hellman [23]. Prior to this, all secret communication was carried out using symmetric ciphers. In case of asymmetric ciphers, there are two keys instead of one. One is the public key which may be freely distributed to everyone. This may be made public at no security risk. Anyone can encrypt a message using a person's public key. Decryption of the message however, is achieved using a private key which is known only to the intended recipient of the message. It is not possible to derive the private key from the public key. The chief advantage of asymmetric encryption is that the sender and the receiver do not need to physically meet or otherwise establish any

secure communication before exchanging messages since a message can be encrypted using the publicly available public key.

2.2 The Principles of Confusion and Diffusion

The principles of Confusion and Diffusion are extremely important in the design of any good cryptosystem. Indeed they are taken so seriously that they may be called the guiding light of modern cryptosystem designers. These principles were first introduced by Claude Shannon in [15].

The chief objectives of confusion and diffusion [16] are to make cryptanalysis on any cryptosystem very difficult by statistical tools and analysis. In general, statistical tools rely on patterns like the frequency distributions of various letters in the plaintext in order to establish a relation between the plaintext and ciphertext by comparing both and trying to guess the secret key or the contents of the plaintext.

By diffusion, we mean that the ciphertext should be completely different from the plaintext so as to mask all statistical patterns between the plaintext and ciphertext. Changing even a small part of the plaintext should result in a ciphertext that is completely different and pseudorandom in nature. Even if multiple plaintext-ciphertext pairs are available, it should be very difficult to find any correlation between the two. In other words, a small change in the plaintext should result in a major, unpredictable change in the ciphertext.

Confusion on the other hand refers to the relationship between the key and the ciphertext. For any good cryptosystem, this relation should be as complicated as possible. It should be extremely difficult to find the key even if there were some statistical correlation between the plaintext and the ciphertext. In general, even a small change in the key should result in a completely different ciphertext. These principles are considered so central to the design of cryptosystems - specially block ciphers, that they have become the foundations upon which modern ciphers are designed [12].

2.3 Basic Concept of Finite Automata

By finite automata [18] as applicable to this thesis, we mean a finite state machine :

$$\mathrm{M} = <X, Y, S, \delta, \lambda>$$

where:
X - Set of all input alphabets
Y - Set of all output alphabets
S - Set of all states of the finite machine
δ - Transition function where $\delta : S$ x $X \rightarrow$ S
λ - Output function where: $\lambda : S$ x $X \rightarrow$ Y

The finite automata cryptosystems need h previous inputs and k previous encrypted outputs in order to encrypt a new byte of data. h and k are user defined variables.

2.4 Invertibility of Finite Automata

If M' is an automaton through which we can find the input x_0, given the output $\lambda(s, x_0.....x_\tau)$ of M then M' is said to be the weak inverse of M with delay τ where τ is any nonnegative integer.

```
sarshad@Sarsh:~/Documents/Programming/Java/JavaWorkspace/FACryptography/singleKey-fixedparams$ java FAPKC3
Generating a Random linear Finite Automata M0:

y(i) =

01110001              00000111              10000000              01011101              01000000
01100111              00011110              10000000              00110000              10010111
00001011              00001100              00000000              00011000              11010010
11011110 y(i-1) +     00110001 y(i-2) +     10000000 x(i) +       01011001 x(i-1);   P = 01011100
00001010              01100001              10000000              00111110              01111100
01010100              01100111              10000000              01000111              10010000
11101001              10111101              10000000              00010100              10100100
00110100              01010011              00000000              00010111              01101001



Now Generating the Inverse of linear Finite Automata M0 called M0*:

x(i) =

00110000           00000000           01000000           01100111           00011110
00000000           10010111           00100110           10111111           00000000
00000000           11010010           00100001           10010101           00000000
00000000 x(i - 1) + 01011100 y(i) +   11100111 y(i-1) +   00101001 y(i-2) +   00000000 y(i - 3)
00000000           01111100           11101100           00100101           00000000
00000000           10010000           10101111           00110110           00000000
00000000           10100100           00101110           01101100           00000000
00000000           01101001           01010010           00100000           00000000
```

Figure 2.1: Generated linear FA and its inverse using a java implementation

Let $M' = <X, Y, S', \delta', \lambda'>$. M is said to be weakly invertible [18] with delay τ if for any $x_i \in X$, $i = 0, 1, 2, \ldots$ τ and $s \in S$, x_0 can be uniquely determined by the state S and the output $\lambda(s, x_0.....x_\tau)$.

For any state $s \in S$ and $s' \in S'$, if:

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

where 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, then (s', s) is a matching pair with delay τ. In other words, s' matches s with delay τ.

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 [9]. Figure 2.1 shows an example of a FA and its inverse.

2.5 (h, k) Order Memory Finite Automata

Let ϕ be a mapping from $Y_k \times X_{h+1}$ to Y. This mapping defines a finite automata

$$M = <X, Y, (Y_k \times X_h), \delta, \lambda>$$

where we have:

$y(i) = \phi(y_{i-1}, \ldots y_{i-k}, x_i, \ldots x_{i-h})$, $i = 0, 1, \ldots$

$\delta(<y_{-1}, \ldots, y_{-k}, x_{-1}, \ldots, x_{-h}>, x_0) = <y_0, \ldots, y_{-k+1}, x_0, \ldots, x_{-h+1}>$

$\lambda(<y_{-1}, \ldots, y_{-k}, x_{-1}, \ldots, x_{-h}>, x_0) = y_0$

$y_0 = \phi(y_{-1}, \ldots, y_{-k}, x_0, x_{-1}, \ldots, x_{-h})$

M is said to be an (h, k) order memory finite automata [18] denoted by M_ϕ. What this means is that M needs k previous outputs and h previous inputs to generate the current output. If the mapping is from X_{h+1} to Y, it is said to be and h-order input memory finite automaton.

2.6 Linear and Non-Linear Finite Automata

A finite automaton may 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 finite automaton is greater than one [18]. In Figure 2.1, we present an example of a linear finite automaton and its inverse. Figure 2.2 shows an example of a non-linear finite automaton and its inverse. For all the illustrations in this thesis, we shall refer to a linear finite automaton as M_0 and a non-linear finite automaton as M_1. Their inverses will be referred to as M_0' and M_1' respectively.

```
Now Generating h-order input memory non-linear Finite Automata M1:

y(i) =

00000000             10000000             01001011
00000000             00000000             00100000
00000000             10011111             10100000
00000000 x(i) +      11000000 x(i - 1) +  01110101 x( i - 2) +
00000000             10011111             10100011
00000000             10011111             10010011
00000000             10000000             01110011
00000000             00011111             11010101

00000000                      01111111
00000000                      00000000
00000000                      01111111
00000000 s(x(i),x(i - 1)) +   11001011 s(x(i - 1),x(i - 2))
00000000                      01111111
00000000                      01111111
00000000                      01111111
00000000                      00000000

Where function 's' means the componentwise multiplication.


Now Generating the inverse of non-linear Finite Automata M1 called M1*:

x(i) =

01000000                   00000000                   01111111                   00000000
00101011                   00000000                   10110100                   00000000
00000000                   00000000                   00000000                   01000000
00000000 x(i - 1) +        00000000 x(i - 2) +        00000000 t(x(i),x(i - 1)) + 01001100 y(i) +
00000000                   00000000                   00000000                   10001110
11100000                   00000000                   00000000                   11101010
11100000                   00000000                   00000000                   11001001
11100000                   00000000                   00000000                   11100001

00100110            00000000
11010111            00000000
00000000            00000000
00000000 y(i - 1) + 00000000 y(i - 2)
00000000            00000000
00000110            00000000
00000110            00000000
00000110            00000000

Where function 't' means the componentwise multiplication.
```

Figure 2.2: Generated non-linear FA and its inverse using a java implementation

2.7 Combination of Finite Automata

By combination of finite automata, we mean the combination of two different finite automata with the output of the first being the input of the second, such that the result using the combined automata is equivalent to using the two consituent automata seperately one after the other. We will consider two different combinations of finite automata - $C(M_1, M_2)$ and $C'(M_f, M_g)$.

For any two finite automata, $M_i = <X_i, Y_i, S_i, \delta_i, \lambda_i>$, i = 1,2; where $Y_1 = X_2$ (i.e. the output of the first automata is the input of the second) we denote $C(M_1, M_2)$ as:

$$C(M_1, M_2) = <X_1, Y_2, S_1 \times S_2, \delta, \lambda>$$

where:

$$\delta(<s_1, s_2>, x) = <\delta_1(s_1, x), \delta_2(s_2, \lambda_1(s_1, x))>$$
$$\lambda(<s_1, s_2>, x) = \lambda_2(s_2, \lambda_1(s_1, x))$$
$$s_1 \in S_1; s_2 \in S_2; x \in X_1$$

In the second case, let g be a (p, r)-order memory FA: $U_r \times V_{p+1} \rightarrow U$ and f be a t-order input memory FA: $W_{t+1} \rightarrow V$

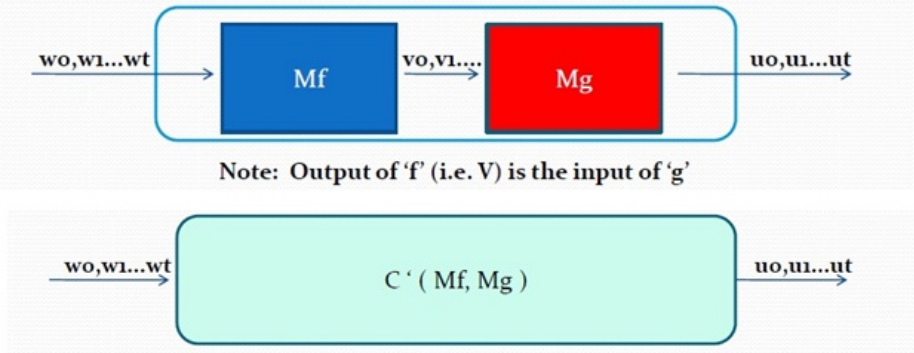


Figure 2.3: Illustration of combined automata $C'(M_f, M_g)$

The combined automata [9] $C'(M_f, M_g)$ is a $(p + t, r)$ order memory FA where:

$$w_i = g(w_{i-1} \ldots w_{i-r}, f(w_i, \ldots w_{i-t}), f(w_{i-p}, \ldots w_{i-p-t}))$$
$$= g'(w_{i-1} \ldots w_{i-r}, w_i, \ldots w_{i-p-t})$$
$$i = 0, 1, \ldots$$

We shall use this variant of combined automata for the cryptosystems in this thesis. Please note that the notation $C'(M_f, M_g)$ should not be confused with the inverse of the combined automata. It is simply the second type of combined automata as explained above.

The output of the combined automata is equivalent to the outputs of the two automata M_f and M_g executed independently in succession. Figure 2.4 shows the generation of combined automata using the linear and non-linear automata illustrated

in Figure 2.1 and Figure 2.2 using the Java implementation created as part of this thesis.

```
Now Generating the Combined Automata (M1, M0):

y(i) =

01110001            00000111            00000000            10000000            10010100            10110000
01100111            00011110            00000000            10000000            00010100            11010101
00001011            00001100            00000000            00000000            01011111            11010110
11011110 y(i-1) +   00110001 y(i-2) +   00000000 x(i) +     10000000 x(i - 1) + 00001011 x(i - 2) + 00100011 x(i-3) +
00001010            01100001            00000000            10000000            10010100            10010110
01010100            01100111            00000000            10000000            01001011            00010101
11101001            10111101            00000000            10000000            00010100            11100110
00110100            01010011            00000000            00000000            11000000            01000000

00000000                    01111111                    11001011
00000000                    01111111                    10110100
00000000                    00000000                    10110100
00000000 t(x(i),x(i-1)) +   01111111 t(x(i-1),x(i-2) + 10110100 t(x(i-2),x(i-3))
00000000                    01111111                    11001011
00000000                    01111111                    00000000
00000000                    01111111                    10110100
00000000                    00000000                    11001011

Where function 't' means the componentwise multiplication.
```

Figure 2.4: Generation of combined automata using java implementation

2.8 R_a Transformation

The R_a transformation is one of the basic transformations used for the finite automaton cryptosystems. It is defined as follows.

Let $M = <X, Y, (Y_t \times X_r), \delta, \lambda>$ be a (r, t) order memory FA over $GF(q)$ defined by $y_i = f(y_{i-1}, \ldots y_{i-t}, x_i, \ldots, x_{i-r})$, $i = 0, 1, \ldots$; where X and Y are column vector spaces over $GF(q)$. x_i and y_i are column vectors of dimensions l and m respectively.

For the purpose of our discussion and implementation, we will limit ourselves to the case of $GF(2)$. Also x_i and y_i are column vectors of dimension 8 obtained from the multiplication of (8×8) coefficient matrices and column vectors $x'_i \in X$ and $y'_i \in Y$ of dimension 8, respectively.

Let $eq_k(i)$ be an equation in the form of:

$eq_k(i) : f_k(x_i, \ldots, x_{i-r}, y_{i+k}, \ldots, y_{i-t}) = 0$

Let φ_k be a transformation on $eq_k(i)$ in the form:

$eq'_k(i) : f'_k(x_i, \ldots, x_{i-r}, y_{i+k}, ..., y_{i-t}) = 0$

If $eq_k(i)$ and $eq'_k(i)$ are equivalent, then $eq'_k(i)$ is said to be obtained from $eq_k(i)$ by Rule R_a [18] using φ_k, denoted by:

$eq_k(i) \overset{Ra_\phi(k)}{\rightarrow} eq'_k(i)$

In order to make the concept clearer, consider the following equation $eq_k(i)$ using 3x3 matrix coefficients:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

If we multiply $eq_k(i)$ by a matrix:

$$P = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

We get the equation $eq'_k(i)$ which is equivalent to $eq_k(i)$

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

In this case, if we multiply $eq'_k(i)$ with the inverse of P, we will end up with the original equation $eq_k(i)$. This is one of the important properties used for cryptographic purposes.

2.9 R_b Transformation

Proceeding further, assume that $eq'_k(i)$ is of the form:

$f'_k(x_i, ..., x_{i-r}, y_{i+k}, ..., y_{i-t}) = 0$

and that the last $m - rk + 1$ components of the left side of $eq'_k(i)$ don't depend on x_i. Now let $eq_{k+1}(i)$ be:

$$\begin{pmatrix} I'_k f'_k(x(i), ...x(i-r), y(i+k), ...y(i-t)) \\ I''_k f'_k(x(i+1), ..., x(i+1-r), y(i+1+k), ..., y(i+1-t)) \end{pmatrix} = 0$$

where I'_k and I''_k are submatrices of the first r_{k+1} and of the last $m - r_{k+1}$ rows of the $(m \times m)$ identity matrix respectively.

Now $eq_{k+1}(i)$ is said to be obtained from $eq'_k(i)$ by rule R_b [18].

$eq'_k(i) \overset{R_b(r_{k+1})}{\rightarrow} eq_{k+1}(i)$

Now let us clarify this transform with an example.

Consider our previous equation $eq'_k(i)$

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

An R_b Transform of height 1 or $R_b(1)$ would yield:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

```
sarshad@Sarsh:~/Documents/Programming/Java/JavaWorkspace/FACryptography/singleKey-fixedparams$ java FAPKC3 m0 debug
Initial X0 and X1:
10000000
01000000
00100000
00010000
00001000
00000100
00000010
00000001

11000011
00000000
00000000
00000000
00000000
00000000
00000000
00000000

Values of X0 and X1 after initial Rb Transform:
10000000        11000011
00000000        01000000
00000000        00100000
00000000        00010000
00000000        00001000
00000000        00000100
00000000        00000010
00000000        00000001

Values of X0 and X1 after Ra Transform:
10000000        10011011
00000000        01110101
10000000        10110100
00000000        01010100
10000000        11011010
00000000        00011111
10000000        10001001
00000000        00000101
```

Figure 2.5: R_a and R_b transformations in java implementation

If reverse R_a transform is applied to the equation above, followed by an $R_b(1)$ transform in the opposite direction, we get the original equation. Repeated application of this property forms the basis of finite automata cryptography.

What is significant here is that the transformations:

$eq_k(i) \overset{R_a(\phi_j)}{\rightarrow} eq'_k(i) \overset{R_b(r_{j+1})}{\rightarrow} eq_{k+1}(i)$;j = 0,1,.....k

can be completely reversed by using the reverse sequence:
$eq_{k+1}(i) \overset{R_b(r'_{j+1})}{\rightarrow} eq'_j(i) \overset{R_a(\phi'_j)}{\rightarrow} eq_j(i)$

Where ϕ'_j denotes the inverse of the matrix used for the original Ra transform and r'_{j+1} denotes reverse R_b transform in the opposite direction [9]. Figure 2.5 shows an example of R_a and R_b transformations.

2.10 Symmetric Single Key Cryptosystem

For a simple single key cryptosystem [9], we first compute an (h, k)-order memory FA with delay τ . Assuming $h = 1; k = 2$; τ =1; a suitable $(1, 2)$-order FA could be:

$M_0 \rightarrow y(i) =$

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

Then compute the inverse of M_0 - which is a $(\tau + k, h)$-order memory FA, say M'_0. Given the above parameters, a suitable $(3, 1)$-order FA is:

$M'_0 \rightarrow x(i) =$

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

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \text{y(i - 1)} + \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \text{y(i - 2)} + \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{y(i - 3)}$$

Note that in these examples, (3×3) matrix coefficients are considered. However, for our implementation we use (8×8) matrices with the vectors $x(i)$ and $y(i)$ with dimension 8 representing 8 bits as shown in the various illustrations of our Java implementation.

Now we choose random vectors $y(i-1), y(i-2)$ and $x(i-1)$ to define the starting state of M0

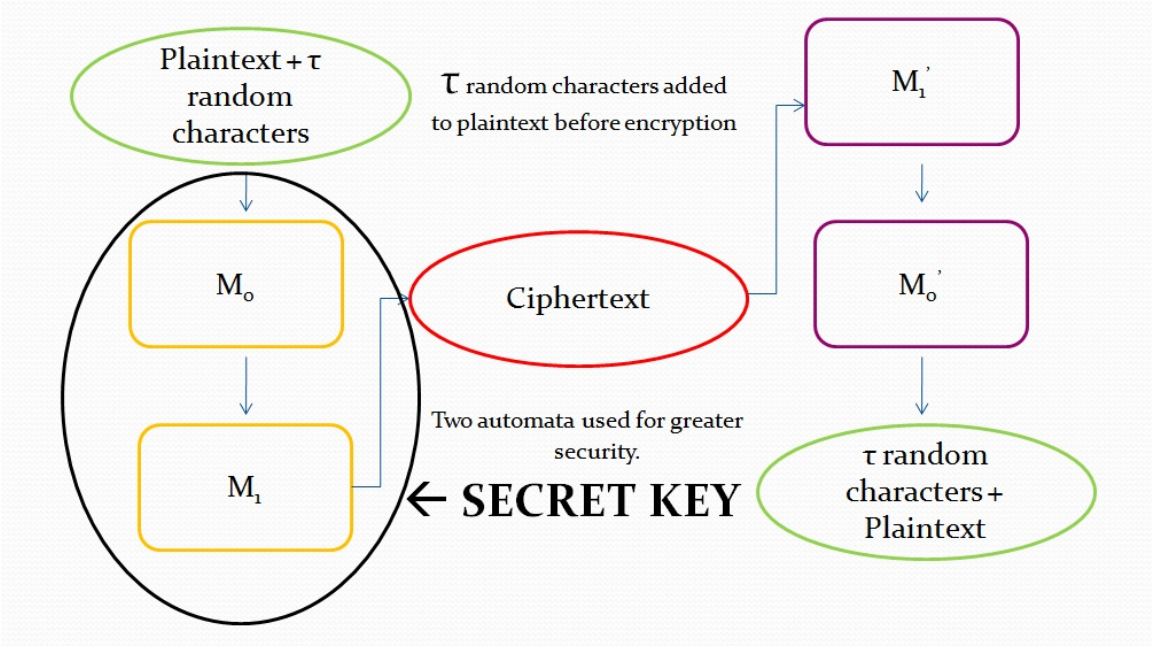


Figure 2.6: Illustration of symmetric single Key Encryption

Using these, we encrypt the plaintext $(x_0, x_1, \ldots x_n)$. Note that since it is a τ memory FA, τ random characters need to be added to the end of the original plaintext before encryption.

Our secret key will consist of: (1) The automaton M_0 and (2) The random vectors $y(i-1), y(i-2)$ and $x(i-1)$. Usually, in order to ensure greater security, this process is repeated twice using a linear automata M_0 and a non-linear automata M_1. That is to say, the plaintext is encrypted first using M_0 and then the resultant ciphertext is encrypted again using M_1. In this case, in order to retrieve the plaintext, the inverse of M_1 has to be first applied to the resultant ciphertext followed by the inverse of M_0. This sequence is shown figure 2.6. The secret key in this case consists of both the automata M_0 and M_1.

As will be shown in the next section, the format of single key encryption using two automata is the basis of the public key encryption. The only difference between the two lies in the fact that for public key encryption, the combined automata is used instead of the two individual automata as illustrated in the next section.

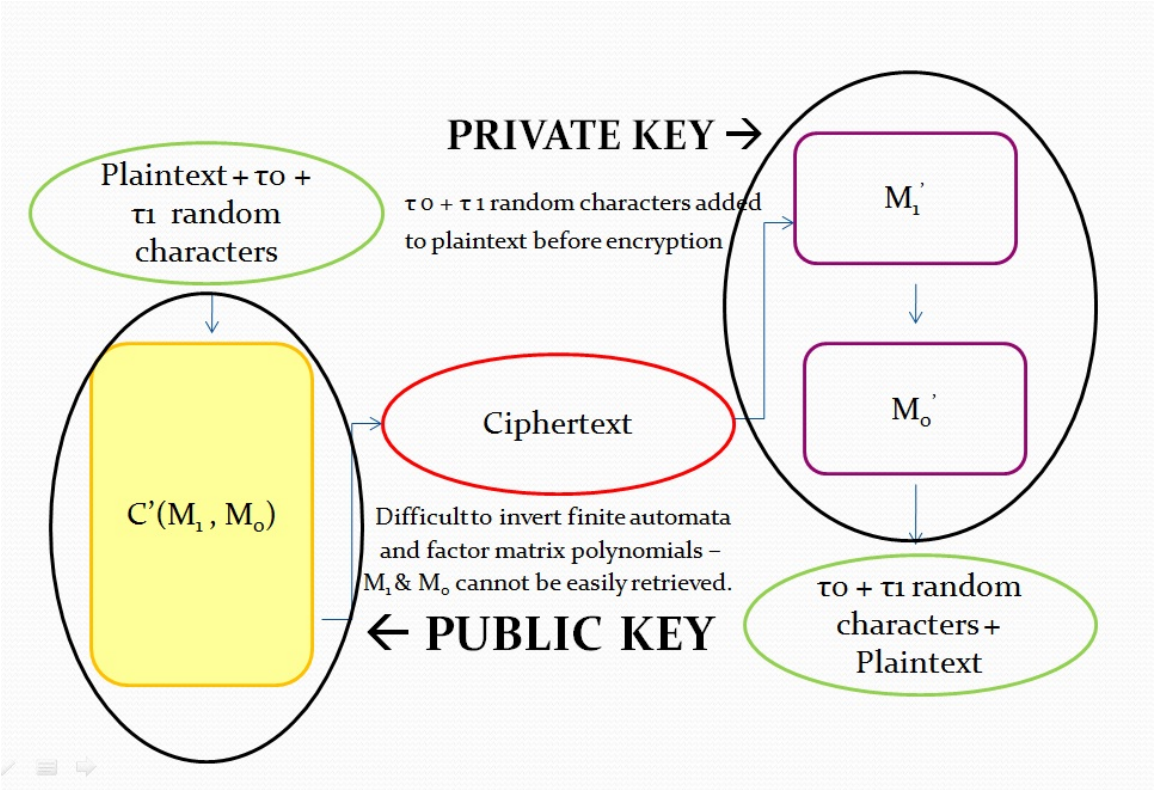


Figure 2.7: Illustration of asymmetric public key Encryption

2.11 Asymmetric Public Key Cryptosystem

As is evident from the diagram in figure 2.7, the major difference between the earlier single key encryption using two finite automata M_0 and M_1 and the public key cryptosystem [9] is that the combined automata $C'(M1, M0)$ is used instead of the individual automata Please note that C' simply refers to the second type of combined automata as explained in Section 2.7 and should not be confused with the inverse of the combined automata. The private key consists of the inverses of the two components of the combined automata. It is generally hard to invert this combined automata without knowledge of the private key automata. However, using the inverses of the private key automata, we can easily calculate the inverse.

In this case, $\tau_0 + \tau_1$ random characters are added to the end of the plaintext before encryption.

We shall consider the implementations of both symmetric and public key cryptosystems based on finite automata as well as a cryptosystem using finite automata

but based on the design of the Data Encryption Standard (DES) in this thesis.

2.12 A Brief Introduction to the Data Encryption Standard

The Data Encryption Standard (DES) [7] is a widely used cryptosystem. It was established as a standard by the National Institute of Standards and Technology (NIST) in 1977. It continued to enjoy widespread acceptance till it was replaced by the Advanced Encryption Standard (AES) [7] which has a more secure algorithm. However, it is still used in practise today. It was developed by IBM and was initially known as the LUCIFER [3] cipher.

DES is a symmetric block cipher which works on 64-bit blocks of data using a 56-bit key. It encrypts a 64-bit block of text and outputs a 64-bit block of ciphertext using a concept known as a feistel network. In a feistel network, each block of plaintext is divided into left and right halves denoted as L_n and R_n. There are sixteen rounds in encryption and for each of these rounds, $L_n = R_{n-1}$ and $R_n = L_{n-1} \oplus F(R_{n-1}, K_i)$. This is was first proposed by Horst Feistel who was a researcher at IBM. The function F involves both permutation and substitution of the 32-bit input that is processed through it in each of the sixteen rounds.

Each of the rounds use sixteen different 48-bit subkeys K_i which are generated from the main 64-bit key using a very complex key generation algorithm. Though the key length is 64-bits for DES, it is processed using a permutation table to derive the 56-bit key as mentioned earlier. This key is then split into two halves and are subjected to a sequence of left shifts or rotations, based on a predetermined shift table. Finally, a second permutation table is used to derive each of the final 48-bit keys. In our DES-based FA cryptosystem, we have modified this key generation algorithm for use with the traditional finite automata cryptosystems as described in Chapter 3. The operation of the key generation algorithm is very similar to that of the original DES algorithm.

Chapter 3

A DES-Based Finite Automaton Cryptosystem

3.1 Features

In this chapter, we present a new version of the traditional finite automata cryptosystems. The key generation algorithm is based on the popular and widely used Data Encryption Standard (DES) [10]. The main features of the proposed algorithm include:

- It uses a 128-bit key. Unlike the traditional finite automata 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 and is commonly regarded as sufficient for most applications.
- A new parameter, μ, is introduced to determine how many linear/non-linear automata pairs are to be generated and used for encryption/decryption purposes.
- Though this is a stream cipher, the plaintext is split up into 64-bit blocks. Each block is encrypted by a linear and non-linear automata pair in succession. This

is equivalent to encrypting each block with the combined automata comprised of the linear and non linear auomata. Further, there are μ different linear/non-linear automata pairs and these are alternately cycled by the algorithm for each new block.

- Since μ can have any value between 1 and 7, cryptanalysis on the resultant cipher is difficult since firstly, each automata 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. Also, μ different pairs of automata are used for different blocks as explained previously.

- Though key generation times are greater (depending on μ) than those for the traditional FA cryptosystems, the essential speed for encryption and decryption remains the same as that using the standard public key finite automata cryptosystem. The security however is vastly increased due to the increase in the number of automata used and the introduction of extra randomness due to the random characters appended in each block.

3.2 Permutation Tables

The DES based finite automata cryptosystem described in this chapter uses various permutation tables for its operation, similar to the original DES cryptosystem. The permutation tables are randomly chosen. However, care has been taken wherever possible to ensure that the permuted output is evenly spread across the entire input. 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.

3.3 Specifications

The DES-based finite automata cryptosystem is a symmetric stream cipher. Despite being a stream cipher, it operates on 64-bit plaintext blocks. The three main components which make up this cryptosystem include:

- Key processing
- Automata and starting state generation
- Encryption and decryption

3.3.1 Key Processing

As mentioned previously, the DES-based finite automata cryptosystem is based on a 128-bit key. This key is processed using a key generation algorithm similar to DES. This algorithm creates sixteen subkeys, each of which are 96 bits in length. These subkeys 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 according to the PC-1 permutation table. This table permutes the key to a 112-bit key. Every 8^{th} bit is discarded for the first 64 bits (similar to DES) [3] and then bit numbers 68, 76, 82, 96, 103, 109, 117 and 121 are discarded. As we can see, the first entry in the table is 57. This means that the 57^{th} bit of the original 128-bit key now becomes the first bit of the permuted key. The 49^{th} bit of the original key becomes the second bit of the permuted key and so on till the 91^{st} bit of the original key becomes the 112^{th} bit of the permuted key.

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

Figure 3.1: The PC-1 permutation table

- 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 sixteen blocks L_n and R_n for n=1, 2, 3, ... 16. A schedule of left shifts of the previous blocks is used to derive each of these 16 pairs of blocks using the shift table SH-1. By left shift, we mean that we move each bit one place to the left. The first bit, however is cycled to the end of the block. So, as we can see from the shift table, the first shift is 2 places to the left, the second shift is 4 places to the left and so on. More specifically, L_1 and R_1 are obtained by shifting L_0 and R_0 2 places to the left, L_2 and R_2 are obtained by shifting L_1 and R_1 4 places to the left and so on till we get 16 pairs of subkeys each 56 bits long.

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

Figure 3.2: The SH-1 shift table

- Step 3: We now concatenate the L_n and R_n pairs to form 16 subkeys which are each 112 bits long. This 112-bit key is now permuted according to the table PC-2. This table permutes each key to a 96 bit key. The bits 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 16, 96 bit keys generated in a fashion similar to that in the DES cipher.

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

Figure 3.3: The PC-2 permutation table

3.3.2 Automata and Starting State Generation

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

- First we need to generate μ pairs of linear and non-linear finite automata for the cryptosystem. These finite automata will be derived completely from the generated subkeys described above and will have no random element in them. This ensures that encryption and decryption are completely based on the key. The linear automata will have the variable parameters h_0, k_0 and τ_0 and the non-linear automata will have the variable parameters h_1 and τ_1.

- For the linear automata, we need to generate $h_0 + k_0$ matrices as the component matrices for generating the finite automata. We also need to generate τ_0 full rank matrices. The specifics of how this can be generated using the subkeys will be discussed next. 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 $M-3$ in order to derive 64 random bits from the 96-bit keys. These three permutation tables are used in sequence in a cyclical manner. Thirty two random bits are discarded by the M tables to generate a 64-bit 8x8 matrix. Using this process we create

the $h_0 + k_0$ component matrices.

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

Figure 3.4: The M-1 permutation table

In order to create the τ_0 full rank matrices, a slightly different approach is adopted. The τ_0 matrices are generated as normal using the same method as for the first $h_0 + k_0$ matrices. However, it is unlikely that these will be full rank, given the pseudorandom nature of the key generation process. In order to overcome this hurdle we proceed 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 automata matrices) or upper triangular (for non linear automata matrices) by setting all values in the diagonal row 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. Let us 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

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

Figure 3.5: The M-2 permutation table

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. Thus using this process, we can create random but predictable full rank matrices for use in construction of the finite automata.

- For the non linear automata, we need $h_1 + 1$ component matrices. These are generated as before 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 $M-1, M-2$ 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 automata. These component matrices, once derived, are used to create the non-linear finite automata as normal.

- After generation of each linear or non-linear automata, we derive the starting states for that particular automata before proceeding to generate the next one. The starting states are derived from alternate subkeys immediately following the last key that was used to generate a particular finite automata. For instance,

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

Figure 3.6: The M-3 permutation table

if the first linear automata was generated using subkeys K_1, K_3 and K_5 (say) then the three 8 bit vectors that will be required as the starting states of this automata are generated from the sequential keys K_7, K_9 and K_{11}. This is done using the look up tables $SS-1, SS-2$ and $SS-3$. These tables are used alternately in order to provide confusion as to the selection of the 8 bits from the 96-bit subkeys. 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 non linear automata.

The automata and starting state creation process is purposely complicated in order 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 automata and starting state generation process takes very little time even for large values of $h_0, k_0, \tau_0, h_1, \tau_1$ and μ.

3.3.3 Encryption and Decryption

The main difference in encryption/decryption using our DES based finite automata cryptosystem, compared to the original FA cryptosystem, lies in the fact that all the component matrices, which in the original finite automata based cryptosystem were

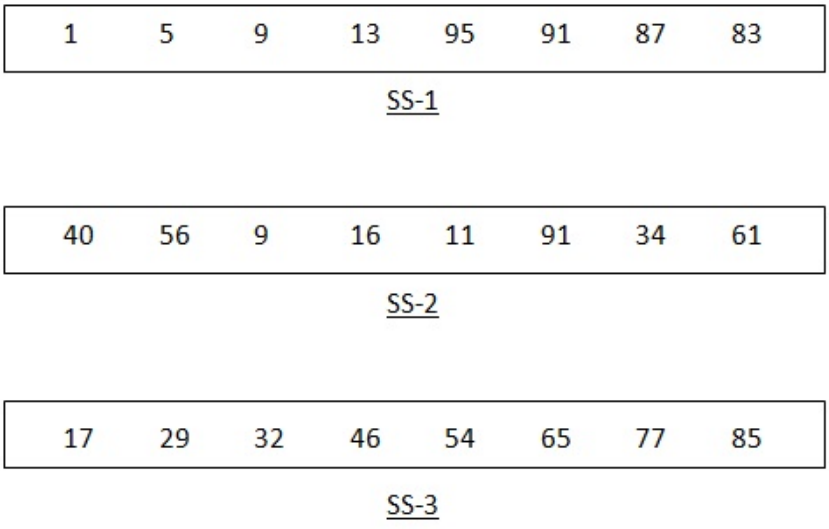


Figure 3.7: The starting state permutation tables

chosen at random, are now chosen based on the 16 subkeys that we have generated during the key generation phase. This process has been detailed above. Figure 3.8 shows the high level block diagram of our cryptosystem.

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

For example if μ has a value of 2, then two linear/non-linear automata pairs are generated by the algorithm. Let us denote these by L_1, NL_1, L_2 and NL_2. Let us suppose that the plaintext is split up into eight 64-bit blocks $B_1, B_2, \ldots, B_8$. The ciphertext c will be generated as follows:

$$c := Enc(B_1^{L1,NL1}, B_2^{L2,NL2}, B_3^{L1,NL1}, B_4^{L2,NL2}, B_5^{L1,NL1}, B_6^{L2,NL2}, B_7^{L1,NL1}, B_8^{L2,NL2})$$

where Enc denotes the encryption algorithm, B_i denotes the plaintext blocks, L_i denotes the linear automata and NL_i denotes the non-linear automata.

Decryption is carried out in the reverse order. That is, let us assume that the ciphertext is split up into eight 64-bit blocks $C_1, C_2, ..., C_8$. The plaintext p will be

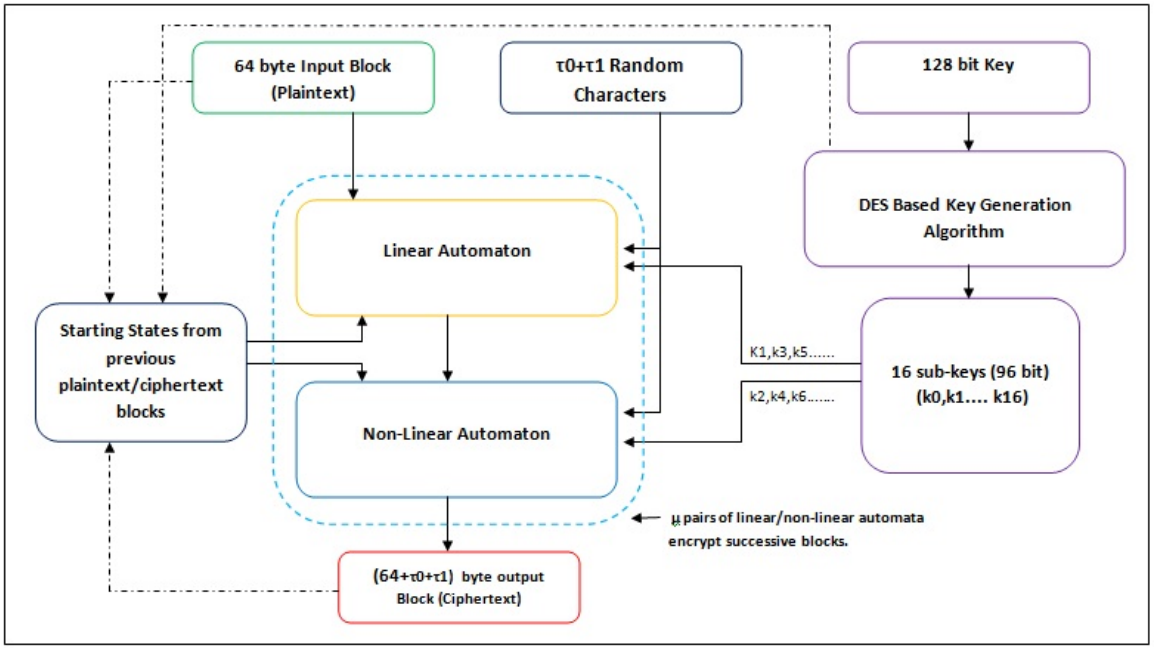


Figure 3.8: High level block diagram of the DES based FA cryptosystem

generated as follows:

$$p := Dec(C_1^{NL1,L1}, C_2^{NL2,L2}, C_3^{NL1,L1}, C_4^{NL2,L2}, C_5^{NL1,L1}, C_6^{NL2,L2}, C_7^{NL1,L1}, C_8^{NL2,L2})$$

where Dec denotes the decryption algorithm, C_i denotes the ciphertext blocks, L_i denotes the linear automata and NL_i denotes the non-linear automata.

Note that besides the use of alternating linear and non linear automata pairs, each block also uses the last n bytes of ciphertext in the previous blocks as the starting states where the value of n depends on the choice of h_0, k_0, τ_0, h_1 and τ_1 for the linear and non-linear automata. This highly complex arrangement ensures that the ciphertext follows the required principles of confusion and diffusion quite efficiently. If a single bit of the plaintext is altered, the ciphertext undergoes a drastic change. Also there is very little correlation between patterns in the plaintext and the ciphertext.

The linear and non-linear automata can use any combination of $h_0, k_0, \tau_0, h_1, \tau_1$ and μ subject to the restrictions mentioned in the chapter on the finite automata cryptosystem software design. The configuration is contained in a config file. Besides the 128-bit key, these parameters also need to be known by both sender and receiver before any meaningful encryption or decryption can be carried out.

Chapter 4

Finite Automata Cryptosystem Software Design and Implementation

4.1 Introduction and Overview

In this section, we discuss the general software design and implementation of a single key cryptosystem as well as a public key cryptosystem based on finite automata, which to the best of our knowledge has no publicly available implementation so far. Having no prior precedent, programming, debugging and optimizing this application has been a non trivial and rewarding task. Though introduced in the early 80's, such cryptosystems did not find much mass appeal because they were primarily published in Chinese. Various versions of these cryptosystems like FAPKC0 in [19], FAPKC1 and FAPKC2 in [20] and FAPKC3 in [22] among others have been proposed so far. Our implementation is based on FAPKC3 as discussed in [9].

We further proceed to describe our novel DES based finite automata cryptosystem. This is a symmetric cryptosystem which uses a modified version of the well respected key generation algorithm of the Data Encryption Standard (DES) to generate linear as well as non-linear finite automata which are then used for the actual encryption as detailed in Chapter 3. Our design makes use of μ pairs of linear/non-linear automata for encryption, which produces the effect of encrypting with 2μ differenct keys using two types of automata simultaneously, vastly improving the security of the traditional cryptosystem while preserving its speed and efficiency.

Based largely on inexpensive bit operations, these cryptosystems are generally efficient in software despite being stream ciphers. For this reason, they are suitable for use in energy constrained and resource limited mobile devices like smartphones and PDA's. Both the single key cryptosystem as well as the public key cryptosystem detailed in this section have been tested on the Nokia N900 smartphone with satisfactory results, even though they are implemented in Java which is generally considered much slower than languages like C and C++. As with any stream cipher, however, the best results can only be achieved if implemented in hardware since operation of the cipher is bit by bit as opposed to block ciphers which generally encrypt blocks of data at a time.

The single key cryptosystem consists of a linear (or non linear) finite automata which is constructed out of (8×8) matrices over $GF(2)$. The strength and complexity of the automata so constructed depend on three parameters h, k and τ. Intutively speaking, h refers to the number of previous plaintext bytes fed into the automata for encryption along with the current plaintext byte, k refers to the number of previous ciphertext bytes fed into the automata, and τ refers to the number of random matrices based on cryptographically secure pseudo random number generators and the R_a/R_b transformations detailed in Chapter 2. Each plaintext byte is fed into the finite automata so created in sequence. As is evident from the very nature of its construction, given the fact that each byte depends on h previous plaintext bytes and k previous ciphertext bytes respectively, this results in a very strong cryptosystem with respect to diffusion and confusion [15] which is a necessary part of any good cipher. For a single key cryptosystem, the secret key consists of the automata so created and the initial $(h + k)$ bytes used as the starting state of encryption.

The public key cryptosystem is comprised of both a linear and non linear automata. While the linear automata is identical to that used for single key encryption, the non-linear automata is a h-order input memory automata. The ciphertext output of the linear automata is fed into the non-linear one and the output from the non-linear automata depends on the h previous bytes of ciphertext generated by the linear automata. However, in order for a public key cryptosystem to work, we need a public key which clearly cannot be the two individual automata discussed so far since then, their inverses could be trivially calculated. For this purpose, a special combined automata is constructed out of the two previously mentioned automata and this is used as the public key. The private key in this case consists of the two component automata used to construct the combined automata, along with the starting states.

It is hard to invert the combined automata in order to retrieve the two component automata and the security of the public key cryptosystem rests on this fact.

The DES-based finite automata cryptosystem is a symmetric cryptosystem which has been designed to achieve the security provided by encrypting with several pairs of linear and non-linear automata in succession, while at the same time retaining the speed and efficiency of the traditional public key cryptosystem. While its cryptographic operation is similar to that used in the public key cryptosystem, the finite automata are generated using a DES-based key generation algorithm.

4.2 Purpose

The purpose of this thesis is to create a fully working implementation of both a single key cryptosystem and a public key cryptosystem based on [9]. We further proceed to create a fully functional implementation of our proposed DES based symmetric key finite automata cryptosystem. Care is taken to ensure that the systems adhere satisfactorily to the principles of confusion and diffusion [16].

During the construction of the finite automata, certain random matrices are used. The implementation ensures that these random matrices are comprised of a cryptographically secure pseudo random number generator which is provided by the java "Secure-Random" class [4] using the SHA1 algorithm [13]. The prototype produces satisfactory results even on resource limited smartphones and PDA's. In our case we have tested it on the Nokia N900 Internet tablet. Both single key and public key cryptosystems, as well as the DES-based cryptosystem so implemented are capable of efficiently and accurately encrypting and decrypting any kind of data.

4.3 Compliance with System Design

The software design for all the finite automata cryptosystems presented in this thesis are based on the specifications as detailed in [9] with the following modifications:

- All parameters for both linear and non-linear automata (viz. h, k and τ) are completely variable. Certain restrictions imposed on the choice of these parameters to ensure security and accuracy of the algorithm are that the value of the h parameter shoud be at least 1, the value of k should be at least 0, and the value of τ should never exceed the value of h and in any case should never exceed 7.

This requirement is because of the nature of R_b transform. Since there are only 8 rows in each of the component matrices, R_b transform cannot be performed for values of τ greater than 7. A cyclical mechanism could be implemented where a value greater than 7 rolls back to 1 but this has no cryptographic value and hence is not incorporated.

- In case of our DES-based finite automata cryptosystem, the limitations on values of paramaters remain the same as in the traditional cryptosystems. In addition, the choice of the new introduced variable μ should be between 1 and 7, to balance the security strength and running speeds. A value beyond 7 would result in the cryptosystem being unacceptably slow and is in any case not required as is demonstrated by our tests in Chapter 5.

- The java "Secure-Random" class introduced in [4] is used with the SHA1 algorithm to generate pseudo random bits for all the random matrices. Though this causes a penalty in terms of speed, the very nature of a secure algorithm demands a cryptographically strong pseudo random number generator in order to maintain maximum security.

- The core Matrix class has been specially created with emphasis on the unique requirements of the cryptosystem. Existing matrix classes have not been used since they are not optimized for data over $GF(2)$ and hence suffer penalty in terms of speed.

- A simple graphical user interface has been implemented for the sole purpose of demonstration on the Nokia N900 Internet tablet.

- The single key cryptosystem can be used with either the linear or the non-linear finite automata. Further, due to the completely variable nature of the parameters, the linear automata can also be used as an h-order input memory finite automata.

- Checks and safeguards have been built into the program in order to ensure maximum efficiency, security and accuracy.

4.4 Software Architecture and Design - Core Components

We now proceed to study the software architecture and design of the single and public key automata based on FAOKC as well as our DES-based finite automata cryptosystem. All three cryptosystems have been implemented in Java. While differing significantly in their operation, all the three implemented variations of the finite automata cryptosystems have a common engine for the creation of automata, handling matrix operations and performing the R_a/R_b transformations. These common components will be detailed first and then we move on to the full explanation of the various specific components of the three implemented cryptosystems. The implementation is in Java 2 Standard Edition and has been tested on an Intel Centrino Core 2 Duo computer running at 2.16 Ghz with 3 Gb of RAM. The operating system is Ubuntu 10.10.

4.4.1 The Matrix Class

The Matrix class is the core component of all the finite automata cryptosystems implemented. It has various functions. It allows creation of (8×8) bit binary matrices as well as (8×1) binary column vectors, allows matrices to be added and multiplied, and creates random full rank matrices using Gaussian Elimination and pivoting techniques. The UML class diagram of the Matrix class is illustrated in Figure 4.1.

Requirements

The Matrix class must satisfy the following requirements.

- Generate random (8×8) binary matrices on demand.
- Generate random (8×8) binary full rank matrices on demand.
- Generate random 8 bit vectors on demand (for the starting states).
- Generate (8×8) identity matrices on demand.
- Generate the inverse of a given full rank binary matrix.
- Add and multiply 8x8 binary matrices and 8 bit vectors.

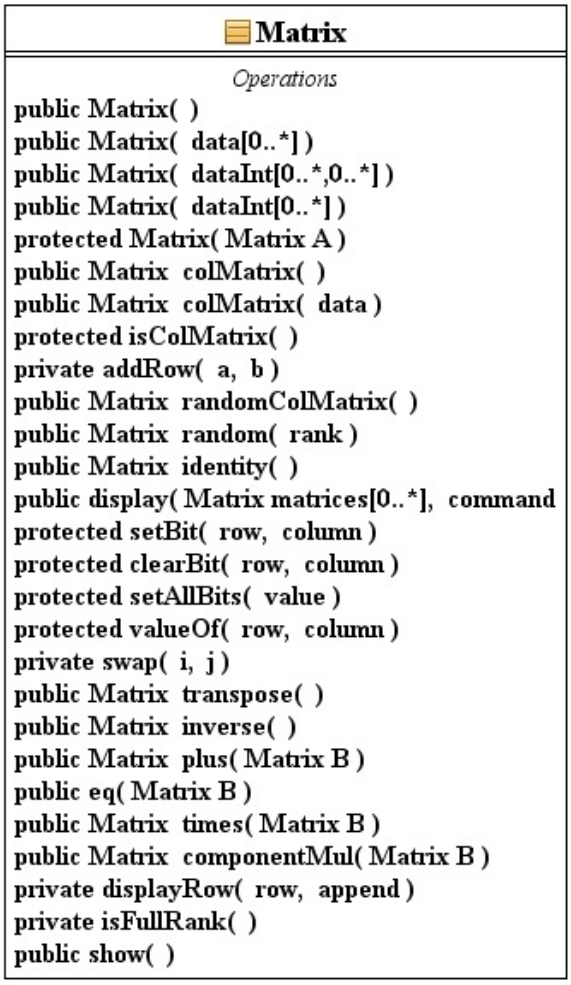


Figure 4.1: UML class diagram of the Matrix class

- Check if a given matrix is full rank or not.
- Swap specified rows of a given binary matrix.
- Add specified rows of a given binary matrix.
- Display either a single matrix or a sequence of matrices on the command line
- Try to ensure lowest possible number of operations for greatest speed.

Design

The Matrix class fulfils all specification requirements. It uses a cryptographically secure pseudo random number generator provided by the java "Secure-Random" [4]

class for random generation of the matrices. It can also create identity matrices, the inverse of a given matrix and the transpose of a given matrix. It allows to display either single matrices or an equation involving matrices as a simple command line display. It can check if a given matrix is full rank or not using Gaussian Elimination and pivoting techniques and contains general methods to perform bit operations on the matrices as well as swap rows within matrices etc. The Matrix class is dynamically instantiated as an object by the other classes. All the other classes depend on the Matrix class for their operations.

4.4.2 The Transform Class

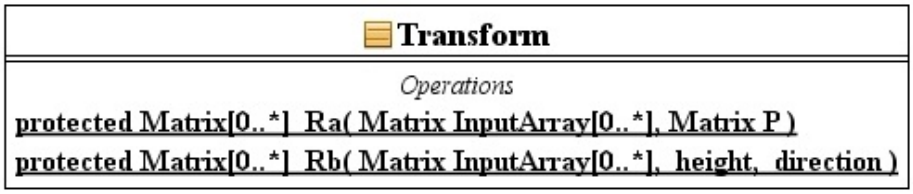


Figure 4.2: UML class diagram of the Transform class

The Transform class is responsible for the R_a and R_b transformations as detailed in Chapter 2. These function form the core of the finite automata generation process. Thus proper operation of this class is critical to correct encryption and decryption.

Requirements

The Transform class must satisfy the following requirements.

- Take in an array of $n(8 \times 8)$ binary matrices as well as a full rank binary matrix P and perform an R_a transformation on it using principles discussed in Chapter 2. It should return the transformed matrices as a new array.
- Take in an array of $n(8 \times 8)$ binary matrices and perform an R_b transformation on it using principle discussed in Chapter 2. It should be able to perform the transformation using any given height. It should also be able to perform the transformation in either direction i.e. from left to right or right to left and should resize the resultant matrix array accordingly. It should return the transformed matrices as a new array.

- Ensure that the transformations utilize the lowest number of operations necessary so as to be as efficient as possible.

Design

The transform class fulfils all specification requirements. For the R_a transformation, it takes in the original matrix array which comprises the automata as well as a full rank matrix and multiplies each component with this matrix. It performs R_a transform on an array of Matrices passed in correct order and returns the result as a new array of Matrices. The R_b transformation function performs a x height R_b transform either from left to right or from right to left on an array of Matrices passed in correct order and returns the result as a new array of Matrices. The transform class is called during the generation of the finite automata itself and does not play a part in the actual encryption and decryption. It is primarily used to generate the keys for the cryptosystem.

4.4.3 The FAPKC3 Class

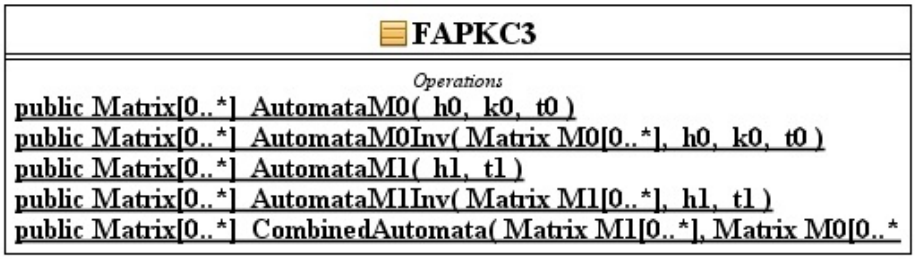


Figure 4.3: UML class diagram of the FAPKC3 class

The FAPKC3 class is responsible for the generation of the linear and non-linear automata as described earlier as well as their inverses. It is also responsible for the generation of the combined automata which comprises of the combination of a set of previously generated linear and non-linear automata.

Requirements

The requirements of the FAPKC3 class are as follows:

- Generate a random linear finite automata as per specifications in Chapter 2. It should be capable of generating linear automata for any value of h, k and τ subject to given restrictions.
- Generate a random non-linear finite automata for any value of h_1 and τ_1 as specified in Chapter 2.
- In the case of the DES-based finite automata cryptosystem discussed in Chapter 3, it should be capable of generating automata based on matrices derived from the keys and provided as input instead of random automata. All other specifications remain the same in this mode.
- Generate inverses of both linear and non-linear automata, given the original linear/non-linear automata array and the array of matrices used for R_a transformation.
- Generate a combined automata as specified in Chapter 2, given a pair of linear and non-linear automata as input.
- Perform the above operations as efficiently and with the least number of operations possible.

Design

The FAPKC3 class conforms to all the requirements. All functions have variable parameter and are capable of creating automata based on any combination of values for h, k and τ as described earlier (subject to certain constraints). This class also incorporates standard checks and safeguards to ensure that the automata are correctly generated in order to ensure accurate encryption and decryption. It is also capable of displaying detailed information about the actual generation of the automata if the relevant debug flags are set. It can also generate automata from predetermined matrices as required by the DES-based implementation.

4.4.4 The Crypto Class

Once the finite automata are created by the FAPKC3 class, they need to be provided with the relevant input consisting of the bytes of the plaintext and the starting states. The Crypto class is responsible for performing the actual processing of the finite automata during encryption and decryption.

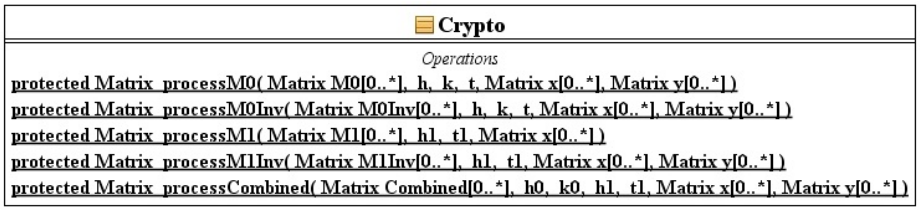


Figure 4.4: UML class diagram of the Crypto class

Requirements

- Accept the finite automata and their inputs and process them to return the result.
- Process the inverses of the finite automata and return the results.
- Process the combined automata and return the results.

Design

The Crypto class is responsible for providing the finite automata created by the FAPKC3 class with their inputs. It contains different functions for processing the linear automata (termed as M_0), the non-linear automata (termed as M_1), as well as their inverses and the combined automata. All functions accept the array of matrices comprising the automata, the array of matrices comprising the plaintext and starting states and other parameters and process them in order to return the final result.

4.5 Software Architecture and Design - Single Key Cryptosystem

The SecretKey, encryptSingleKey and decryptSingleKey classes are used for single key automata. The SecretKey class is a serializable object in Java and so it can be stored as a key file.

4.5.1 The SecretKey Object

The SecretKey Object contains the key used for the single key cryptosystem. It consists of the automata for the single key cryptosystem, the starting states used for

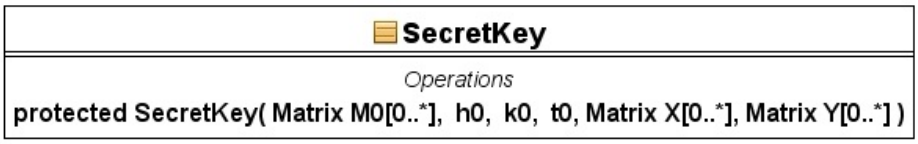


Figure 4.5: UML class diagram of the SecretKey object

the automata and the automata parameters (h, k and τ). This class is serializable in Java so that it is capable of being written to a file. The secret key file is generated randomly on demand by the encryptSingleKey class during encryption. It is required by the decryptSingleKey class in order to successfully decrypt the ciphertext.

4.5.2 The EncryptSingleKey Class

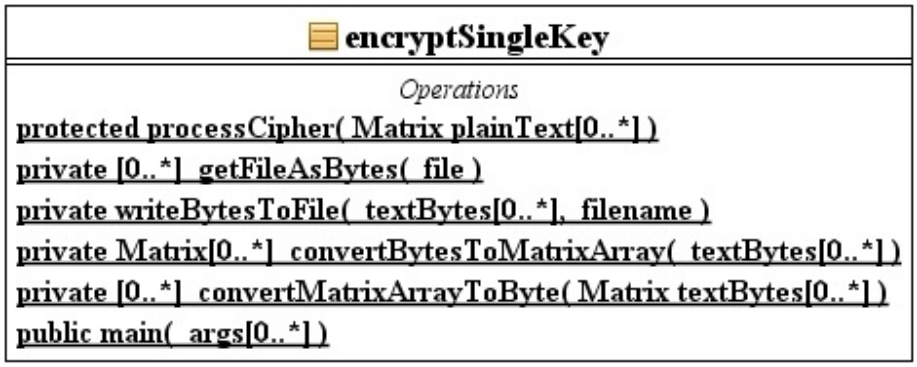


Figure 4.6: UML class diagram of the encryptSingleKey class

The EncryptSingleKey Class is responsible for the actual encryption process of the single key cryptosystem. It reads in the plaintext and converts it into an array of Matrices and then processes it using the generated automata.

Requirements

The requirements of the EncryptSingleKey class are as follows.

- Read in the text bytes and convert it into an array of Matrices.
- Generate a new random secret key if it is not specified and write it to a file as a SecretKey object. The secret key consists of a linear automata with completely variable parameters h_0, k_0 and τ.

- Process the plaintext using the secret key and generate the ciphertext as per specifications detailed in Chapter 2.
- Write the ciphertext to a file.

Design

The EncryptSingleKey class contains methods to read in the plaintext as bytes, to convert the bytes to a Matrix array, to process the Matrix array and generate the ciphertext and to convert the ciphertext to bytes and write it to a file. It is capable of generating a linear automata based secret key and writing it to a file using the SecretKey object. The encryptSingleKey class reads in an existing secret key generated previously or generates one randomly as per the requirement. It is responsible for reading in the bytes of the plaintext and processing them using the Crypto class. After the processing, it writes the result to the file. τ random characters are appended to the plaintext and it is then processed byte by byte. Though our implementation uses a linear autoamton for encryption, it can be easily modified to generate and use a non-linear automaton as well, with a few minor modifications to the processCipher class.

4.5.3 The DecryptSingleKey Class

Figure 4.7: UML class diagram of the decryptSingleKey class

The decryptSingleKey class similarly uses the inverse functions of the Crypto class to perform decryption. The single key cryptosystem implemented here uses

the linear automaton (or automaton M_0) of the Crypto class. However, simply by changing functions, it can be made to use the non-linear automaton (or automaton M_1) for encryption and decryption.

Requirements

The requirements of the decryptSingleKey class are as follows.

- Read in the ciphertext and convert it into an array of Matrices.
- Read in the secret key from file. The secret key will consist of the linear automaton, the starting states and the configuration parameters.
- Generate the inverse of the automaton using the FAPKC3 class.
- Process the ciphertext using the secret key and as per specifications detailed in Chapter 2.
- Write the plaintext to a file.

Design

The design of the decryptSingleKey class is similar to that of the encryptSingleKey class. However, it generates the inverse of the automata using the FAPKC3 class before processing the ciphertext. τ characters at the beginning of the processed ciphertext need to be discarded in order to retrieve the plaintext, which is then written to a file.

4.5.4 Syntax

The syntax for running the single key encryption program is:

java encryptSingleKey <inputfile ><outputfile ><secretkey ><h ><k ><τ >

The syntax for running the single key decryption program is:

java decryptSingleKey <inputfile ><outputfile ><secretkey >

Note that the decrypt program does not need the h, k and τ parameters to be specified since it is already part of the secret key.

4.6 Software Architecture and Design - Public Key Cryptosystem

The public key cryptosystem uses the GenerateKeys, encrypt and decrypt classes along with the core components. It also makes use of two new objects for the public and the private keys. It is similar in operation to the single key cryptosystem except for the fact that the encryption and decryption processes do not use the same key. While encryption is carried out using the public key, decryption is carried out using the private key.

4.6.1 The PriKey and PubKey Objects

The PriKey and PubKey classes are simply serializable objects in Java which comprise the private and the public keys respectively. The private key consists of a linear and a non-linear automata, their starting states and the configuration parameters, while the public key consists of a combined automata generated from the two linear and non-linear automatas, the starting states and the configuration parameters. These objects are serializable so that they can be written to file. They are used by the encrypt and decrypt classes for encryption and decryption, respectively.

4.6.2 Configuration Settings

The configuration settings (viz. h_0, k_0, τ_0, h_1 and τ_1 for both linear and non-linear automata) for the public key cryptosystem are contained in the "ppk.config file" (unlike the single key cryptosystem which takes these parameters on the command line). This file is read by both the encrypt and the decrypt classes during processing.

4.6.3 The GenerateKeys Class

The public key cryptosystem has an additional component, the GenerateKeys class. Prior to encryption and decryption, the GenereateKeys class has to be invoked in order to create the private and the public keys. These keys are then used for encryption and decryption.

Requirements

The requirements of the GenerateKeys class are as follows.

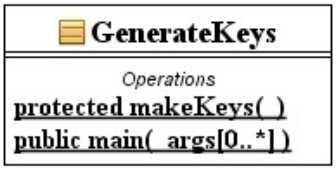


Figure 4.8: UML class diagram of the GenerateKeys class

- Generate random linear and non-linear automata and their combined automata using the FAPKC3 class.
- Generate suitable starting states for the linear and non-linear automata and use these to generate suitable starting states for the combined automata.
- Create the PubKey and PriKey objects for encryption and decryption, respectively.

Design

The GenerateKeys class follows the specifications outlined in Chapter 2 for generation of a random linear and non-linear automata and then uses these to compute the combined automata. Similarly it also computes random starting states for the linear and non-linear automata and combines them to generate a consistent starting state for the combined automata. By consistent starting state, we mean that this state is always the same for the same secret key. Proper operation of this class is crucial for correct encryption and decryption of the public key cryptosystem.

4.6.4 The Encrypt and Decrypt Classes

The operation of the encrypt and decrypt classes are similar to those of the encryptSingleKey and decryptSingleKey classes of the single key cryptosystem with the following exceptions.

- It uses the public and the private keys generated by the GenerateKeys class of the public key cryptosystem for encryption and decryption, respectively.
- Encryption is performed using the combined automata which is part of the public key. Decryption on the other hand, is performed using the two individual

inverses of the component linear and non-linear automata, which are used to generate the combined automata. These inverses are part of the private key.

- The linear and non-linear automata have completely variable parameters h_0, k_0, τ_0, h_1 and τ_1. Hence the encrypt and decrypt classes are modified to deal with the additional automata and their configuration settings.

4.6.5 Syntax

In order to use the public key cryptosystem, we first need to generate the keys using the GenerateKeys class. The syntax for generating the keys is:

java GenerateKeys <publickeyname ><privatekeyname >

Note that the configuration settings file "ppk.config" must be present in order for the GenerateKeys command to work. Also, keys once generated can be used multiple times by the encrypt and decrypt classes.

The syntax for running the public key encryption program is:

java encrypt <inputfilename ><outputfilename ><publickeyname >

The syntax for running the public key decryption program is:

java decrypt <inputfilename ><outputfilename ><privatekeyname >

4.7 Software Architecture and Design - DES Based Cryptosystem

The DES-based cryptosystem is implemented as per the specifications in Chapter 3. Its core operation is similar to encryption and decryption using the public key cryptosystem except for the fact that it is designed to be a probabilistic encryption scheme based on a 128-bit key. This key is used to generate sixteen 96 bit sub keys using an algorithm similar to that used in the Data Encryption Standard (DES). Also, despite being a stream cipher, its operation is split up into blocks where each block is encrypted in turn by multiple linear and non-linear automata pairs. This has the same effect as encrypting each block with a combined automata constructed out of the respective automata pair.

4.7.1 The FA Object

The FA Object is used internally for encryption and decryption. It contains the collection of μ pairs of linear and non-linear automata required by the algorithm to work. It also contains the starting states for the initial M_0 (linear) and M_1 (non-linear) automata which are generated as part of the key generation algorithm. This class is not serializable like the other objects since it need not be written to file. It is generated dynamically each time either encryption or decryption is performed using the DES-based cryptosystem.

4.7.2 Configuration Settings

There are two files which provide the DES-based cryptosystem with its configurations. One is the "FAC.key" file which simply contains the 128-bit key used by the cryptosystem. The other is the "FAC.config" file which is similar to the "ppk.config" file used in the public key cryptosystem, except that in addition to the h_0, k_0, $\tau 0$, h_1 and τ_1 parameters, it also specifies the special parameter μ to determine how many pairs of linear/non-linear finite automata will be generated.

4.7.3 The GenerateKeys Class

The GenerateKeys Class is responsible for generating the sixteen 96-bit subkeys from the main 128-bit key. It follows the specifications detailed in Chapter 3.

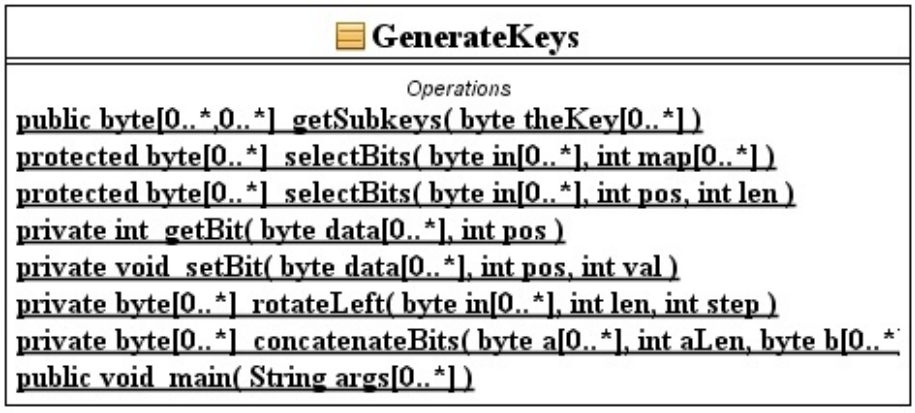


Figure 4.9: UML class diagram of the DES GenerateKeys class

Requirements

The requirements of the DES GenerateKeys Class are as follows.

- Acquire the main 128-bit key and transform it into sixteen 96-bit keys using permutation table $PC-1$ and shift table $SH-1$ using procedures as specified in Chapter 3.
- Return the created sub-keys as an array of bytes to the calling function.

Design

The main function in the DES GenerateKeys class is the getSubKeys function. This function takes in the 128-bit key and applies all required transformations to it to generate the sixteen 96-bit subkeys. It then returns the subkeys as an array of bytes to the calling funcition.

4.7.4 The GenerateFA Class

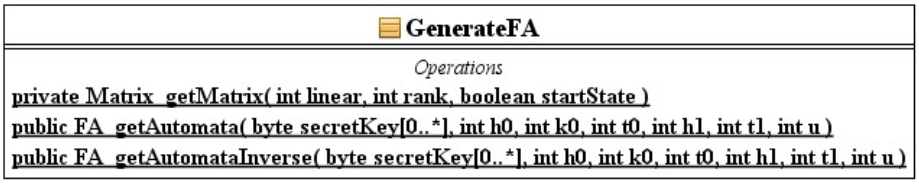


Figure 4.10: UML class diagram of the GenerateFA class

The GenerateFA class is responsible for interfacing with the GenerateKeys class and generating the required number of finite automata using the generated subkeys.

Requirements

- Provide a getMatrix function which is capable of generating either (8×8) binary matrices for use in the construction of the automaton or 8-bit vectors for the starting states. It should be capable of generating random general or full-rank matrices on demand using techniques specified in Chapter 3.
- Use the generated matrices to create μ linear and non-linear automata pairs (or their inverses depending on whether encryption or decryption is being carried out) and return it as an FA object.

- The FA object also contains the starting states for the initial linear and non-linear automata required for encryption. The GenerateFA class is also required to create these starting states and write it to the returned FA object.

Design

The GenerateFA class fulfils all its specified requirements. It is capable of using the sixteen 96-bit subkeys in order to create the required number of linear/non-linear automata and their starting states. It returns the results as an FA object. It is called upon by both the encrypt and the decrypt classes for their operations. Depending on which class calls it, it returns either the list of finite automata or their inverses.

4.7.5 The Encrypt and Decrypt Classes

The encrypt and decrypt classes for the DES-based finite automata cryptosystem are similar to that used in both the single key and the public key cryptosystems. However, they are specially adapted as per the specifications in chapter 3. In particular they differ in the following respects.

- The linear and non-linear automata generation processes are not based on random matrices. The matrices used for their generation are derived from the 128-bit key using the GenerateKeys and GenerateFA classses.
- The encryption is done using multiple linear and non-linear automaton pairs depending on the value of the μ parameter. For instance, if the value of μ is 7, then seven pairs of linear and non-linear automata are generated. Each block is encrypted in sequence with one pair of linear and non-linear automata (similar to encrypting with their combined automata). Once the μ^{th} block is encrypted, the next block reverts back to the first automata pair for encryption. It is clear in this particular case that for the encryption to achieve its maximum security, there must be at least μ blocks of plaintext. If this is not the case, then all the μ automaton pairs will not be used for encryption since there are less than μ blocks to encrypt.

4.7.6 Syntax

The syntax for running the DES based FA encryption program is:

java encrypt <inputfilename ><outputfilename ><secretkeyname >

The syntax for running the public key decryption program is:

java decrypt <inputfilename ><outputfilename ><secretkeyname >

Note that the secret key file for the DES-based cryptosystem simply consists of a 128-bit string which is the key. It is not the same as the secret key file used for the single key cryptosystem which actually contains the finite automata and its starting states. In the DES-based cryptosystem, the finite automata is dynamically (re)constructed using the 128-bit key. Also, in order for the cryptosystem to work, the configuration file "FAC.config" must be present.

Chapter 5

Prototype Testing and Analysis

5.1 Unit and Functional Testing

The reliability and correctness of all three implemented algorithms have been extensively tested by using them to encrypt and decrypt several types and sizes of files. All sub processes were also tested independently for consistency. Since all the parameters h_0, k_0, τ_0, h_1 and τ_1 are variables, the tests were carried out by putting the program in a loop and testing each one of the parameters through their entire range. Some of the tests carried out include the following.

- The R_a and R_b transformations were tested for accuracy by outputting the results for various transformations on screen using the display function of the Matrix class. These results were then manually verified to be correct. Similar unit tests were conducted on the other parts of the system such as the Crypto class, the key generation classes and the FAPKC3 classes.

- Reliability of encryption and decryption were tested by generation of multiple 1MB random plaintexts and then encrypting and decrypting them using various different keys.

- The program was run in a nested loop with h_0 running from 1 to 25, k_0 running from 0 to 20, τ_0 running from 1 to 7, h_1 running from 1 to 25, τ_1 running from 1 to 7 and μ running from 1 to 7. Each time a different randomly generated 1MB plaintext was used and checked for consistency in encryption and decryption. The program was designed to immediately stop with an error if any of the

decryptions did not match the original plaintext. Over 5000 iterations were performed on all three cryptosystems and all tests were successful.

- Multiple tests were carried out by changing one byte of the key in the DES-based FA cryptosystem and trying to decrypt a message encrypted with the previous key. The results were completely random and the decrypted text did not reveal any information about the original plaintext.
- Multiple tests were carried out to ensure that changing a single byte of the plaintext resulted in a completely different ciphertext. All tests were successful with 65 percent (on average) of the bytes in the ciphertext completely changing with change in just 1 byte of the plaintext.Further, statistical tests carried out on the ciphertexts show that the frequency distributions of the 35 percent of unchanged bytes are completely different in each iteration of the encryption algorithm.
- The ciphertext is also completely different when the same plaintext is encrypted more than once using the same key, demonstrating the probabilistic nature of the cryptosystem. The previous result is also true in this case.

5.2 Security Analysis

The security of the FAPKC3 cryptosystem has been discussed in [22] and [20]. Since our DES-based FA cryptosystem consists of the same core components used in these cryptosystems, the core security is at least as much as that afforded by these cryptosystems. In addition, the use of 2μ different automata pairs and keys along with the block based encryption scheme introduces further randomness in the algorithm and increase security drastically.

5.2.1 Probabilistic Encryption/Decryption

Most of the commonly used single key cryptosystems in use today such as DES and AES are deterministic in nature. What this means is that, given a particular plaintext and a particular key, they always encrypt to the same ciphertext. To be semantically secure, i.e. to hide even partial information about the plaintext, an encryption algorithm must be probabilistic [2]. In case of public key cryptosystems, this property is very important since the public key is common knowledge and if an

adversary guesses the contents of the plaintext, he could simply encrypt his guess and compare it to the original plaintext under the public key. Symmetric block ciphers could also achieve this property by encrypting in a chaining mode such as Cipher Block Chaining (CBC) which was originally invented by IBM [24]. However, our DES-based enncryption algorithm integrates random padding into every block of text encrypted, resulting in a truly probabilistic symmetric encrytpion scheme which is semantically secure and produces a different ciphertext each time encryption is done - even if the plaintext and the keys remain unchanged!

There are two reasons for the probabilistic nature of the DES-based FA algorithm.

- The reason the encryption is probabilistic in the DES based FA cryptosystem is that for every 64-byte block of plaintext encrypted, $\tau_0 + \tau_1$ random characters are appended at the end for encryption with the linear/non-linear automata pair. As a property of the finite automata, the decryption algorithm discards the specified number of bytes at the beginning of the decrypted text rather than at the end. This is in contrast to simply padding the plaintext with random bytes at a specified location and discarding them from the same location while decrypting. In our cryptosystem, though the random bytes are added at the end of the plaintext while encrypting, they are removed from the beginning while decrypting due to the nature of the FA cryptosystem. This is significant because it produces a cascading effect which alters the entire block depending on the random bytes added at the end.

- As part of the design of the cryptosystem, $h_0 + k_0$ 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 non-linear 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 states for the next block are completely random and result 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 attack (IND-CPA) [1]. 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.

5.2.2 Multiple Keys and Alternating Automata types

As detailed in Chapter 3, 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. Since the keys are used cyclically, no two automata generated are ever the same. In effect, the encryption algorithm uses 2μ different keys to construct as many different automata. In addition, half of the constructed automata are linear and the other half are non-linear and each pair is applied alternately on successive blocks of text. This introduces a high degree of complexity making cryptanalysis difficult.

5.2.3 Statistical Analysis

The statistical tests conducted on the DES-based FA cryptosystem use ENT - a pseudorandom number sequence test program [6]. They have also been influenced in part by the tests conducted in [17]. In order to analyze if statistical regularities in the plaintext carry over into statistical regularities in 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 (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.

The tests were performed on the original plaintext, the ciphertext and the XOR of the plaintext and ciphertext. All tests are conducted with $h_0 = 1, k_0 = 2$, $\tau_0 = 1, h_1 = 2, \tau_1 = 2$ and μ=7. This represents a relatively low security level afforded by the system. Obviously, the security can be greatly increased by increasing these values, but the tradeoff lies in the speed of encryption/decryption.

Five types of test were conducted. These tests and their results are as follows.

Tests for Entropy

Entropy was first introduced by Claude Shannon [14] 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 can be considered to be random. Our tests show that files encrypted with the DES-based FA cryptosystem generally demonstrate high entropy.

Type of Plaintext (Size: 4 KB)	Plaintext Entropy (bits/byte)	Ciphertext Entropy (bits/byte)		XOR (Pt ^ Ct) Entropy (bits/byte)	
		Min	Max	Min	Max
Pt1 (AAAA...HHHH...)	3.002000	7.951620	7.954884	7.950837	7.955376
Pt2 (000000000.....)	0.003282	7.458688	7.961747	7.460545	7.961712
Pt3 (11111111.....)	0.003282	7.703361	7.960781	7.703615	7.960766
Pt4 (Random English)	4.874877	7.949182	7.961380	7.949731	7.957172

*This table shows the results of multiple entropy tests carried out using both random and uniform keys

*Entropy approaching 8 means that the data is very dense and is essentially random

Figure 5.1: Results for entropy tests

Chi Square (χ^2) Tests

The χ^2 test [5] 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. This is essentially the degree to which the sequence tested is suspected of being non-random [6]. If the percentage is greater than 99 percent or less than 1 percent, the sequence is almost certainly not random.

It is known that timing radioactive decay events result in a completely unpredictable and random number sequence since it is impossible to predict when a given atom will decay [11]. Using this test, the chi-square distribution for 50,000 samples of a genuine random sequence created by timing radioactive decay events is 249.51, and randomly would exceed this value 40.98 percent of the time [6].

Type of Plaintext (Size: 4 KB)	Plaintext		Ciphertext				XOR (Pt ^ Ct)			
	χ^2 Dist	Random %	χ^2 Dist		Random %		χ^2 Dist		Random %	
			Min	Max	Min	Max	Min	Max	Min	Max
Pt1 (AAAA...HHHH...)	126912.13	0.01	255.87	271.74	22.52	47.28	255.50	275.73	17.78	47.94
Pt2 (000000000.....)	1043968.13	0.01	212.90	284.85	9.63	97.43	212.78	285.90	8.87	97.47
Pt3 (11111111.....)	1043968.13	0.01	220.77	257.12	45.10	94.05	220.90	258.00	43.58	93.98
Pt4 (Random English)	77677.25	0.01	216.15	280.35	13.20	96.30	240.13	282.98	11.02	73.95

*Random % denotes how often the χ^2 Dist value would be randomly exceeded.
*Any value between 1% and 99% is accepted as random.
*This table shows the results of multiple tests carried out using both random and uniform keys.

Figure 5.2: Results for χ^2 tests

Arithmetic Mean Tests

For the Arithmetic Mean test [6] we add the values of all the bytes in the file and divide it by the file length. If the data is close to random, this should be about 127.5 since there are 256 possible ASCII (American Standard Code for Information Interchange) values that each byte of data can represent. If the mean departs from this value, the values are consistently high or low. Almost all results for our cryptosystem show values very close to 127.5.

Type of Plaintext (Size: 4 KB)	Plaintext A.M.	Ciphertext Avg. A.M	XOR (Pt ^ Ct) Avg. A.M
Pt1 (AAAA...HHHH...)	68.4800	128.1556	127.9576
Pt2 (000000000.....)	47.9900	128.4913	127.6275
Pt3 (11111111.....)	48.9900	127.4694	126.7773
Pt4 (Random English)	70.6484	127.6016	127.3298

*For random data the mean should be close to 127.5
*Results for multiple tests using both random & uniform keys

Figure 5.3: Results for arithmetic mean Tests

Tests for Monte Carlo value for Pi (π)

In the test for the Monte Carlo Value for Pi [8], 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. A 500000 byte file created by timing radioactive decay events (as explained earlier) yields the Monte Carlo value for Pi as 3.143580574 (error 0.06 percent) [6].

Type of Plaintext (Size: 4 KB)	Plaintext		Ciphertext		XOR (Pt ^ Ct)	
	M.C. Pi	Error %	Avg. M.C. Pi	Avg. Error %	Avg. M.C. Pi	Avg. Error %
Pt1 (AAAA...HHHH...)	4.000000	27.32	3.121522	0.64	3.168374	0.85
Pt2 (000000000.....)	4.000000	27.32	3.098096	1.38	3.121522	0.64
Pt3 (11111111.....)	4.000000	27.32	3.039531	3.25	3.168379	0.85
Pt4 (Random English)	4.000000	27.32	3.144948	0.11	3.156661	0.46

*For random data, the Monte Carlo value of Pi should be close to the actual value of Pi = 3.141592 (approx)
*Results for multiple tests using both random & uniform keys.

Figure 5.4: Results for monte carlo tests

The Serial Correlation Coefficent

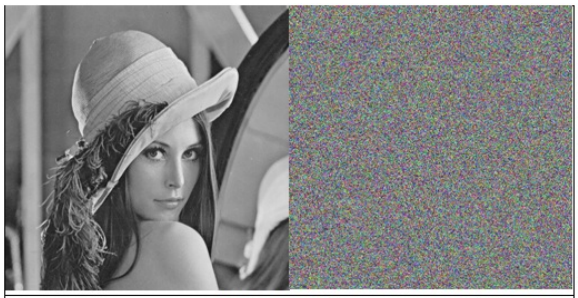
The degree to which neighboring bytes are related to each other are measured by the Serial Correlation Coefficient. The lower the relation, the lower will be the value of this measure. If the bytes are totally uncorrelated then the Serial Correlation Coefficient would be close to zero [6]. We can see that in our results this value almost always converges very close to zero.

Type of Plaintext (Size: 4 KB)	Plaintext S.C.C.	Ciphertext Avg. S.C.C.	XOR (Pt ^ Ct) Avg. S.C.C.
Pt1 (AAAA...HHHH...)	0.719264	-0.012199	0.009118
Pt2 (000000000.....)	-0.000244	0.001394	0.014184
Pt3 (11111111.....)	-0.000244	0.000779	0.006640
Pt4 (Random English)	0.468316	-0.015820	0.009958

*For random data the Serial Correlation Coefficient should be close to 0.
*Results for multiple tests using both random & uniform keys.

Figure 5.5: Results for serial correlation coefficient tests

The results of all tests conducted on the DES-based FA cryptosystem are satisfactory even at the basic security level detailed above. Depending on the application, the security can be vastly increased by choosing greater values for the parameters, at the cost of speed of computation. The randomness of the encrypted plaintext can be assessed from the figure showing the result after encrytping a bitmap image with our cryptosystem. As is evident from both the encrypted image and the statistical figures, there are no 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	

Figure 5.6: Results after encryption of a bitmap image

5.3 Performance

The performance of the single key, the public key as well as the DES based variants of the implementations were tested on both an Intel Core 2 Duo 2.16 Ghz machine with 3 GB of RAM and the Nokia N900 Internet Tablet which has an ARM Cortex A8 600 MHz processor with 256 MB of RAM. These speeds might be improved drastically if implemented in a language such as C versus Java and if the program is further optimized. The testing results shown in the figure denote the average of 50 different tests conducted.

	DES Based FA		FA – Single Key	FA – Public Key
Intel Core 2 Duo 2.16 Ghz, 3GB RAM	Enc:	31 ms.	24 ms.	32 ms.
	Dec:	46 ms.	24 ms.	37 ms.
Nokia N900 ARM Cortex A8 600 MHz, 256MB RAM	Enc:	406 ms.	219 ms.	379 ms.
	Dec:	554 ms.	230 ms.	521 ms.

- Average throughput: DES Based FA: 142 Kbits/sec; FA-Single Key: 286 Kbits/sec; FA-Public Key: 176 Kbits/sec
- Filesize: 256 bytes

Figure 5.7: Performance tests of single key and public key cryptosystems

5.4 Possible Improvements

5.4.1 Plaintext Padding

The current implementation of our DES-based finite automata cryptosystem is largely intended to be a proof of concept. Though complete in every respect, it does not incorporate plaintext padding. If the plaintext is smaller than 64 bytes in length, then the security of the algorithm will decrease considerably because in that case, only one linear/non-linear automata pair will be utilized for encrypting the only 64-bit block. Since the security is considerably enhanced as a result of alternating linear/non-linear automata pairs, with the encrypted values of the first block being used as the starting state of the automata in the second block, it is suggested that the plaintext be padded with bytes from one of the 16 subkeys at the end to make it a multiple of 64 bytes and then to add another extra block at the end using another one of the keys.

This serves two purposes. Firstly, based on cryptanalysis attempted on linear and non-linear automata based cryptosystems, it has been found that if the ending of the plaintext is known then it is easier to mount a successful attack on the traditional cryptosystem. Though our proposed algorithm makes this almost impossible because of the DES key generation algorithm and the use of μ pairs of linear/non-linear automata, appending the extra block at the end based on the sub-keys would ensure that the ending of the plaintext is never known, thus affording the system an extra

layer of security. However, if padding the plaintext this way, the algorithm must be capable of recognizing the start of the keys appended to the end of the plaintext and discarding them during decryption.

Secondly, even if the plaintext is smaller than 64 bytes (1 block) in length, padding increases the effective length of the plaintext to a minimum of two blocks. Therefore, at least two pairs of linear and non-linear automatas are used for encryption to improve security.

5.4.2 Automata Caching and Multi Threading

Our DES-based implementation does not cache the 2μ automata generated as part of the encryption process. This results in an average throughput of 142 Kbits per second using reasonable values for the variable parameters as compared to the DES algortihm of the Java Crypto class which is considerably faster. This could be vastly improved by caching the keys instead of regenerating them from the sub-keys each time the algorithm is run. Secondly, multiple threads could be used to pre-process each individual block while it is waiting for the encrypted result of the previous block to be computed. This, should speed up the algorithm geometrically. Also implementing in a low level language such as C may also result in drastic speed improvements. This is left as future work for further revisions of the algorithm.

Chapter 6

Conclusion

In this thesis, we introduced and implemented a single key cryptosystem as well as a public key cryptosystem based on FA as discussed in [9]. To the best of our knowledge, this is the first publicly available implementation of these cryptosystems. Having no prior precedent, implementation of these cryptosystems have been a non-trivial though rewarding task. We also proposed and implemented a new DES-based finite automata cryptosystem which utilizes the best qualities of both the linear and non-linear cryptosystems to create a probabilistic cryptosystem which has good security properties as we have demonstrated via statistical tests.

We tested the reliability, speed and security properties of our cryptosystem on both an Intel Centrino Dual Core 2.16 Ghz machine with 3GB of RAM as well as the Nokia N900 Internet Tablet and found them to be acceptable. Since our prototype is largely a proof of concept and is implemented in Java, we propose to further improve the speed and performance by implementing in a language such as C and also by introducing multithreading and automata caching.

We also plan to carry out further tests on the proposed cryptosystem and devise a formal proof of security in future. It is expected that over time, FA based cryptosystems, and particularly our DES-based variant, will earn credibility in the cryptographic world as a viable alternative to current cryptosystems and stands the tests of further cryptanalysis. This thesis makes a valuable contribution to the field of cryptography and serves as a stepping stone towards further research in the area of cryptosystems based on finite automata.

Bibliography

[1] S. Goldwasser and S. Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. *Annual ACM Symposium on Theory of Computing*, 1982.

[2] Shafi Goldwasser and Silvio Mical. Probabilistic encryption. *Special issue of Journal of Computer and Systems Sciences*, 28-2:270–299, 1984.

[3] J. Orlin Grabbe. The data encryption standard illustrated. *Laissez Faire City Times*, 2-28, 2003.

[4] Standard Edition 1.4.2 Java 2 Platform. Secure random class.

[5] Barry L. Nelson Jerry Banks, John S. Carson and David M. Nicol. *Discrete-Event System Simulation (Fifth Edition).* Prentice Hall, 2010.

[6] Inc. John Walker, founder of Autodesk and co-author of AutoCAD. Fourmilab ent - a pseudorandom number sequence test program.

[7] Yehuda Lindell Jonathan Katz. *Introduction to Modern Cryptography.* Chapman and Hall, CRC, August 2007.

[8] John H. Mathews. Monte carlo estimate for pi. *Pi Mu Epsilon Journal*, 5:281–282, 1972.

[9] Tommi Meskanen. On finite automaton public key cryptosystems. *Technical Report 408, TUCS - Turku Centre for Computer Science, Turku, Finland*, 2001.

[10] U.S. Department of Commerce National Bureau of Standards. Data encryption standard. *Federal Information Processing Standard (FIPS), Publication 46*, 1977.

[11] Asher Peres. When is a quantum measurement? *Annals of the New York Academy of Sciences, New Techniques and Ideas in Quantum Measurement Theory*, 400:438–448, 1986.

[12] M. Robshaw. Block ciphers. *RSA Laboratories Technical Report*, TR-601, 1995.

[13] M. Robshaw. Md2, md4, md5, sha and other hash functions. *RSA Laboratories Technical Report*, TR-101, 1995.

[14] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379423, 1948.

[15] Claude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28-4:656–715, 1949.

[16] William Stallings. *Cryptography and Network Security (Third Edition)*. Prentice Hall, August 2002.

[17] Hermetic Systems. Testing the me6 cryptosystem.

[18] Renji Tao. *Finite Automata and Application to Cryptography*. Tsinghua University Press, January 2008.

[19] Renji Tao and Shihua Chen. A finite automaton public-key cryptosystem and digital signatures (in chinese). *Chinese Journal of Computers*, 8:401–409, 1985.

[20] Renji Tao and Shihua Chen. Two varieties of finite automaton public key cryptosystem and digital signatures. *Journal of Computer Science and Technology*, 1:9–18, 1986.

[21] Renji Tao and Shihua Chen. A variant of the public key cryptosystem fapkc3. *Journal of Network and Computer Applications*, 20:283–303, 1997.

[22] Renji Tao, Shihua Chen, and Xuemei Chen. Fapkc3: A new finite automaton public key cryptosystem. *Journal of Computer Science and Technology*, 12 No. 4:289–304, 1997.

[23] Diffe W. and Hellman M. New directions in cryptography. *IEEE Trans. Information Theory*, 22:644–654, 1976.

[24] John L. Smith William F. Ehrsam, Carl H. W. Meyer and Walter L. Tuchman. Message verification and transmission error detection by block chaining. *US Patent 4074066*, 1976.