iJET EXPLORE International Journal of Engineering & Technology
CENTRE OF PROFESSIONAL RESEARCH PUBLICATIONS
ISSN 2049-3444

International Journal of Engineering and Technology Volume 2 No. 9, September, 2012

Comparative Analysis of Public-Key Encryption Schemes

Alese, B. K., Philemon E. D., Falaki, S. O.
Department of Computer Science
The Federal University of Technology, Akure; Nigeria

ABSTRACT

The introduction of public-key cryptography by Diffie and Hellman in 1976 was an important watershed in the history of cryptography. The work sparked off interest in the cryptographic research community and soon several public-key schemes were proposed and implemented. The Rivest, Shamir and Adleman (RSA), being the first realisation of this abstract model, is the most widely used public-key scheme today. However, increased processing power and availability of cheaper processing technology occasioned by the exponential growth in digital technology has generated some security concerns, necessitating the review of security parameters for enhanced security. Enhanced processing power requirement does not favour the present class of ubiquitous mobile devices that are characterised by low power consumption, limited memory and bandwidth as they may not be able to run this cryptographic algorithm due to computational burden associated with long key lengths. And since future increase in key lengths looks likely given the current technological developments, Elliptic Curve Cryptography (ECC) has been proposed as an alternative cryptosystem because it satisfies both security requirements and efficiency with shorter key lengths.

This research work focuses on the comparative analysis of RSA Encryption algorithm, ElGamal Elliptic Curve Encryption algorithm and Menezes-Vanstone Elliptic Curve Encryption algorithm. These elliptic curve analogues of ElGamal Encryption scheme were implemented in Java, using classes from the Flexiprovider library of ECC. The RSA algorithm used in the comparison is the Flexiprovider implementation. Performance evaluation on the three algorithms based on the time lapse for their Key generation, encryption and decryption algorithms, and encrypted data size was carried out and compared. The results show that our elliptic curve-based implementations are more superior to the RSA algorithm on all comparative parameters.

Keywords: *Security, Elliptic, Curve, RSA, Crptosystem*

1. INTRODUCTION

Public-key cryptography was originally invented as an elegant solution to the problems associated with Symmetric-key cryptography. Since Its introduction in 1976 by Diffie and Hellman, numerous public-key schemes have been proposed and implemented (Rabah,2005a), each relying on the difficulty of a classical mathematical problem such as Integer Factorization Problem (IFP), Discrete Logarithm Problem (DLP), Elliptic Curve Discrete Logarithm Problem (ECDLP) etc. However, over the years, with the increase in processing power of computers, there has been a reduction in the work factor required to solve IFP and DLP problems (Berta and Mann, 2002). As a result, key sizes grew to more than 1000-bits so as to attain a reasonable level of security. In constrained environments, however, carrying out thousand-bit operations is impractical. Therefore, a matter of growing importance in cryptography is the need for algorithms with low resource requirements that can be deployed on resource-constrained ubiquitous devices. This explains why other public-key methods would be welcomed, Elliptic Curve Cryptosystem (ECC) being a probable candidate.

Elliptic curves are the basis for a relatively new class of Public-key schemes. It is predicted that Elliptic Curve Cryptosystems (ECC) will replace many existing schemes in the near future. However, the complicated mathematical background of ECC results in more sophisticated algorithms, which raises the question whether the required computational power to run the ECC algorithm would be smaller compared to that of RSA. This opens up new vista for comparative studies on the performance of RSA and ECC (Certicom, 2002, 2004).

Literature search has shown that no work exist on the implementation and comparison of Rivest, Shamir and Adleman Encryption Scheme (RSA), Menezes-Vanstone Elliptic Curve Encryption Scheme (MVEC), and Elliptic Curve ElGamal Encryption Scheme (ECEG). And So far no standards have been defined for Elliptic Curve Encryption Scheme (ECES). The only standard specified in most documents is the Elliptic Curve Integrated Encryption Scheme (ECIES), which is a hybrid scheme, combining the best features from asymmetric and symmetric cryptosystems. Hence, it is still a research question whether ciphers based on elliptic curves are ripe enough to be trusted for deployment in commercial products, and probably be adopted as a de facto standard for security on the internet.

The Elliptic Curve ElGamal Encryption protocol and the Menezes-Vanstone Elliptic Curve Encryption protocol consist of three main algorithms: key pair generation, encryption and decryption. In order to reach the goals of implementing these protocols, several functions necessary for their construction were created. Performance evaluations were conducted based on the time lapse for these algorithms and the RSA encryption algorithm. Five test runs were carried out for each protocol on a 100 bytes text data. The results obtained were juxtaposed based on standard parameters such as security level providing key sizes. The size of ciphertext generated by each protocol was noted and compared.

The security levels for the RSA includes: 1024-bit, 2048-bit, 3072-bit, 7680-bit and 15360-bit key sizes, and those of ECEG and MVEC algorithms includes: 160-bit, 224-bit, 256-bit, 384-bit and 521-bit key sizes. These key sizes are taken from the National Institute of Standards and Technology (NIST) guidelines for public key sizes with equivalent security levels (Alese, 2000).

The existing RSA Encryption algorithm benchmarked against these implementations is from the Flexiprovider, a Cryptographic Service Provider.

Entities participating in any of these protocols are required to generate a pair of public and private keys using the appropriate key pair generation algorithm. The public key generated is used for the encryption operation while the private key is used for the decryption operation. Encryption with ECEG is accomplished using the following encryption algorithm:

```
INPUT: Elliptic curve domain parameters (p, E, G, n), public key Q
        Plaintext M.
OUTPUT: ciphertext $(C_1, C_2)$
Represent the message M as a point $P_m$ in $E(F_p)$.
Select $k \in_R [1, n-1]$
Compute $C_1=kG$.
Compute $C_2=P_m + kQ$
Return $(C_1, C_2)$
```

The MVEC encryption algorithm is a variant of this algorithm, employing the use of “masking” instead of “point embedding” as in the case of ECEG. The encryption function in each case is a bijection. Thus the original message can be recovered from the encrypted result by applying its inverse transformation (decryption algorithm) with the appropriate trapdoor information.

A software version of the ElGamal Elliptic Curve Encryption protocol and that of Menezes-Vanstone Elliptic Curve Encryption protocol were implemented. A Comparison between these protocols and the RSA protocol was established. The implementation environment for the chosen algorithms is Java JDK 6 update 19, and the platform for our experiment is Windows Vista Home Basic, running on Intel Pentium Dual Core 1.6GHZ processor and 512MB of RAM (Brown et al. 2001)

2. IMPLEMENTATION AND COMPARATIVE ANALYSIS

2.1 Implementing Elliptic Curve Systems

Prior to the implementation of any elliptic curve systems, several choices have to be made concerning the underlying finite field, elliptic curve, and cryptographic protocols. More elaborately, these choices include the underlying finite field, representation for the finite field elements, algorithms for performing finite field arithmetic, choice of an appropriate elliptic curve, representation for the elliptic curve points, algorithms for performing elliptic curve arithmetic (windows methods in affine or projective coordinates), choice of elliptic curve cryptographic protocol and algorithms for performing protocol arithmetic (Brown et al., 2001). Usually, these selections are influenced by security considerations, application platform, constraints of the particular computing environment, and constraints of the particular communications environment. Hence, it is difficult to decide on a single “best” set of choices. At best what is regarded as an optimal choice represents a compromise between efficiency and security. On the whole, care should be taken to ensure that the set of choices represent the nexus of the selection criteria.

2.1.1 Choice of the Underlying Finite Field

For any implementation of elliptic curve cryptosystems the choice of the underlying finite field is crucial. Almost always the choice is between $GF(p)$ or $GF(2^m)$ for some prime p or some integers m respectively. The $GF(p)$ is the choice for our implementation for the simplicity of its arithmetic which is implemented in terms of integers modulo p (Diffie and Hellman, 1976).

2.1.2 Choice of Appropriate Elliptic Curve

The choice of appropriate elliptic curve to use is one of the most crucial steps in developing an elliptic curve cryptosystem. Some elliptic curves are susceptible to attacks which makes them no more secure than existing systems today. According to (Rabah, 2005a) the most important qualities to look for in a curve includes:

a. The curve has a large order #E(GF(p)).
b. The curve is not susceptible to the MOV attack (super-singular curves)

c. The curve order is divisible by a large prime factor.

d. The large prime factor does not satisfy the divisibility property: $P_c(\mathbb{F}_p)/2^{ni}-1$, for small i.

Two types of curves exist, namely

a. Pseudo-random curves whose coefficients are generated from the output of a seeded cryptographic hash.

b. Special curves whose coefficients and underlying field have been selected to optimize the efficiency of the elliptic curve operations.

The security of any elliptic curve cryptosystems depends primarily on its order. Therefore, to make an ECC secure, we must first find curves which have an order satisfying the following requirements:

- The order of the curve must be a large prime number.
- The curve must be immune to special attacks.

However, since not every elliptic curve offers strong security properties, and for some curves the ECDLP may be solved efficiently, and since a poor choice of the curve compromises security, standards organization like NIST and SECG published a set of recommended curves with well understood security properties. These curves have been recommended for use so as to facilitate interoperability between different implementations of a security protocol. Thus, for our implementation, the following curves from NIST and SECG were adopted (Certicom Corp., 2000).

a. Prime 192v1 and Prime 256v1 from NIST.

b. Secp160r1, Secp224r1, Secp384r1 and Secp521r1 from SECG.

2.1.3 Choice of Elliptic Curve Protocol

Several elliptic curve protocols are in use today. Some have been standardized and packaged in a user-friendly way for developers to include in their applications. For instance, Legion of Bouncycastle and Flexiprovider have implemented and included the following elliptic curve protocols in their Java Cryptographic Extension (JCE): ECDSA, ECIES, ECDH and ECNR. There are no standards defined for the elliptic curve encryption scheme, and no implementations for the pure scheme exist for developers to use straight out-of-box. Hence, in this project we implemented the elliptic curve encryption protocol. In particular, we implemented the elliptic curve variants of ElGamal encryption algorithm, namely

a. Elliptic Curve ElGamal Encryption Algorithm (ECEG), and

b. Menezes- vanstone elliptic curve encryption algorithm (MVEC).

c. Each scheme consists of three main algorithms: key pair generation, encryption and decryption algorithms. We implemented these algorithms in java, using classes from the Flexiprovider besides those we created (Rabah, 2005b, 2005c).

3. STRUCTURE OF THE IMPLEMENTATION

The implementation has been divided into two separate packages: ECEG and MVEC. The ECEG package contains all the classes needed to implement the Elliptic Curve ElGamal Encryption protocol, and the MVEC package contains classes needed to implement the Menezes-Vanstone Elliptic Curve Encryption protocol. The classes in each package are shown in figure 1.

As can be seen from figure 1, the only difference between the ECEG package and the MVEC package are the classes: GFElement, PointGFP, EllipticCurve, EllipticCurveGFP, and IntegerFunctions. This is because the ECEG package implements the point embedding algorithm, which requires the above classes to facilitate the process. The MVEC which uses masking instead of point embedding requires the “Point class” to accomplish its task (Rabah, 2006).

The common classes and those listed above are imports from the Flexiprovider package: “de.flexiprovider.common.maths”. Some of these classes, like ScalarMult and FlexiBigInt, were heavily overloaded in our implementations.

The ECEG package contains all the classes used in the implementation of ElGamal Elliptic Curve Encryption algorithm. It consists of four classes which we constructed beside those we incorporated into our work from other sources. The classes we constructed are: ElGamalECCipher, ElGamalECKeyPair, ElGamalECKeyPairGenerator and ElGamalMain. The structure of these classes is as shown in figure 2 below.

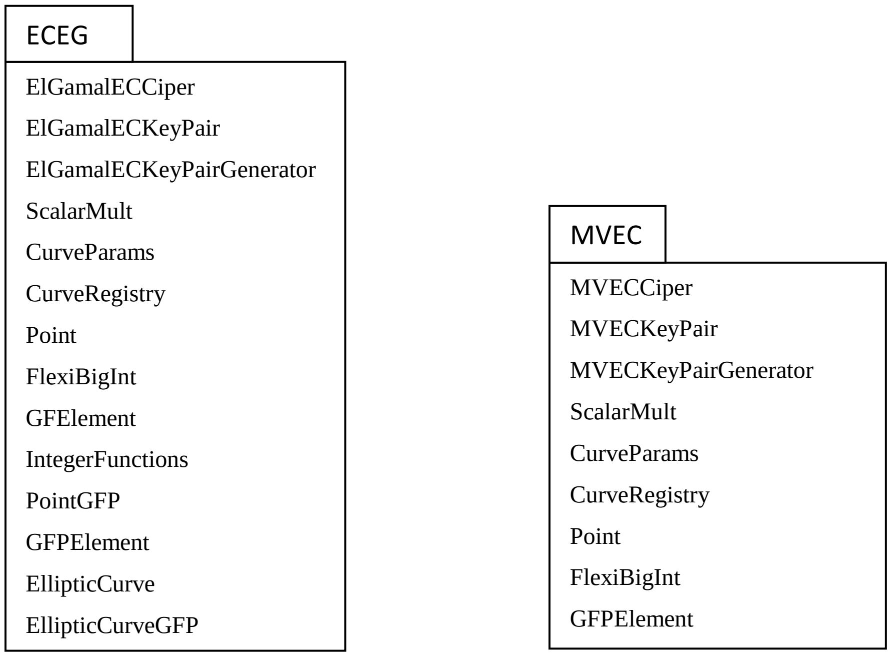


Figure 1: Structure of Implementation

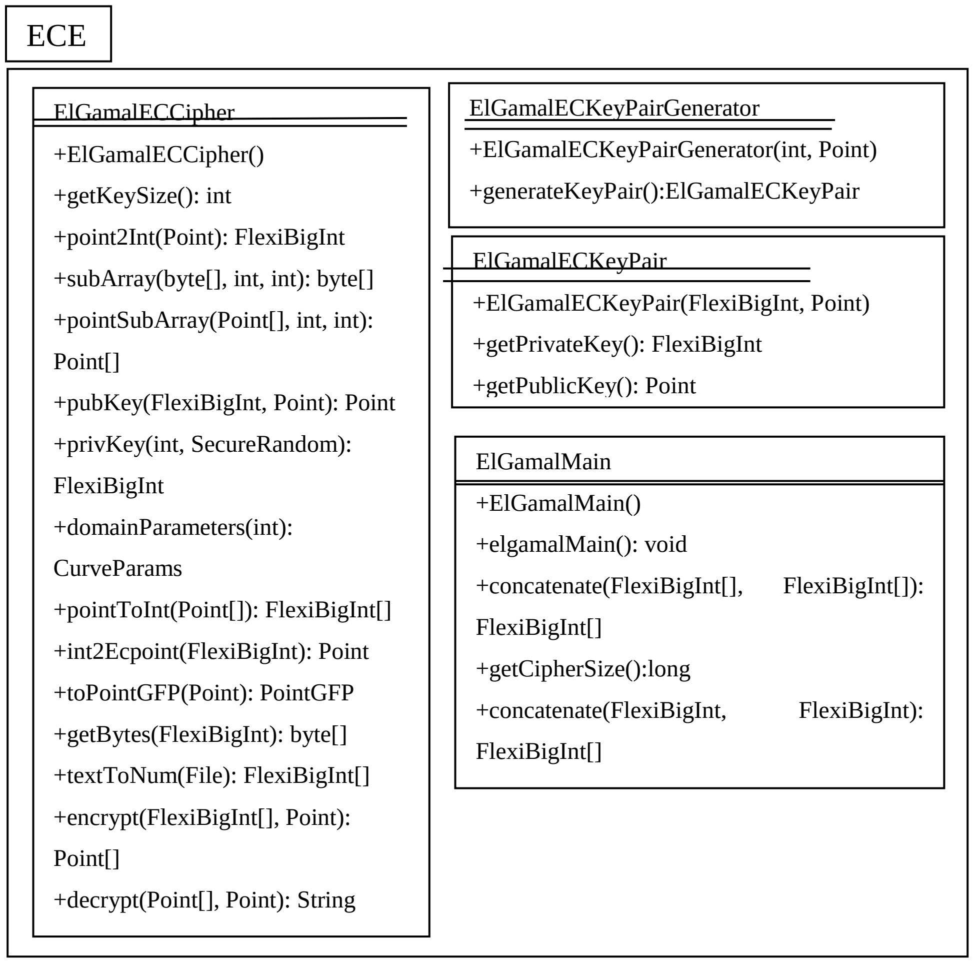

ECE
ElGamalECCipher
+ElGamalECCipher()
+getKeySize(): int
+point2Int(Point): FlexiBigInt
+subArray(byte[], int, int): byte[]
+pointSubArray(Point[], int, int): Point[]
+pubKey(FlexiBigInt, Point): Point
+privKey(int, SecureRandom): FlexiBigInt
+domainParameters(int): CurveParams
+pointToInt(Point[]): FlexiBigInt[]
+int2Ecpoint(FlexiBigInt): Point
+toPointGFP(Point): PointGFP
+getBytes(FlexiBigInt): byte[]
+textToNum(File): FlexiBigInt[]
+encrypt(FlexiBigInt[], Point): Point[]
+decrypt(Point[], Point): String
ElGamalECKeyPairGenerator
+ElGamalECKeyPairGenerator(int, Point)
+generateKeyPair():ElGamalECKeyPair
ElGamalECKeyPair
+ElGamalECKeyPair(FlexiBigInt, Point)
+getPrivateKey(): FlexiBigInt
+getPublicKey(): Point
ElGamalMain
+ElGamalMain()
+elgamalMain(): void
+concatenate(FlexiBigInt[], FlexiBigInt[]): FlexiBigInt[]
+getCipherSize():long
+concatenate(FlexiBigInt, FlexiBigInt): FlexiBigInt[]

Figure 2: Structure of ElGamal Elliptic Curve Encryption Implementation

The MVEC package is the location for all the classes we use in the implementation of Menezes-Vanstone Elliptic Curve Encryption algorithm. The package consists of MVECCipher, MVECKeyPair, MVECKeyPairGenerator and MVECCipherMain classes, which we constructed besides the ones we use from the Flexiprovider library. The structure of these classes is as shown in figure 3 below. Next we give the description of those classes that are common to the two packages.

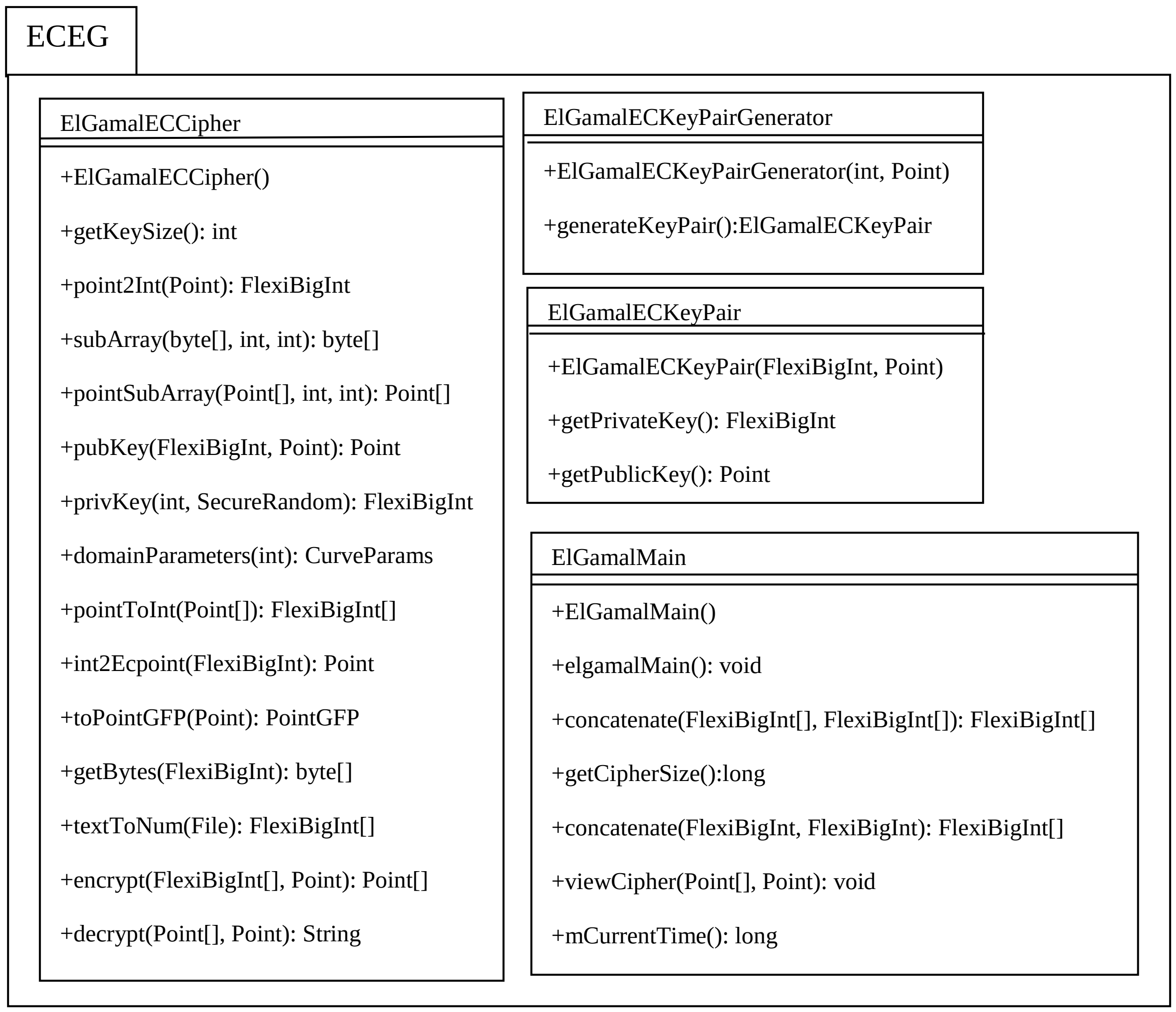

ECEG
ElGamalECCipher
+ElGamalECCipher()
+getKeySize(): int
+point2Int(Point): FlexiBigInt
+subArray(byte[], int, int): byte[]
+pointSubArray(Point[], int, int): Point[]
+pubKey(FlexiBigInt, Point): Point
+privKey(int, SecureRandom): FlexiBigInt
+domainParameters(int): CurveParams
+pointToInt(Point[]): FlexiBigInt[]
+int2Ecpoint(FlexiBigInt): Point
+toPointGFP(Point): PointGFP
+getBytes(FlexiBigInt): byte[]
+textToNum(File): FlexiBigInt[]
+encrypt(FlexiBigInt[], Point): Point[]
+decrypt(Point[], Point): String
ElGamalECKeyPairGenerator
+ElGamalECKeyPairGenerator(int, Point)
+generateKeyPair():ElGamalECKeyPair
ElGamalECKeyPair
+ElGamalECKeyPair(FlexiBigInt, Point)
+getPrivateKey(): FlexiBigInt
+getPublicKey(): Point
ElGamalMain
+ElGamalMain()
+elgamalMain(): void
+concatenate(FlexiBigInt[], FlexiBigInt[]): FlexiBigInt[]
+getCipherSize():long
+concatenate(FlexiBigInt, FlexiBigInt): FlexiBigInt[]
+viewCipher(Point[], Point): void
+mCurrentTime(): long

Figure 3: Structure of Menezes-Vanstone Elliptic Curve Encryption Implementation

FlexiBigInt

This is a wrapper class for Sun java "bigInt" class. It includes methods for addition, subtraction, multiplication and division of large numbers. It also contains methods for doing modular arithmetic such as modular exponentiation (modPow), $g^x mod\ n$, and modular inversion (modInverse),

$$ax \equiv 1\ mod\ n. \tag{1}$$

The FlexiBigInt class is based on mutable arrays and thus there is no need for dynamic memory allocation. The public methods used from this class are shown in figure 4.

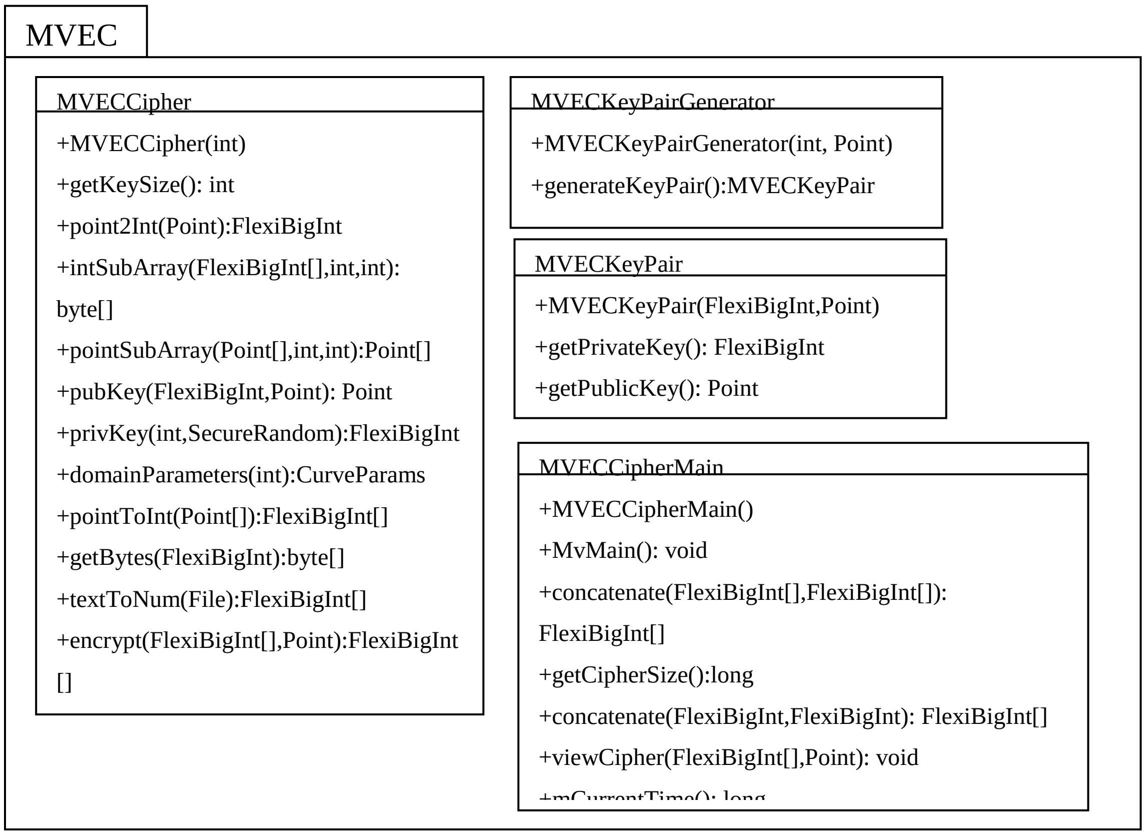

MVEC
MVECCipher
+MVECCipher(int)
+getKeySize(): int
+point2Int(Point):FlexiBigInt
+intSubArray(FlexiBigInt[],int,int): byte[]
+pointSubArray(Point[],int,int):Point[]
+pubKey(FlexiBigInt,Point): Point
+privKey(int,SecureRandom):FlexiBigInt
+domainParameters(int):CurveParams
+pointToInt(Point[]):FlexiBigInt[]
+getBytes(FlexiBigInt):byte[]
+textToNum(File):FlexiBigInt[]
+encrypt(FlexiBigInt[],Point):FlexiBigInt []
MVECKeyPairGenerator
+MVECKeyPairGenerator(int, Point)
+generateKeyPair():MVECKeyPair
MVECKeyPair
+MVECKeyPair(FlexiBigInt,Point)
+getPrivateKey(): FlexiBigInt
+getPublicKey(): Point
MVECCipherMain
+MVECCipherMain()
+MvMain(): void
+concatenate(FlexiBigInt[],FlexiBigInt[]): FlexiBigInt[]
+getCipherSize():long
+concatenate(FlexiBigInt,FlexiBigInt): FlexiBigInt[]
+viewCipher(FlexiBigInt[],Point): void
+mCurrentTime(): long

Figure 4: class structure of FlexiBigInt class

IntegerFunctions

This class is contained in "de.flexiprovider.common.math" package. It contains number-theory related functions for use with integers represented as int's or FlexiBigInt objects. It includes methods for determining the Jacobi symbol, which we incorporated in our ECEG implementation. The public methods of this class used in our ECEG implementation are shown in figure 4.5.

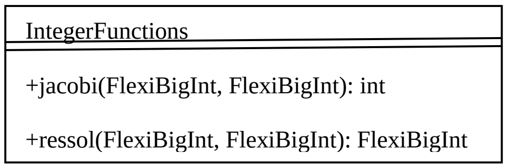

IntegerFunctions
+jacobi(FlexiBigInt, FlexiBigInt): int
+ressol(FlexiBigInt, FlexiBigInt): FlexiBigInt

Figure 5: Class Structure for IntegerFunctions class

ScalarMult

Scalar multiplication is the most dominant operation in elliptic curve cryptography. This class, ScalarMult, implements the scalar multiplication algorithms described in chapter three. This is a class located in the Flexiprovider package: "de.flexiprovider.common.maths". It includes methods for determining the NAF of an integer. The public method used from this class is shown in figure 6.

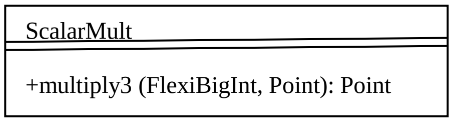


Figure 6: Class Structure for the ScalarMult class

CurveRegistry

This class is located in "de.flexiprovider.ec.parameters" package of the Flexiprovider. It is a container class for some approved EC domain parameters for elliptic curve cryptography. The parameters consist of the chosen elliptic curve to be used, the point order of the curve, and the base point of the chosen curve etc. It supports domain parameters from ANSI X9.62, BrainPool, CDC group, SEC2 and NIST. For our implementations, we utilized domain parameters supported by SEC2 and NIST (see appendix B for list of parameters).

Point

This is an abstract class located in "de.flexiprovider.common.ellipticcurves". It implements points and their arithmetic on elliptic curves over finite prime fields as well as finite binary fields.

PointGFP

This class is a direct subclass of the Point class. It implements points and their arithmetic on elliptic curves over finite prime field. It includes methods for point addition, subtraction and multiplication in both affine and projective representations. This class is one of the focal points of our ECEG implementation since the finite prime field is our chosen underlying field. The public methods used from this class are shown in figure 7.

GFElement

This is an Interface contained in "de.flexiprovider.common.math.finitefields" package. It defines a finite field element and suggests methods for field operations on field elements such as addition, subtraction, multiplication and division (inversion).

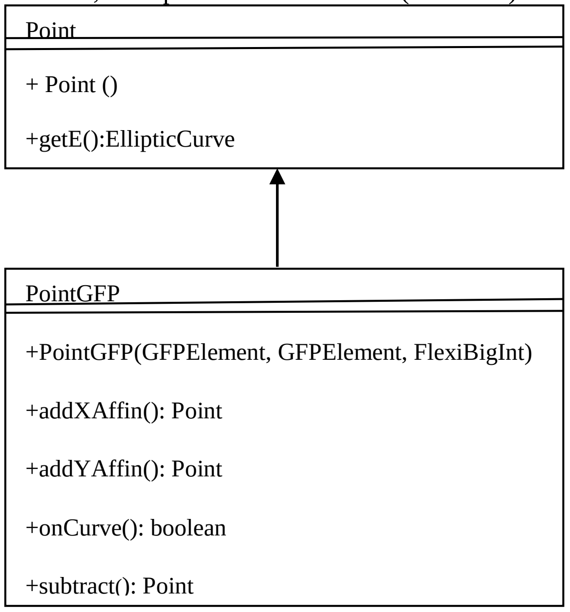


Figure 7: Class Structure for the abstract class Point and PointGFP class

GFPElement

This class is the implementation of the GFElement Interface. It implements an element of the finite prime field, and includes methods for field arithmetic such as addition, subtraction and multiplication etc. The public methods used from this class are shown in figure 4.8.

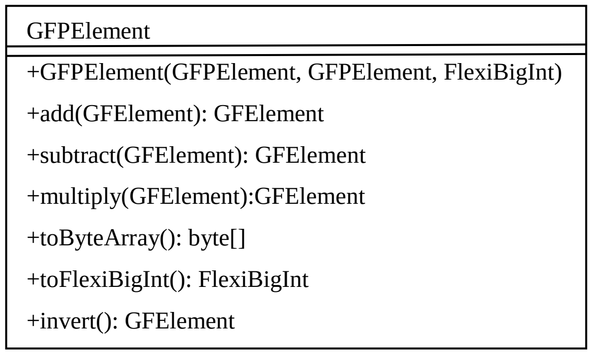

GFPElement
+GFPElement(GFPElement, GFPElement, FlexiBigInt)
+add(GFElement): GFElement
+subtract(GFElement): GFElement
+multiply(GFElement):GFElement
+toByteArray(): byte[]
+toFlexiBigInt(): FlexiBigInt
+invert(): GFElement

Figure 8: Class Structure for GFPElement class

EllipticCurve

This class is the top-interface for elliptic curves over finite fields. It is located in the "de.flexiprovider.common.maths.ellipticcurves" package. It stores the size of the underlying field as an instance of FlexiBigInt, and the curve parameters, a and b, as instances of GFElement.

EllipticCurvesGFP

This class implements the EllipticCurve class. It holds elliptic curves over finite prime fields in Weierstrass short form, and stores the field elements, a and b, as instances of FlexiBigInt respectively. The public methods used from this class are shown in figure 9.

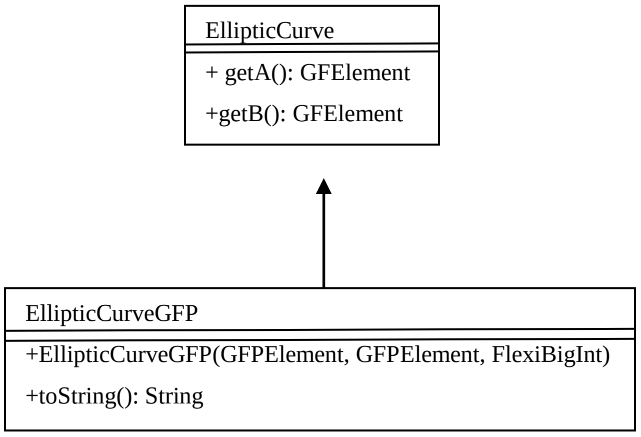


Figure 9: Class Structure for the abstract class EllipticCurve and EllipticCurveGFP class

4. PERFORMANCE COMPARISON OF ECC AND RSA ENCRYPTION SCHEMES

Robshaw and Yin (1997) compared the operational characteristics of RSA and ECC. In their article, they claimed that assuming all necessary parameters and initialization processes have been performed encryption with ECC will be almost 8 times longer than with RSA, and decryption will be almost 6 to 7 times faster. These findings, however, contrasted that of Certicom Corporation who adjudged the most efficient implementation of ECC 10 times faster than comparable RSA systems. Next, we look at the theory behind RSA public key encryption to enable us perform our comparison (Stallings, 2003; Trappe and Washington, 2002)

4.1 Theory of RSA Cryptosystem

The RSA is the most widely used cryptosystem today. Unfortunately, encrypting a message, m, involves exponentiation, $c = m^e mod\, n$, a mathematical procedure which requires a lot of computations, making it impossible to achieve the speeds of private key systems such as DES, a phenomenon that is true for all public key systems (Hankerson et al., 2004).

To set up a RSA cryptosystem, a user (say Alice) picks two large primes p and q and computes their product, $n = pq$. The group used is the multiplicative group $(G = Z^*)$ of units in the integer modulo n. It is well known that the order of G is $\varphi = (p-1)(q-1)$, where φ denotes the Euler phi function. Clearly, Alice's public key is the pair of integers {n, e} and her private key is d.

4.2 RSA Key Generation

An RSA key pair can be generated using Algorithm 4.1. The public key consists of a pair of integers (n, e) where the *RSA modulus n* is a product of two randomly generated (and secret) primes p and q of the same bit length. The *encryption exponent e* is an integer satisfying $1 < e < \varphi$ and gcd $(e, \varphi) = 1$ where $\varphi = (p-1)(q-1)$. The private key d, also called the *decryption exponent*, is the integer satisfying $1 < d < \varphi$ and $ed \equiv 1 \pmod{\varphi}$. It has been proven that the problem of determining the private key d from the public key (n, e) is computationally equivalent to the problem of determining the factors p and q of n (Menezes et al., 1991); the latter is the *integer factorization problem* (IFP).

Algorithm1 RSA key pair generation

```
INPUT: Security parameter $l$.
OUTPUT: RSA public key $(n, e)$ and private key $d$.
1. Randomly select two primes $p$ and $q$ of the same bitlength $l/2$.
2. Compute $n = pq$ and $\varphi = (p-1)(q-1)$.
3. Select an arbitrary integer $e$ with $1 < e < \varphi$ and gcd $(e, \varphi) = 1$.
4. Compute the integer $d$ satisfying $1 < d < \varphi$ and $ed \equiv 1 \pmod{\varphi}$.
5. Return $(n, e, d)$.
```

4.3 RSA Encryption/ Decryption Scheme

RSA encryption schemes use the fact that $m^{ed} \equiv m \pmod{n}$ for all integers m. The encryption and decryption procedures for the (basic) RSA public-key encryption scheme are presented as Algorithms 4.2 and 4.3. Decryption works because $c^d \equiv (m^e)^d \equiv m \pmod{n}$, as derived from expression. The security relies on the difficulty of computing the plaintext m from the ciphertext $c = m^e$ mod n and the public parameters n and e. This is the problem of finding e-th roots modulo n and is assumed (but has not been proven) to be as difficult as the integer factorization problem (Weil, 1998).

Algorithm 2: Basic RSA encryption

```
INPUT: RSA public key $(n, e)$, plaintext $m \in [0, n-1]$.
OUTPUT: Ciphertext $c$.
1. Compute $c = m^e$ mod $n$.
2. Return($c$).
```

Algorithm 3: Basic RSA decryption

```
INPUT: RSA public key $(n, e)$, RSA private key $d$, ciphertext $c$.
OUTPUT: Plaintext $m$.
1. Compute $m = c^d$ mod $n$.
2. Return ($m$).
```

5. RUN-TIME COMPARISON BETWEEN ECC AND RSA

To test and compare the performance characteristics of the RSA and ECC encryption algorithms, we independently tested each of the following three main components for timings: key generation, encryption and decryption. Timings are not absolute so each operation for every test parameter was run 20 times in order to reach satisfactory level of confidence interval. A 99.9% confidence interval was calculated from the test results using the student T-distribution. We also measured the size of the data files used to store the encrypted results.

The parameters of the operations are:

a. the size of the applied key

b. the size and content of the input data

Tests were performed on Intel Pentium dual core 1.6GHZ machine with 512MB of RAM. The message used for encryption is the 100 byte text:" *ECDLP is believed to be harder than both the Integer Factorization and Discrete Logarithm Problems*". The operating system is Windows Vista Home Basic. A selected output results for encryption with the implemented algorithms using different keys on the same input text are as presented in appendix A.

Estimates are given for parameter sizes providing comparable levels of security for RSA and EC systems. The parameter sizes, also called key sizes, that provide equivalent security levels for RSA and EC systems are as listed in table .1.

Table 1: Comparable key sizes between ECC and RSA

ECC	RSA
160	1024
224	2048
256	3072
384	7680
512	15360

These five specific security levels were selected because they represent the amount of work required to perform an exhaustive key search on the symmetric key encryption schemes: SKIPJACK, TRIPLE-DES, AES-Small, AES-Medium, and AES-Large respectively.

5.1 Test Results

This section contains the test results of our experiment. These results are made up of the lower and upper limits of 99.9% confidence interval calculated using the T-distribution (see tables 2-4).

Table 2: Test Results for RSA Encryption Scheme

RSA Key	Key Generation Time (milliseconds)	Encryption Time (milliseconds)	Decryption Time (milliseconds)
1024	1312.7 ± 190.8	166.9 ± 46.3	15.7 ± 0.4
2048	6804.6 ± 2540.6	290.2 ± 29.8	122.4 ± 9.1
3072	32108.0 ± 18947.7	310.5 ± 75.5	293.2 ± 71.8
7680	322843.0 ± 233809.0	352.1 ± 154.1	2932.8 ± 44.7
15360	N/A	N/A	N/A

Table 3: Test Results for Elliptic Curve ElGamal Encryption Scheme

Elliptic Curve	Key Generation Time (milliseconds)	Encryption Time (milliseconds)	Decryption Time (milliseconds)
P-160	198.6 ± 12.5	17.9 ± 4.9	15.7 ± 0.1
P-224	208.3 ± 13.4	95.9 ± 6.8	18.7 ± 5.5
P-256	243.5 ± 22.2	35.1 ± 6.1	21.1 ± 6.8
P-384	294.0 ± 26.5	74.9 ± 7.1	47.7 ± 3.2
P-521	447.8 ± 90.9	138.2 ± 4.9	109.9 ± 0.3

Table 4: Test Results for Menezes-Vanstone Elliptic Curve Encryption Scheme

Elliptic Curve	Key Generation Time (milliseconds)	Encryption Time (milliseconds)	Decryption Time (milliseconds)
P-160	198.6 ± 12.5	15.7 ± 0.4	15.5 ± 0.4
P-224	208.3 ± 13.4	18.8 ± 10.0	17.2 ± 8.0
P-256	243.5 ± 22.2	25.0 ± 5.6	20.3 ± 6.4
P-384	294.0 ± 26.5	50.1 ± 5.4	47.6 ± 3.0
P-521	447.8 ± 90.9	109.9 ± 2.9	108.5 ± 5.3

5.2 Analysis of Test Results

Key Generation Time

In RSA, the *generation of the prime numbers* is a crucial sub-process, which requires generating random numbers and testing for primality, a highly probabilistic procedure. Consequently, the times of execution for RSA key generation are not always the same even for the same key length; occasionally it can be very long. Nevertheless, this time depends on the key size, but does not depend on the size of the input data. The measured values for the Flexiprovider RSA implementation range from 1312.7ms – 322843ms (figure 10 below). Typically, for the recommended 1024-bit key, the time is 1312.7±190.8 milliseconds.

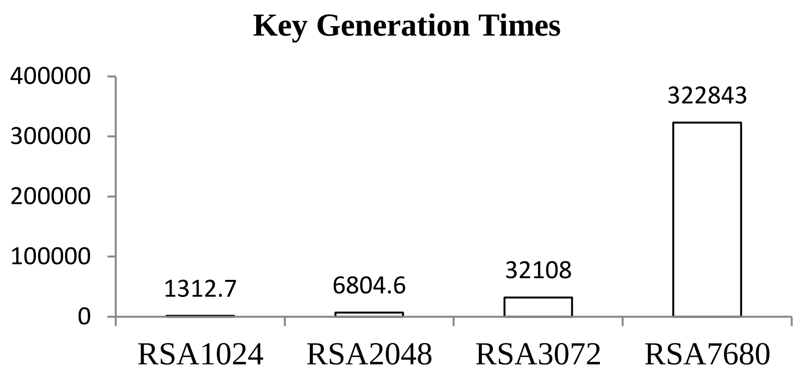


Figure 10: Key Generation Time (milliseconds)

In the case of our ECC implementations, generation of new common parameters is difficult and often results in curves that are susceptible to certain specialized attacks. Hence, for real life applications, certain curves with reliable parameters have been recommended. They are believed to enhance interoperability between disparate systems, a desirable quality for the applications programmer. The time for key generation in ECC depends on the key size, the type of ECC and the usage of pre-computed tables, an efficiency factor included in some implementation for scalar point multiplication, the most dominant operation in elliptic curve field operation. Measured values for the ECP (in this case ECEG and MVEC) key pair generation range from 198.6ms to 447.8ms (figure 11 below). Typically, that is, for the recommended 160-bit key, this time is 198.6 ± 12.5ms.

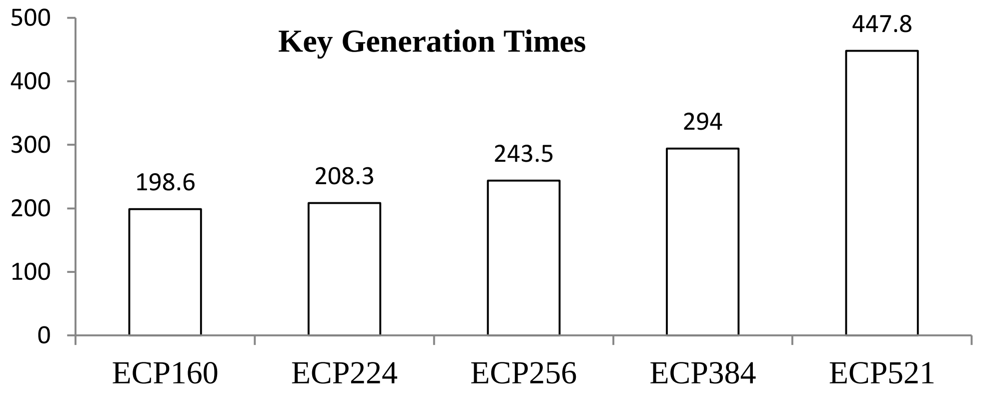


Figure.11: Key Generation Time (milliseconds)

Our comparison revealed that key pair generation for the ECC systems outperforms RSA at all key lengths, and is especially apparent as the key length increases. ECC can create the private/public key pair in superior speed to RSA comparable key lengths. This is, however, so because ECC does not have to devote resources to the computationally intensive generation of prime numbers. ECC key pair generation time grows linearly with key sizes, while that of RSA grows exponentially. This conclusion is obvious from figures 10 and 11 respectively.

Encryption Time

The time lapse for RSA encryption depends on the key size, but does not depend on the size and content of the data to be encrypted. The value of the exponent used by our provider is e=65537, which is recommended for the

1024-bit key in commercial use today. The measured times for 100 bytes of data range from 166.9ms to 352.1ms.

Encryption time with ECC, however, relies on the key size, the algorithm type, and the size of the data to be encrypted. But for the increased storage requirement, the use of pre-computed tables is highly recommended as this speed up the elliptic curve operation more than twice. Encryption times with MVEC appear to be superior when compared to ECEG. Using 100 bytes of input data, the measured values for MVEC range from 15.7ms to 109.9ms, while that of ECEG ranges from 17.9ms to 138.2ms (figure 12 below).

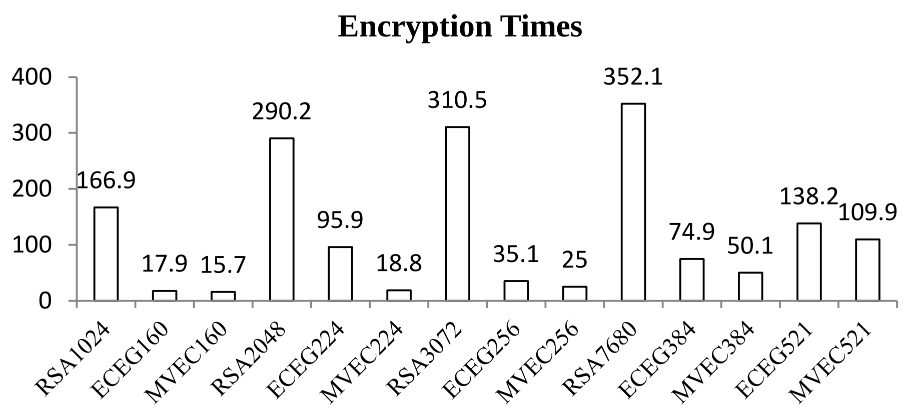


Figure 12: Encryption Times (in milliseconds)

It is obvious from figure 12 that encryption with ECEG and MVEC are superior to that with RSA. The fastest encryption, however, is our MVEC implementation. This is true for all values of the key lengths considered.

Decryption Time

Decryption time for the RSA depends on the key size, but does not depend on the size and content of the input data. It is worth noting here that the Chinese Remainder Theorem (CRT) was used to facilitate the decryption operation. Measured values for 100 bytes of input data ranges from 15.7ms – 2932.8ms (figure 13 below). The RSA15360 is not considered in this performance evaluation because it is not practical.

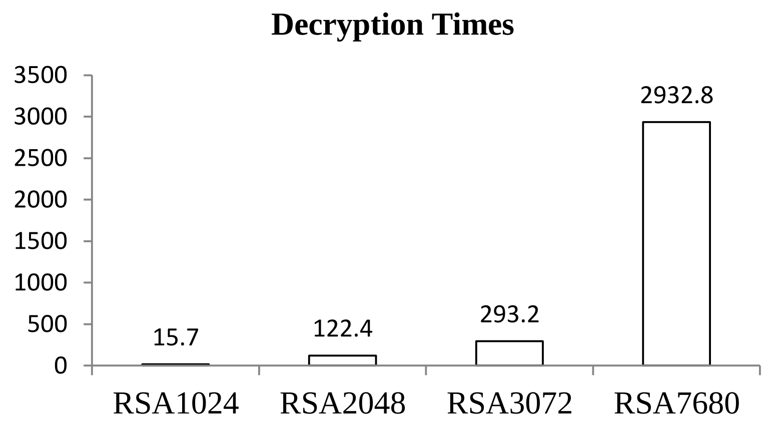


Figure 13: RSA Decryption Times (milliseconds)

Decryption time for ECC, on the other hand, depends on the key size, the algorithm type, the size of the input data, but does not depend on the usage of pre-computed tables. Decryption times with MVEC appear to be superior compared to ECEG. Using 100 bytes of data size, the measured values for MVEC range from 15.5ms to 108.5ms, while that of ECEG ranges from 15.7ms to 109.9ms (figure 14 below).

Our analysis reveals that all the three algorithms perform at equal rate for the lowest security level. But as the key sizes increase ECEG and MVEC fared better at almost equal rate. However, at the 521-bit key length, MVEC decryption time was superior to that of ECEG.

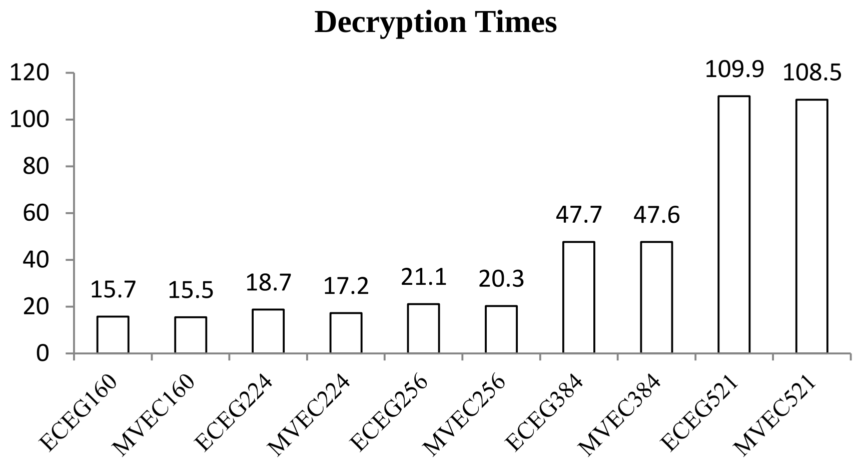


Figure 14: ECP (ECEG and MVEC) Decryption Times (in milliseconds)

6. SIZE OF ENCRYPTED DATA FILES

The size of the encrypted data for RSA depends on the size of the key and the size of input data. Previous research efforts in this area revealed that the size of encrypted data file is equal to the size of the RSA modulus (Soram and Khomdram, 2009). The results we obtained with the Flexiprovider JCE were consistent with this submission. In this RSA implementation, the size of the encrypted data file is always equal to the modulus in all test cases, a phenomenon that results in considerable bandwidth savings. However, there is a huge limitation placed on size of the input data to be encrypted. The data to be encrypted is required to be smaller than the modulus in each case.

On the other hand, for the Elliptic Curve ElGamal Encryption scheme (ECEG) the size of encrypted data depends on the key size and the input data size. Also, the maximum size of data that can be encrypted in one step depends on the key size, a limitation that makes it practically impossible for considering ECEG for large data size encryption. This scenario also holds for Menezes-Vanstone Elliptic Curve Encryption scheme (MVEC), but in this case the size of the encrypted data file appears to be far smaller in size compared to ECEG for all key lengths. This implies MVEC is far more superior to ECEG and RSA on all counts, and therefore suitable for implementation on various platforms, even the constrained environments.

Our Comparison further revealed that MVEC offers considerable bandwidth savings in that the encrypted data size is larger than the input data size by a factor of 0.5 and this scale linearly as the key size increases. RSA on the other hand, scales poorly as the key size increases. For instance, the encrypted data size for RSA-7680-bit and 15360-bit keys is 960 bytes and 1920 bytes respectively compared to MVEC-384 bits and MVEC-521 bits keys with encrypted data size of 271 bytes and 321 bytes respectively (see table 5 and figure 15 below).

Table 5: Size of encrypted data when making comparison between RSA and ECC

Common key sizes (bits)	Size of Data to be encrypted (byte)	Encrypted data size with RSA (byte)	Encrypted data size with ECEG (byte)	Encrypted data size with MVEC (byte)
ECC160=RSA1024	100	128	288	145
ECC224=RSA2048	100	256	342	170
ECC256=RSA3072	100	384	387	196
ECC384=RSA7680	100	960	485	243
ECC521=RSA15360	100	1920	525	263

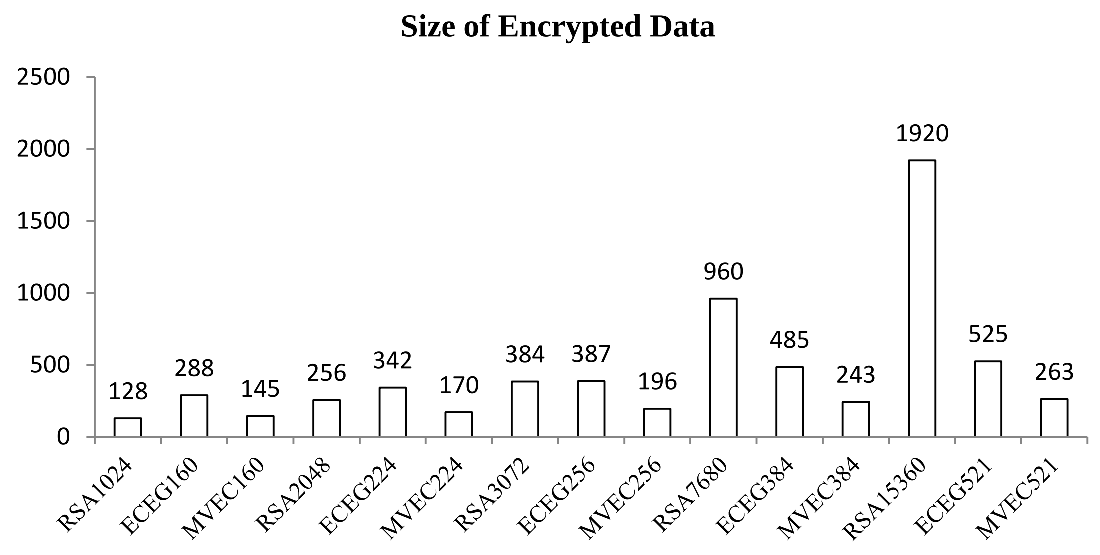


Figure 15: Size of Encrypted Data (bytes)

7. CONCLUSION

Public-key encryption can be used to eliminate problems involved with conventional encryption. However, it has not managed to be as widely accepted as conventional encryption because it introduces a lot of overheads. Therefore, it is very important to find ways to reduce the overheads yet not sacrificing on other aspects of security so that the desirability in public-key can be exploited.

We have described ECC, which is a promising candidate for the next generation public-key cryptosystem. Although ECC's security has not been completely evaluated, it is expected to come into widespread use in various fields in the future.

After comparing the RSA and ECC ciphers, the ECC has proved to involve much less overheads compared to RSA. The ECC has been shown to have many advantages due to its ability to provide the same level of security as RSA yet using shorter keys. However, its disadvantage which may even hide its attractiveness is its lack of maturity, as mathematicians believed that enough research has not yet been done in ECDLP.

Also, we believed that even though both systems are valid, the RSA is better than ECC for now, as it is more reliable because its security can be trusted more. However, the future of ECC looks brighter than that of RSA as today's applications (smart cards, pagers, and cellular telephones etc) cannot afford the overheads introduced by RSA. Finally, both systems can be considered as good given the low success rate associated with attacking them.

REFERENCES

[1] Alese, B.K. (2000). *Vulnerability Analysis of Encryption/Decryption techniques of compute Network security*. Master's Thesis, Federal University of Technology, Akure, Nigeria.

[2] Berta, I.Z., and Z. A. Mann (2002). *Implementing Elliptic Curve Cryptography on PC and Smart Card*, Periodica Polytechnica Ser. El. Eng. Vol 46 NO 1-2, PP 47.

[3] Brown, M., D. L. Hankerson, J. L_opez and A. Menezes (2001). Software implementation of the NIST Elliptic curves over prime fields. In Progress in Cryptology - CT-RSA, D. Naccache, Ed.,vol. 2020 of Lecture Notes in Computer Science, pp. 250-265.

[4] Certicom Corp., (2004). *An elliptic curve cryptography (ecc) primer. White paper*, Certicom.

[5] Certicom Press Release (2002). *Certicom announces Elliptic Curve cryptosystem (ECC) ChallengeWinner*. November 6, 2002. http://www.Certicom.com/about/pr/02/021106_ecc_winner.html

[6] Certicom Corp., (2000), Standards for Efficient Cryptography, SEC 1: Elliptic Curve Cryptography. http://www.secg.org/drafts.htm.

[7] Diffie, W. and M. E. Hellman (1976). New directions in cryptography. IEEE Transactions on Information Theory, Vol. 22, No. 6, pp. 644-654.

[8] Friedman, W.F. (1987). The Index of Coincidence and Its Applications in Cryptography, Riverbank Publication No. 22, Riverbank Labs, 1920. Reprinted by Aegean Park Press.

[9] FIPS 113, National Institute of Standards and Technology (formerly National Bureau of Standards), 1985. FIPS PUB 113: Computer Data Authentication, May 30.

[10] Rabah, K. (2005a). *Theory and implementation of elliptic curve cryptography*. Journal of applied sciences, Vol 5: 604-633.

[11] Rabah, K. (2005b). *Elliptic curve ElGamal Encryption and signature Schemes.* Information technology journal,4(3): 299-306.

[12] Rabah, K. (2005c). *Implementation of Elliptic curve Diffie-Hellman and EC Encryption schemes*. Information technology journal ,4(2): 132-139

[13] Rabah, K. (2006). Implementing Secure RSA Cryptosystem Using Your Own Cryptographic JCE Provider. Journal of Applied sciences, 6(3); 482-510.

[14] Robshaw, M. J. B. and Y. L. Yin (1997). Elliptic Curve Cryptosystems. http://www.rsasecurity.com/rsalabs/ecc/elliptic curve.html.

[15] Stallings, W. (2003). Cryptography and Network Security: Principles and Practice, 3rd, Prentice Hall, New Jersey.

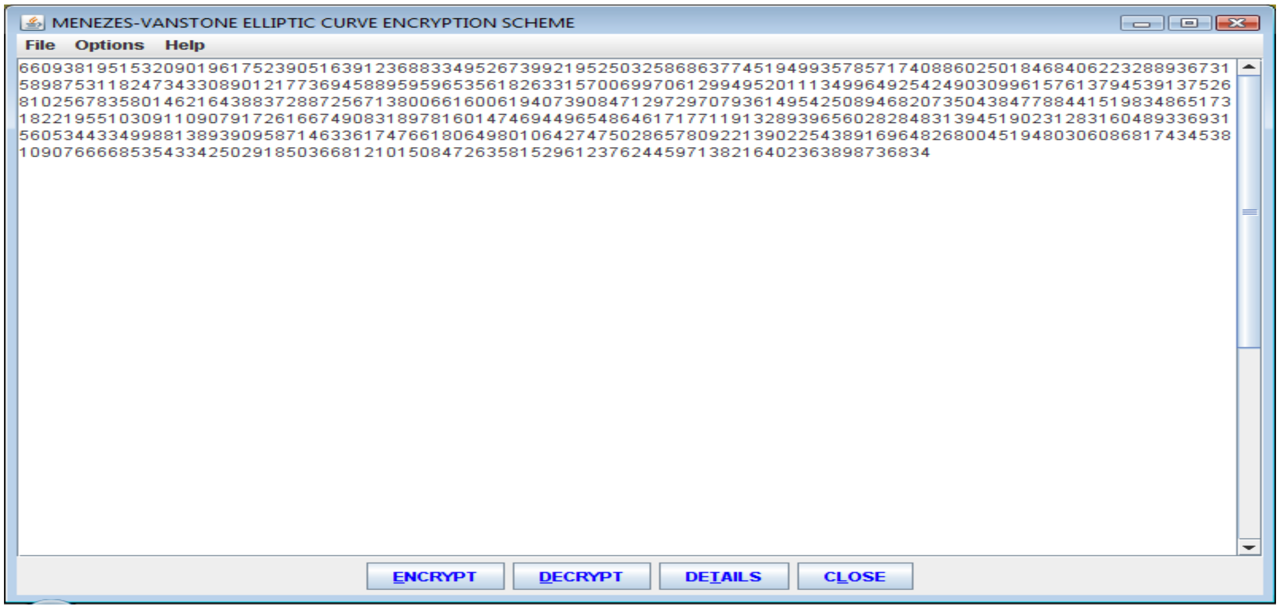
[16] Systronic Inc. Bruce Boyes. *Why use java?*. *www.practicalembeddedjava.com/language/whyusejava.pdf. 8.1*,

[17] Trappe. W. and L. C. Washington. (2002). Introduction to Cryptography with Coding Theory, Prentice Hall, New Jersey.

[18] Weil, N. (1998). U.S. govt.'s encryption standard cracked in record time. Network World. http://www.networkworld.com/news0720des.html

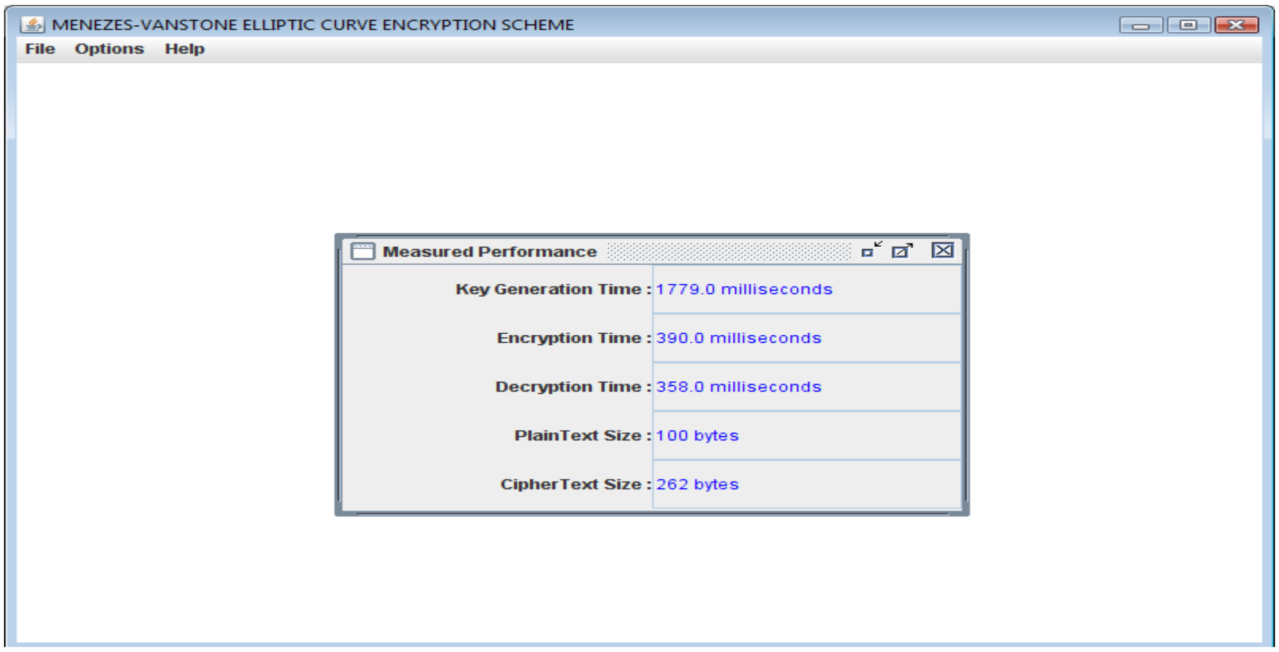
APPENDIX A
Screen Shots of Interfaces showing some Selected Outputs

The screenshot below shows the ciphertext when 521-bits key is used with the MVEC encryption transformation.



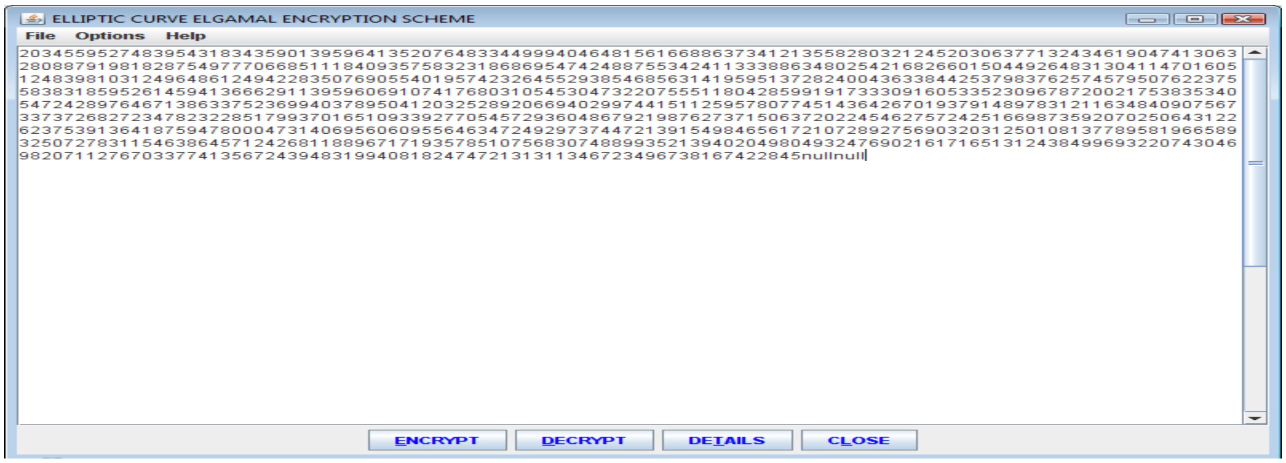
```
66093819515320901961752390516391236883349526739921952503258686377451949935785717408860250184684062232889367 31
589875311824734330890121773694588959596535618263315700699706129949520111349964925424903099615761379453913 7526
810256783580146216438837288725671380066160061940739084712972970793614954250894682073504384778844151983486 5173
182219551030911090791726166749083189781601474694496548646171771191328939656028284831394519023128316048933 6931
560534433499881389390958714633617476618064980106427475028657809221390225438916964826800451948030608681743 4538
1090766668535433425029185036681210150847263581529612376244597138216402363898736834
```

The screenshot below shows the measured performance when 521-bits key is used with the MVEC encryption transformation.



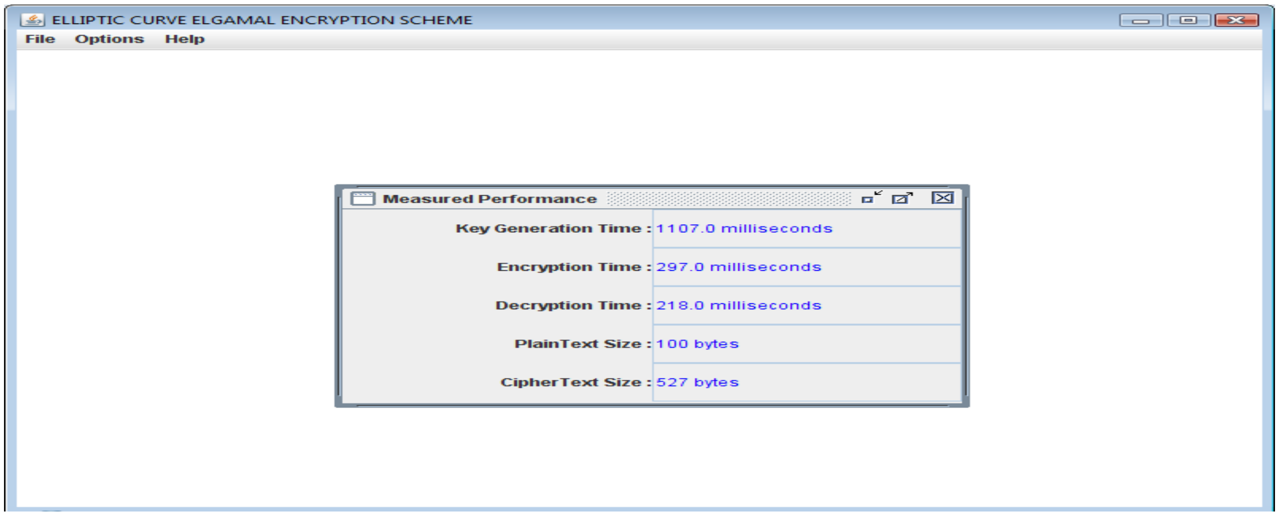
Key Generation Time :	1779.0 milliseconds
Encryption Time :	390.0 milliseconds
Decryption Time :	358.0 milliseconds
PlainText Size :	100 bytes
CipherText Size :	262 bytes

The screenshot below shows the ciphertext when 521-bits key is used with the ECEG encryption transformation.


ELLIPTIC CURVE ELGAMAL ENCRYPTION SCHEME
File Options Help
ENCRYPT DECRYPT DETAILS CLOSE

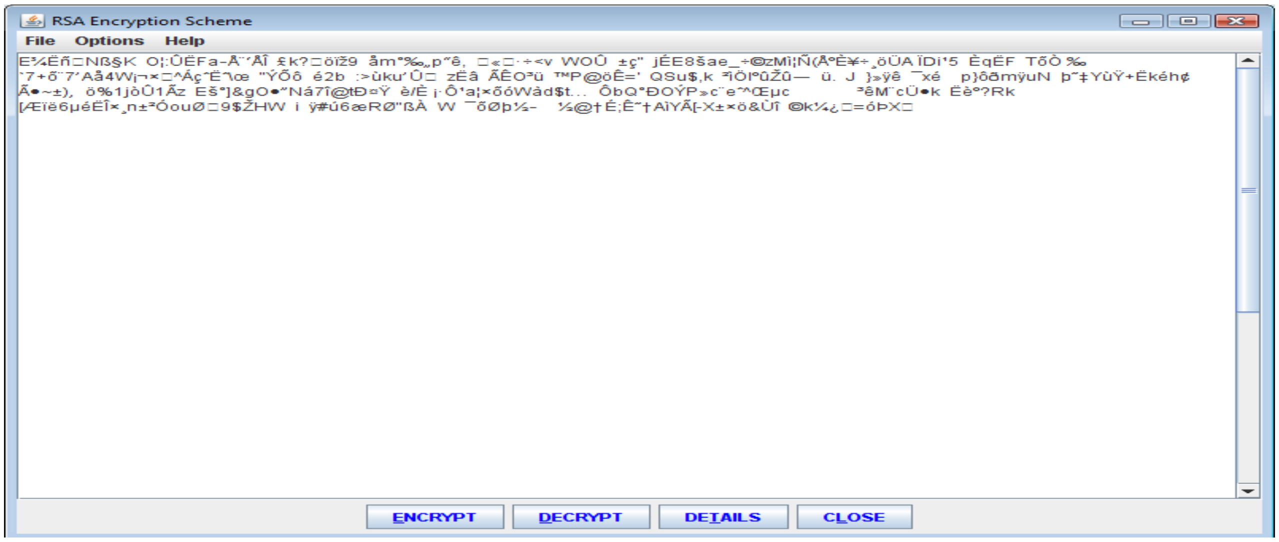
```
2034559527483954318343590139596413520764833449994046481561668863734121355828032124520306377132434619047413063
2808879198182875497770668511184093575832318686954742488755342411333886348025421682660150449264831304114701605
1248398103124964861249422835076905540195742326455293854685631419595137282400436338442537983762574579507622375
5838318595261459413666291139596069107417680310545304732207555118042859919173330916053352309678720021753835340
5472428976467138633752369940378950412032528920669402997441511259578077451436426701937914897831211634840907567
3373726827234782322851799370165109339277054572936048679219876273715063720224546275724251669873592070250643122
6237539136418759478000473140695606095564634724929737447213915498465617210728927569032031250108137789581966589
3250727831154638645712426811889671719357851075683074889935213940204980493247690216171651312438499693220743046
9820711276703377413567243948319940818247472131311346723496738167422845nullnull
```

The screenshot below shows the measured performance when 521-bits key is used with the ECEG encryption transformation.



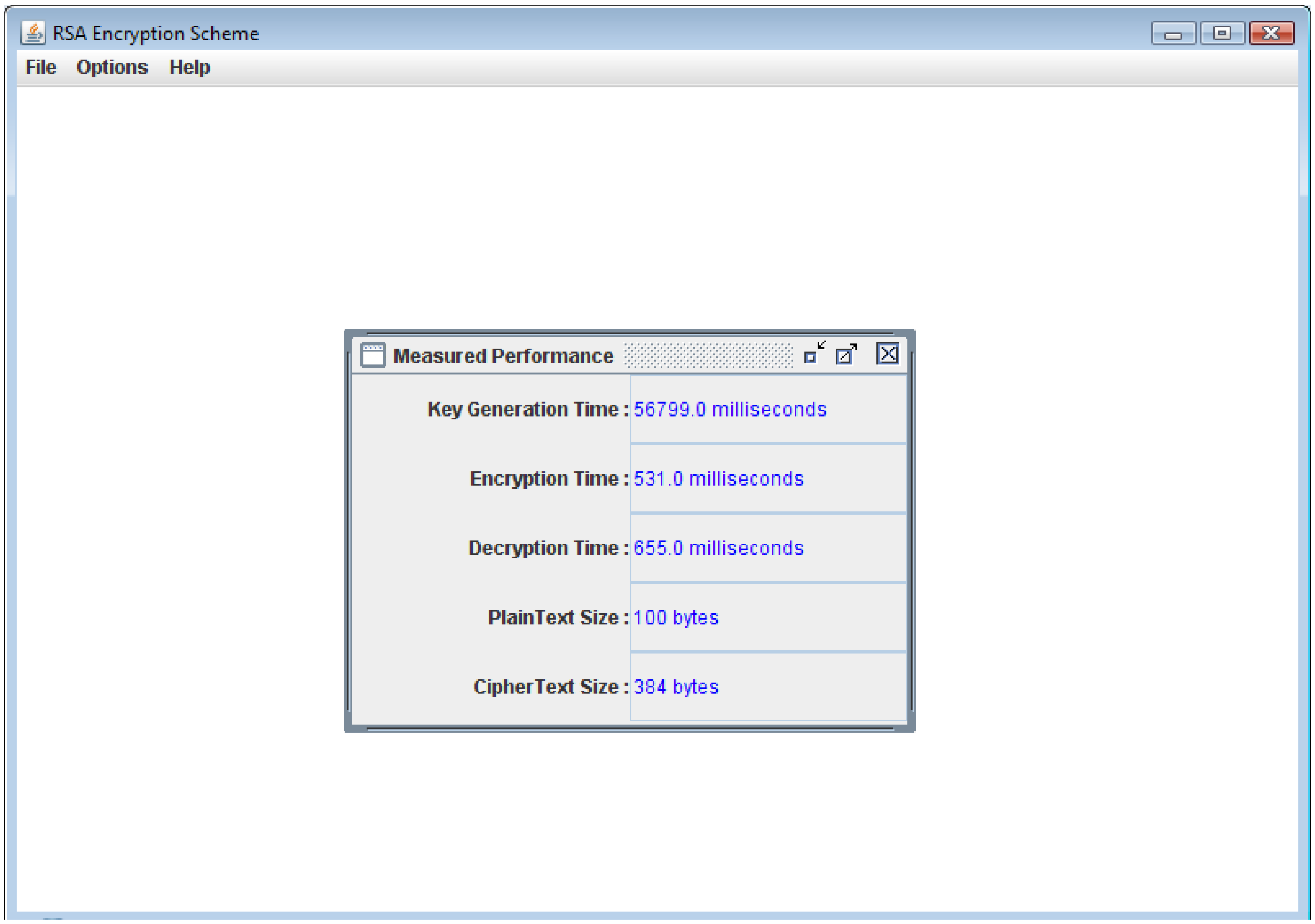
Key Generation Time :	1107.0 milliseconds
Encryption Time :	297.0 milliseconds
Decryption Time :	218.0 milliseconds
PlainText Size :	100 bytes
CipherText Size :	527 bytes

The screenshot below shows the ciphertext when 3072-bits key is used with the RSA encryption transformation.



```
E¾Ëñ□Nß§K O¦:ÛËFa-Å¨´ÅÎ £k?□öïž9 åm°‰„p˝ê, □«□·÷<v WOÛ ±ç" jÉE8šae_÷©zMì¦Ñ(Å°È¥÷¸öÜA ÏDi'5 ÈqËF TőÒ ‰
`7+õ¨7'Aå4W¡¬×□^Áç˜Ë˜\œ "ÝÕô é2b :>ùku'Û□ zËâ ÃÊO³ü ™P@öÊ=' QSu$,k ²ïÖl°ûŽû— ü. J }»ÿê ¯xé p}ôðmÿuN þ˜‡YùŸ+Ëkéh¢
Ã•~±), ö%1jòÛ1Ãz Eš°]&gO•˝Ná7î@tÐ¤Ÿ è/È ¡·Ô¹a¦×õóWàd$t… ÔbQ°ÐOÝP»c¨e^^Œµc ³êM¨cÜ•k Ëè°?Rk
[Æïë6µéËÎ×¸n±²ÓouØ□9$ŽHW i ÿ#ú6æRØ"ßÀ W ¯õØþ½- ½@†É;Ê˜†AìYÃ[-X±×ö&ÙÎ ©k¼¿□=óÞX□
```

The screenshot below shows the measured performance when 3072-bits key is used with the RSA encryption transformation.



Key Generation Time :	56799.0 milliseconds
Encryption Time :	531.0 milliseconds
Decryption Time :	655.0 milliseconds
PlainText Size :	100 bytes
CipherText Size :	384 bytes