Software Implementation of Three Popular Cryptosystems

Akshay Kulkarni, Aravindhan Eswaran and Karthik Sambamoorthy

College of Computer and Information Science, Northeastern University Seattle, Seattle, WA 98109 USA

Abstract - This paper describes the working and backend implementation of a GUI based software tool of three of the most prevalent cryptosystems namely 1) RSA 2) ElGamal 3) AES. The software implementation of these theoretical cryptosystems comes with a lot of complexity and technicalities that needed to be addressed before the actual implementation is performed. Those nuances are descried in this paper. Also a short analysis of the times consumed by each of those procedures for encryption and decryption is described in the end. The paper also deals with the implications on the secureness of these algorithms.

***Index Terms*—Cryptography, RSA, ElGamal, AES**

I. Introduction

This paper details the nuances of implementing certain theoretical cryptosystems in the form of an interactive *Java* based GUI. Cryptography is one of the corner stones of Software security for ensuring confidentiality and providing authentication. Cryptographic algorithms are used to provide digital signatures, secure network communication, securing file transfers, encrypting and protecting classified information. Hence the ability to develop a working and fully functional system that performs encryption and decryption using these cutting edge algorithms deemed very valuable. This project is an attempt to gauge the difficulty involved in taking existing cryptosystems and providing a software implementation of the same. The GUI has been designed keeping in mind, the user pool. A user intending to generate both public and private keys can use the auto key generate functionality, which then produces keys for his usage. This tool may also be used by individuals that are already in possession of encrypting / decrypting keys and intend to use the keys encrypt / decrypt messages, but are unaware of the mathematical procedures behind them. This tool has also been designed keeping in mind the vulnerabilities of the cryptosystems involved. Currently it is deemed that 1024 bit length keys are more are less safe. So the user has a flexibility of specifying a bit length of utmost 1024 for key generation. The specific attacks and safety measures for each cryptosystem has been described in their respective sections.

II. RSA Cryptosystem

A. Introduction

RSA is widely known as one of the oldest and practicable public-key Cryptosystems that is used for online data transmission. It is the brain child of two MIT based computer scientists, Ronald Rivest and Adi Shamir, and a number theorist, Leonard Adleman (hence the name RSA), it was first publicized in the August 1977 issue of the *Scientific American*[1]. A variety of attacks have been deployed on RSA since its inception 40 years ago, but none of them are devastating. Since it has stood the test of time RSA is used in an assortment of applications like ensuring privacy and authenticity of email, securing remote login sessions and most importantly in encrypting internet credit card transactions.

One of the distinguishing and singular features of public-key Cryptosystems is the usage of asymmetric keys i.e. one key (public key) is used for encrypting while and entirely different key (private key) is used for decryption. The keys have a complex mathematical relationship that makes it either impossible or prohibitively expensive to figure out the latter from the former. Below is a brief description of the RSA encryption scheme:

1. Find two distinct prime numbers p and q of similar bit length.
2. Compute $n = pq$, the product n is used as a modulus for both the private and public key.
3. Also compute $\phi(n) = (p-1)(q-1)$. Φ is known as the Euler's totient function and is equal to the number of integers less than n that are co-prime to it.
4. Choose a random integer e where $1 < e < \phi(n)$, such that $gcd(e, n) = 1$ and $gcd(e, \phi(n)) = 1$.
5. Finally compute $d = e^{-1} \bmod \phi(n)$.
6. The public key is the integer pair (e, n). To encrypt a message m compute $m' = m^e \bmod n$.
7. To decrypt message m' compute $m = m'^{d} \bmod n$. The private key is the integer pair (d, n).

The proof for why RSA encryption works can be found in [1].

B. Description and Layout of the GUI

A Java based GUI application that simulates and runs the RSA algorithm in the background has been developed, the details of which are presented in this paper. The strength and security of the RSA encryption relies on the predicament that it not possible to compute the factors of a large integer, in this case n, in a reasonable time using a classical computer. Should n be factorable, it is easy to conceive how the impending attack will proceed. One may immediately compute the

Euler's totient of n from its component primes and thereby the inverse of e modulo $\phi(n)$, which is the private key. Hence it is paramount to ensure that the bit length of the component primes is sufficiently large enough to protect against the above mentioned attack. For this reason the GUI provides the user the flexibility of inputting a bit length of his or her choosing for the component primes. It is currently recommended that if n be of at least 2048 bits to be deemed safe. The GUI allows for a maximum of 1024 bit length primes to be generated. The GUI supports the generation of public and private keys given n and its component primes. But alternatively the user may choose to entirely ignore the key generation functionality and opt to pick their own pair of keys which will then be used for encrypting/decrypting any UTF-8 character based text, using the mathematical procedure mentioned in the introduction. There are input slots for each of these numbers. A snapshot of the GUI has been pasted below for reference.

C. *Software Implementation of RSA*

For the software implementation of the RSA encryption/decryption algorithm we heavily relied on the *java.math.BigInteger* library that comes with *Java 8*. This library has an in-built function that generates primes with a probability of failure of $1/2^{100}$. Ideally we would like such functions to be deterministic, but our literature survey has revealed that the best method for generating an arbitrary bit length prime is still probabilistic so we stuck with it. For computing the value of an integer raised to an exponent modulo another integer, the *gcd* of two integers, the inverse of

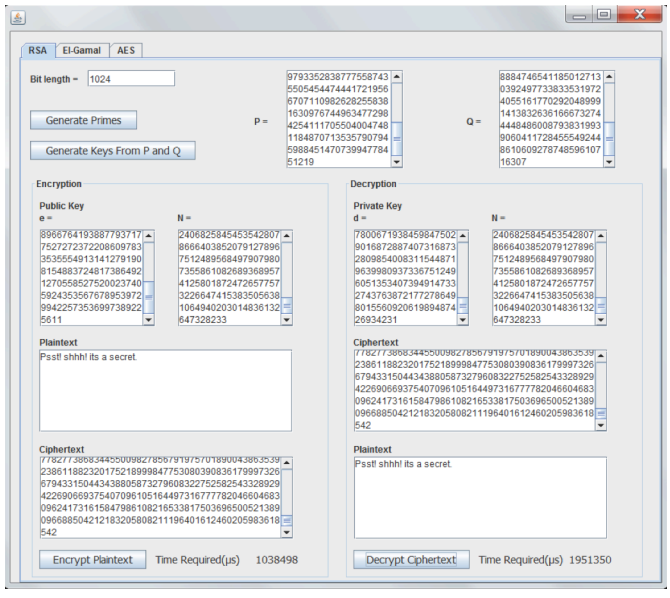

Fig. 1. A snapshot of the RSA tab the GUI toolbox depicting various input and output fields like the bit length, component primes public key, private key, plaintext and ciphertext.

an integer modulo another integer, we have again used in-built functions. The library is very handy and is the most recommended for developing applications involving modular arithmetic.

III. ElGamal Cryptosystem

A. *Introduction*

The ElGamal Cryptosystem is another prevalent public-key cryptosystem that uses an asymmetric key encryption algorithm and is used in the open source software *GNU Privacy Guard* for digital signing and the recent version of *PGP*. The algorithm was developed by Taher Elgamal[2], was published in an article in CRYPTO '84 a conference on the advances of cryptology. Find below the encryption/decryption scheme for the ElGamal cryptosystem:

1. Choose a prime p and integer α, where α is a primitive root of p i.e. every z where $1 \leq z \leq p - 1$ can be written as $z = \alpha^w \bmod p$.
2. Pick an integer a where $1 \leq a \leq p - 1$ and then compute $\beta = \alpha^a \bmod p$. (α, β, p) form the public key and (a) the private key.
3. Generate a random integer k where $1 \leq k \leq p - 2$.
4. For encrypting a message X compute two numbers $y_1 = \alpha^k \bmod p$ and $y_2 = X\beta^k \bmod p$.
5. For decrypting X from ciphertext y_1 and y_2 use $X = y_2(y_1^a)^{-1} \bmod p$.

The strength of the ElGamal encryption lies on the certitude that in-order to guess a from α, β and p, one needs to solve the discrete logarithm problem which was already been established as NP-hard. Hence it is very secure. The disadvantage of this scheme is that the ciphertext is twice as large as the plaintext. However, since k is a random integer and does not feature in the decryption part, we can have many ciphertexts for one plaintext thereby making it harder to decipher without the private key.

B. *Description and Layout of the GUI*

Similar to RSA a very user-friendly and intuitive GUI has been developed, there are fields and input slots for auto generating the public and private keys, alternatively the user may choose to generate keys himself and input them directly exclusively only for encrypting and decrypting his messages. Unlike RSA though the user does not have the flexibility of choosing the bit length for his keys. This shortcoming is further elaborated in the following section. Fig 2 depicts a snapshot of the GUI.

C. *Software Implementation of ElGamal*

Despite using the *java.math.BigInteger Java* library that was previously referred, implementing ElGamal seemed trickier than RSA for two reasons.

Firstly, as mentioned in the previous paragraph for ElGamal the option of entering a bit length was not provided to the user because if p were to be randomly generated for the input bit length, then computing the primitive root α for that specific p is not straightforward. One intuitive way of computing α is to test for every integer from 1 to p - 1 as a candidate for being a primitive root. Since these p could be as large as ~300 decimal digits, the procedure could turn out to take impractically large

time. There are other optimal ways for computing primitive roots but none could be implemented on a computer that could run in a reasonable time. Hence a very rudimentary approach was chosen for this, hardcoding p and α. Although this may seem unsafe at the onset, please note that p and α are part of the public key and will be exposed to attackers in any case. Due to the randomness of generation of a and unbreakable nature of the discrete logarithm problem, this approach also turns out to be safe. However one improvement that could be made on this strategy to generate a table of pre-selected p and α values for the user to choose from.

Secondly, the decryption portion of the algorithm requires us to compute $(y^a)^{-1} \bmod p$. In modular arithmetic there are ways to compute $y^a \bmod p$ without actually knowing y^a. This is extremely crucial because y and a are large integers, even primitive operations like addition or division on such integers are expensive what to talk of exponential operations. Hence ability to compute $y^a \bmod p$ without going through the trouble of computing y^a is very essential. However the formula for decryption has a compounded inverse on top of the exponent, computing which seems like we may need to compute y^a. But an alternative approach was discovered. In modular arithmetic the operations often tend to obey associativity. So another expression was tested for its equality to $(y^a)^{-1} \bmod p$. That expression is $(y^a \bmod p)^{-1} \bmod p$. It turned out that they were equal for multiple examples. Note that the latter expression may be easily computed without going over the grueling computation of y^a. Hence an attempt was made to prove the above equivalence, which was successful. Below is the theorem and its proof

Theorem: $(y^a)^{-1} \bmod p \equiv (y^a \bmod p)^{-1} \bmod p$ *where* $y,a,p > 0$.

Proof: Let,

$$k = y^a \bmod p$$
$$\Rightarrow y^a = k + \alpha p.$$

So,

$$(y^a)^{-1} \bmod p = (k + \alpha p)^{-1} \bmod p.$$

Let,

$$I = (k + \alpha p)^{-1} \bmod p$$

We need to show,

$$I = k^{-1} \bmod p$$

for the proof to be complete, because I is the LHS of the equation in the theorem and $k^{-1} \bmod p$ the RHS.

$$I = (k + \alpha p)^{-1} \bmod p$$
$$\Rightarrow I(k + \alpha p) = 1 \bmod p$$
$$\Rightarrow Ik + I\alpha p = 1 + \beta p$$
$$\Rightarrow Ik = 1 + (\beta - \alpha I)p$$
$$\Rightarrow Ik = 1 + \mu p \text{ where } \mu = \beta - \alpha I$$
$$\Rightarrow Ik = 1 \bmod p$$
$$\Rightarrow I = k^{-1} \bmod p$$

Also it is important to note that the algorithm used generates a new k for encrypting each character and hence the tool is safe from the same k attack (an attack that can be deployed which exploits the fact that the same k value has been used and one can gauge the private key).

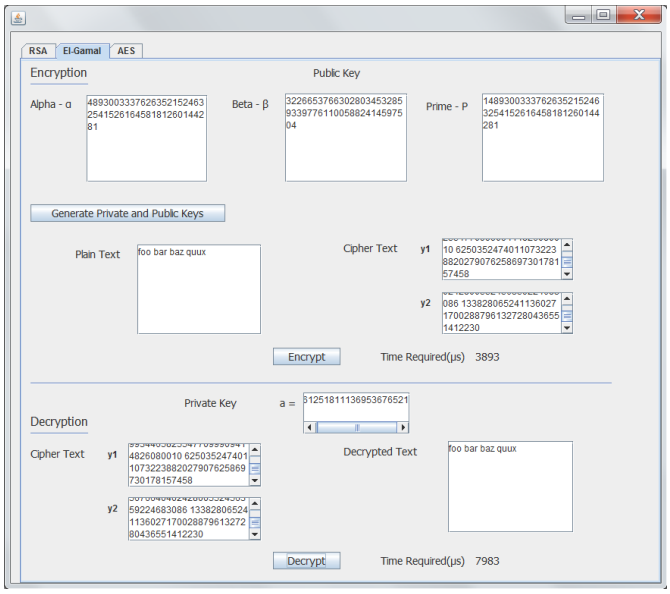

Fig. 2. A snapshot of the ElGamal tab of the GUI toolbox depicting various input and output fields like the public key, private key, plaintext and ciphertext

IV. AES Cryptosystem

A. Introduction

The Advanced Encryption Standard(AES) is an encryption / decryption standard and is based on a design principle known as substitution-permutation network, combination of both substitution and permutation. AES operates on a 4 x 4 column-major order matrix of bytes. AES was released by National Institute of Science and Technology in Nov, 2001. AES is based on Rijndael cipher, which is a generation of symmetric block cipher encryption technique that supports key sizes of 128, 192 and 256 bits. AES is a substitution linear transformation cipher, not requiring a Feistel network. It operates on block of data of size 128 bits. Find below the procedure for AES encryption / decryption:

1. AES allows for three different key lengths: 128, 192, or 256 bits. Most of our discussion will assume that the key length is 128 bits
2. Encryption consists of 10 rounds of processing for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys (Here we are implementing 128-bit encryption)
3. Each round includes on single-byte based substitution step, a row-wise permutation step, a column wise mixing and the addition of round key
4. The 128 bit of data is arranged into a matrix (State Matrix) as follows:

$$\begin{bmatrix} Byte_0 & Byte_4 & Byte_8 & Byte_{12} \\ Byte_1 & Byte_5 & Byte_9 & Byte_{13} \\ Byte_2 & Byte_6 & Byte_{10} & Byte_{14} \\ Byte_3 & Byte_7 & Byte_{11} & Byte_{15} \end{bmatrix}$$

B. Encryption

To encrypt a text, we need:

1. Keys in matrix format
2. Text in matrix format

Keys are generated from the 128-bit initial key using 'Rijndael's Key Schedule' and represented in a matrix similar to the state matrix.. The words (4 bytes of first column) of the key W_0, W_1, W_2, W_3 belong to the first round of encryption. Words W_4, W_5, W_6, W_7 belong to the second round and so on. The first round of keys are taken from the key input and from then on the words are calculated as follows:

$$W_i = W_{i-1} + W_{i-4}$$

for *i* values that are not multiples of 4.

For *i* values that are multiples of 4 there are 3 steps involved:

1. Bytes of W_{4k-1} are left shifted
2. S-Box value substitution is performed
3. $W_{4k} = R + W_{4k-4} + R_{conk}$ where R is the result obtained from the previous step and R_{conk} the value got from the r-con box.

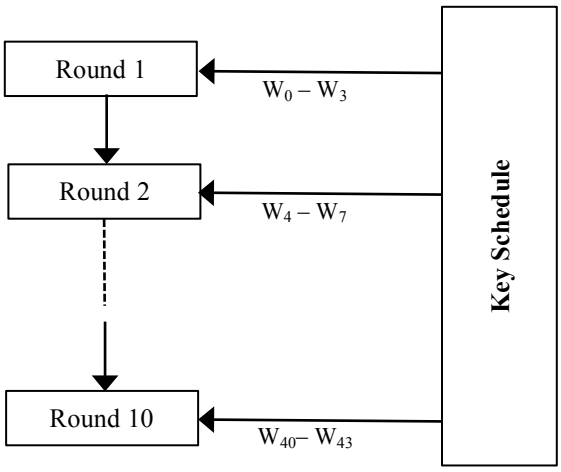

Fig. 3. A schematic model depicting the encryption key generation for many rounds of the AES algorithm

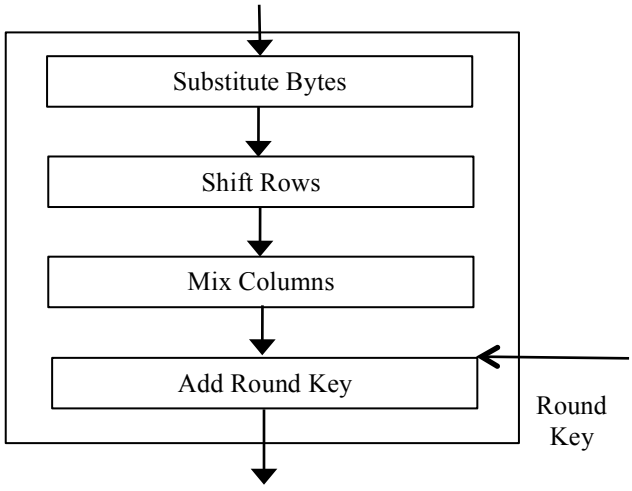

Fig. 4. A flow diagram for AES encryption scheme

C. Decryption:

Decryption of the cipher text is performed by the inverse of encryption steps. The main difference is we use inverse s-box values for the byte substitution step instead of the s-box step.

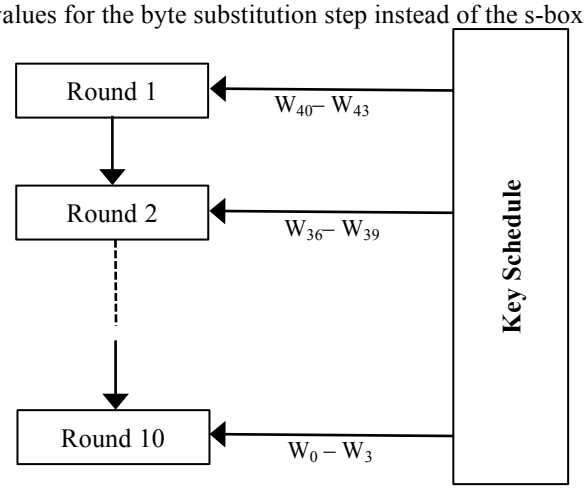

Fig. 5. A schematic model depicting the decryption key generation for many rounds of the AES algorithm

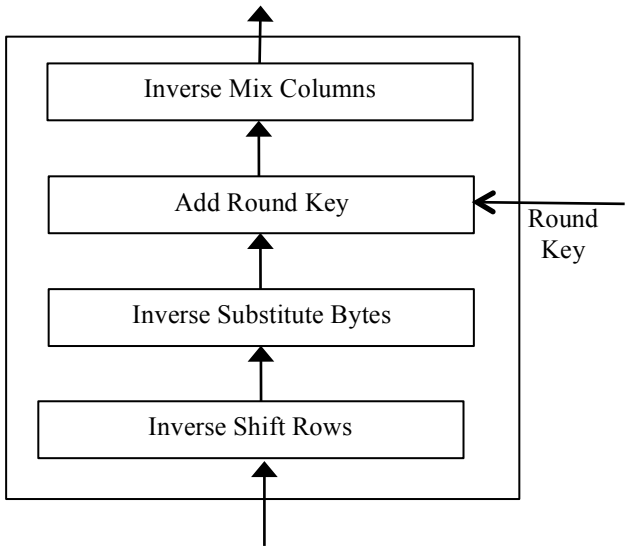

Fig. 6. A flow diagram for AES decryption scheme

D. Implementation:

The GUI application requires the following to be entered by the user:

- Key
- Plain text

'Key' is usually a 128-bit string. The key can contain alphanumeric characters and special characters too.

'Plain text' can be of any length and can contain any of special characters or alphanumeric numbers.

E. Working:

The text input given by user is converted into byte stream using the getBytes() method. Similarly, the key entered by the user is also converted into byte stream using the getBytes() method. These data are then passed as parameters to the encrypt() function. The encrypt function then calls all the helper methods to perform various rounds of encryption and stops after the 10^{th} iteration. This data is the cipher text,

which is returned on the GUI application. Also, the time required to perform the encryption function is also calculated and displayed to the user.

The decryption of the text is also performed on a similar manner and the plain text is displayed to the user.

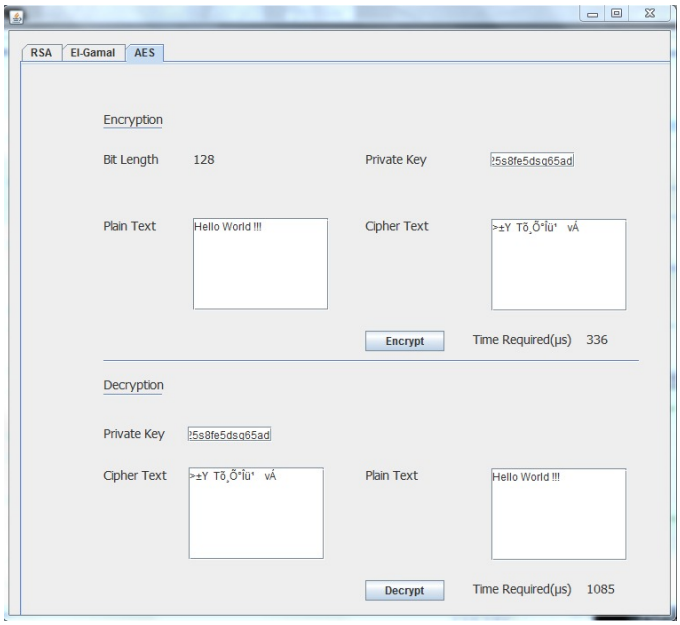

Fig. 6. A snapshot for AES GUI implementation

V. Conclusion

The times required for encryption and decryption using each of the cryptosystems has been tracked. For RSA and ElGamal and AES the decryption times are consistently higher that the encryption times. This can be explained from the fact the message for encryption is always an integer less that 1024 (UTF-8 charset), hence computing its exponent modulo another integer will take lesser time than making a similar computation on a much larger integer (encrypted message can be as large as the modulo values). An interesting point to note is that AES is much faster than the other two, as the key is only 128 bits and the computations do not involve complicated modular arithmetic.

References

[1] R. L. Rivest, A. Shamir and L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key

[2] Taher ElGamal (1985). "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms" *(PDF). IEEE Transactions on Information Theory. 31 (4): 469–472.* doi:10.1109/TIT.1985.1057074. (conference version appeared in CRYPTO'84, pp. 10–18)

[3] https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf