Elliptic Curve Cryptography Design and Implementation

Final Project Report

Prepared by Victor Thompson

Distributed May 6, 2008

Prepared for
Dr. Brad Rubin's Computer Security Class (SEIS-720)
University of St. Thomas

Abstract

The goal of this project was to implement a suite of algorithms intended as an replacement for existing public-key cryptographic systems such as RSA and DSA (Digital Signature Algorithm), which can be used to encrypt messages and generate digital signatures using ECC (Elliptic Curve Cryptography). The implementation uses predefined primitives utilizing elliptic curves over finite fields to forge algorithms for encryption and signing. The following report details the design, implementation, and testing of the system to be built. Wrapping up the discussion, are sections on lessons to be learned and conclusions that can be drawn from this project.

Contents

List of Tables

List of Figures

1 Introduction

The implementation detailed in this project chooses to bravely reject one of the foremost tenets of computer security. While creating a cryptographic algorithm is generally frowned upon, the design of the algorithms targeted in this project borrow from numerous reputable sources. The primitives and general cryptographic steps are all derived from either academic sources or government mandates. The resulting public-key encryption and digital signature protocols are wrapped up into the cutely named *T* protocol. While it may already be apparent—*T* is a play on the common cryptographic standards RSA (Rivest, Shamir, and Adleman) and DH (Diffie-Hellman) quite appropriately (protocol plagiarism aside) named after the author.

1.1 Motivation

Cryptography based on elliptic curves is something that has been discussed for years. However, only recently have increasing security concerns, and the push to provide high levels of security on mobile and embedded devices, made the adoption of elliptic curve cryptography necessary. The computational efforts of the standard RSA and DSA algorithms require increasingly large key lengths and computation time to scale with current levels of security [1]. Smaller key sizes and increased performance provided by elliptic cryptography allow the current security needs to scale with future requirements and also provide fast and compact implementations on mobile and embedded devices. The following table equates popular public key cryptography strengths with those of equivalent symmetric protocols [2].

Table 1: Equivalent Strengths of Various Curve Parameters

RSA/DSA Key Length	EC Key Length	Symmetric Equivalent
512	112	DES (56-bit)
1024	160	SKIPJACK (80-bit)
2048	224	3-DES (112-bit)
3072	256	AES Small (128-bit)

RSA/DSA Key Length	EC Key Length	Symmetric Equivalent
7680	384	AES Medium (192-bit)
15360	521	AES Large (256-bit)

Given that 1024-bit key lengths are the norm for current RSA and DSA implementations, and 2048-bit keys are starting to become more prominent due to increasing fears that 1024-bit keys will soon be breakable, the scalability of elliptic curve cryptography promises to provide reasonable key lengths that linearly scale with symmetric equivalents.

1.2 Objective

The final product of this project is a working implementation of a cryptographic suite based on elliptic curves that is capable of encrypting short messages and generating digital signatures. Included with the final product are benchmarks used to analyze the products performance against existing protocols and standards.

2 Mathematical Background and Primitives

The algorithms implemented in this documents make use of computationally complex problems, much like the current de-facto public-key cryptographic standard—RSA. The most significant difference being that the computations occur over finite fields using elliptic curve equalities.

2.1 Finite Fields

A finite field, or more generally—a field, is an algebraic structure that satisfies the axioms over addition, subtraction, multiplication and division. Fields satisfying these rules that contain a finite number of elements are referred to as finite fields or Galios fields. A typical field that is easy to grasp is the field that covers real numbers. Real numbers (**R**) comprise a field of infinite size in the following ways:

Additive

- Given a=5 and b=2, since a and b belong to **R**, then $a+b$ (7)is in **R** as well, known as closure.
- Given c=3, $a+(b+c) = (a+b)+c$, since addition is associative under **R**, 10=10, which is also in **R**.
- For any element there exists an identity element where $x+I = I+x = x$, for x=a,b,c I=0, which is in **R**.
- Given that I=0 from above, for any element there exists an inverse element in **R** where $x+x' = x'+x = I$, for x=a,b,c x'=-5, -2, and -3, respectively.
- Addition is commutative under **R**, so $a+b = b+a$.

Multiplicative

- $a*b$ = 10 is in **R**, covering closer under multiplication.
- Multiplication is associative under **R**, so $a*(b*c) = (a*b)*c$ = 30, which is in **R**.
- Multiplication is distributive under **R**, so $a*(b+c) = a*b+a*c$ = 25, which is in **R**.
- Multiplication is commutative under **R**, so $a*b = b*a$.
- For any element there exists an identity element where $x*I = I*x = x$, for x=a,b,c I=1, which is in **R**.
- Given that I=1from above, for any element there exists an inverse element in **R** where $x*x' = x'*x = I$, for x=a,b,c x'=5^{-1}, 2^{-1}, and 3^{-1}, respectively, each

in the field **R**.

2.2 Elliptic Curve Algebra Over Real Numbers

An elliptic curve is a 2-dimensional curve typically defined by the equation

$$y^2 = x^3 + ax + b \qquad (2.1)$$

for which solutions form an abelian group that, by definition, covers the additive axioms from the previous section. The variables x and y are then chosen from either a field (like **R**) or a finite field (like prime fields Z_p or binary fields Z_{2m}).

Equation 2.1 defining the curve introduces some interesting additive identities. For a line touching or intersecting the curve, there exists three points defined as:

- if three points (P, Q, R) on the line exist, $P + Q + R = O$ (the additive identity),
- a point tangent to the curve is defined as intersecting the curve twice, and
- a point outside the curve, at infinity, is defined as O the additive identity, can be used as the third point if none exist.

These observations lead to four fundamental equalities that are defined by the four cases created by the above rules.

Table 2: Identity Cases for Elliptic Curves

Case	Identity
Three points on line	$P + Q + R = O$, $P + Q = -R$
Line tangent at Q	$P + Q + Q = O$, $P = -2Q$
No R, substitute O	$P + Q + O = O$, $P + Q = O$

Case	Identity
Line tangent at P, no Q or R	$P + P + O = O$, $P - P = O$

The identities in Table 2 can be proven to form an abelian group for the set of elements in the solution set comprised of *a* and *b* from Equation 2.1.

The group can also be defined algebraically [3]. For the points *P* and *Q*, The slope of the resulting line can be described as

$$m=(y_Q - y_P)/(x_Q - x_P) \tag{2.2}$$

which allows the sum $R = P + Q$ to be expressed as

$$x_R = m^2 - x_P - x_Q \tag{2.3}$$

$$y_R = -y_P + m(x_P - x_R) \tag{2.4}$$

If $x_P = x_Q$ and $y_P = y_Q$ but $y_P \neq 0$, then

$$m = (3x_P^2 - a)/(2y_P) \tag{2.5}$$

$$x_R = m^2 - 2x_P \tag{2.6}$$

$$y_R = y_P + m(x_R - x_P) \tag{2.7}$$

Otherwise, if $y_P = y_Q = 0$, then $P + P = O$.

2.3 Elliptic Curve Algebra Over the Odd Prime Finite Field

While both prime and binary finite fields are used in standard elliptic curve cryptography, this implementation makes use of only prime field curves. The elements in the prime finite field Z_p represent a set of *p* integers where addition and multiplication are defined as $a + b \equiv r \pmod{p}$ and $a * b \equiv s \pmod{p}$ respectively. Likewise, the equation relating the general elliptic curve to real numbers can be be written as either

$$y^2 \bmod p = (x^3 + ax + b) \bmod p \tag{2.8}$$
$$y^2 \equiv x^3 + ax + b \pmod{p} \tag{2.9}$$

where coefficients and variables are in Z_p.

Addition rules can be defined for solutions to Equation 2.8 [4]

- Adding points at infinity, $O + O = O$
- Additive identity, $(x, y) + O = O + (x, y) = (x, y)$
- Adding duplicate x-coordinates, $(x, y) + (x, -y) = O$ or $-(x, y) = (x, -y)$
- Adding differing x-coordinates,

$$m = ((y_Q - y_P)/(x_Q - x_P)) \pmod{p} \tag{2.10}$$
$$x_R \equiv m^2 - x_P - x_Q \pmod{p} \tag{2.11}$$
$$y_R \equiv m(x_P - x_R) - y_P \pmod{p} \tag{2.12}$$

- Doubling a point (basis for multiplication),

$$m = ((3x_P^2 + a)/(2y_P)) \pmod{p} \tag{2.13}$$
$$x_R \equiv m^2 - 2x_P \pmod{p} \tag{2.14}$$
$$y_R \equiv m(x_P - x_R) - y_P \pmod{p} \tag{2.15}$$

The last rule forms the basis for scalar multiplication across points on the curve, which facilitates the computationally complex problem used in both encryption and signing for elliptic curve cryptography.

3 Design Considerations

There were a number of documents taken into consideration while the encryption and digital signature protocols for *T* were being investigated. Gathering standards that effectively utilized the primitives to do both encryption and signing proved to be difficult. Given the difficulty of solving similar computationally complex problems with both public and private key fragments, the two protocols had to be separated and the implementation of each differs greatly.

3.1 Designing an Encryption Algorithm Over Elliptic Curves

The first portion of the *T* algorithm that was investigated was the portion responsible for both encryption and decryption. The FIPS and ANSI X9 mandates did not cover a dedicated encryption method, and instead proposed a key agreement strategy (much like DH) utilizing a separate symmetric encryption scheme. While this was promising, the intent of the project was to develop something more than just key agreement—a candidate seeking to replace RSA would need to utilize arbitrary key transport as well as the ability to encrypt messages of (ideally) short lengths.

3.1.1 Encryption Algorithm in Stallings

One potential protocol [3], described below, seemed able to fulfill the requirements set forth for the encryption algorithm of *T*. The encryption algorithm chosen for *T*, works as follows:

1. Public and private keys are selected such that the private key is a random integer between 1 and the order of the field *n* and the public key is a multiple of the base point, G, times the private key.
2. The message is mapped to a point and encrypted as a pair of points.
3. The ciphertext is decrypted by taking a multiple of the private key and the first point and subtracting it from the second point leaving the mapped message due to the commutative properties of the group.

3.1.2 Message Mapping and Padding Considerations

The mapping of the message to a point on the curve is a seemingly difficult task. However, many of these primitives exist in the Bouncy Castle Crypto APIs [5]. While it was the goal of this project to also implement the mathematical primitives that make up the chosen algorithms, the implementation of some of the pieces were out of scope of this project.

In order to map the point to the curve, the message needs to be sufficiently padded. Some of the padding structures looked at for this implementation include:

- Byte Padding (ANSI X9.23)
- PKCS7
- Zero Padding
- Prefixing with Zero Padding

The padding scheme chosen for the encoded point was that of the ANSI X9.23 standard which is defined by appending zero-bytes to fill the block and specifying the number of padding bytes used in the final byte of the block.

3.2 Designing a Digital Signature Algorithm Over Elliptic Curves

The basics for the algorithm for Elliptic Curve Digital Signature (ECDSA) is well established in the research that has been done to investigate and propose elliptic curve as a potential cryptographic standard. The signature scheme was proposed in ANSI X9.62 [6] and FIPS 186-2 [7] and supported by the security research firm Certicom. The algorithm functions as follows:

1. Public and private keys are selected such that the private key is a random integer between 1 and the order of the field n and the public key is a multiple of the base point, G, times the private key. This is identical to the EC encryption algorithm proposed above.
2. A random number is used as a scalar for the base point of which the x value is used to create an integer r. A second integer, s, is created from the hash of the message and a scalar of the private key and the inverse modulus of

the order value *n* of the random integer. The signature is the pair of these two integers (*r*, *s*).

3. The signature and the hashed plaintext are used to calculate the sum of two multiples containing the base point and the public key of which the *x* value is taken (mod *n*) and compared to the first integer. If these two integers match, the signature has been verified.

3.2.1 Signature Construction and Deconstruction Methodologies

The signature is represented as the pair of two integers. To combine the values into a deconstructible entity, the values need to be matched in length and combined. For instance, if the pair (*r*, *s*) are represented by 123456 and 1234 respectively, then to combine them into an array (a byte array will be used in the actual implementation, but for the sake of demonstration a string will represent the final signature) the values need to be matched in length so that they can be pulled apart by the verification process. To do this, we can use one of the following padding mechanisms:

- Byte Padding (ANSI X9.23)
- PKCS7
- Zero Padding
- Prefixing with Zero Padding

Since these will be split into two integers, the most logical padding method is to prefix the shortest value with zeros until both values are of equal length. This leads to the strings of 123456 and 001234, for *r* and *s*, respectively. The signature can then be passed on as the array 123456001234. Since the two integers are equal in length, the array can be equally split at the midpoint to preserve the original values.

3.2.2 Hash Function Selection

Since the algorithm being derived will not benefit from the pluggable hashing and padding algorithms, a standard hash function should also be selected. A secure, yet compact, message digest function should be used for this purpose. The candidates for this selection are:

- MD5
- SHA-1
- SHA-256

The broken 128-bit MD5 sum was passed over due to its partial insecurity [8] [9] and the SHA-256 function was left unimplemented, favoring the smaller hash value of the 160-bit SHA-1 function (which is also vulnerable to attack) [10].

4 Implementation

The phase to implement the algorithms developed in the design of *T* had seven potential sections

- developing the mathematical primitives,
- defining selectable curves,
- creating protocol class and key establishment,
- implementing the encryption algorithm,
- implementing the digital signature algorithm,
- creating test parameters and benchmark cases, and
- implementing the algorithms as a JCA Provider.

In order to finish the project with something substantial to report, the mathematical primitives were borrowed from the Bouncy Castle Crypto APIs and the protocols were not implemented as a Java Cryptography Architecture Provider [11]. The following sections detail the five tasks that were completed.

4.1.1 Defining Selectable Curves

The first step taken in implementing the algorithms discussed in this report was to generate a method of creating curves based on recommended parameters put out by the SEC and NIST [2]. The `ApprovedCurves` class derives and implements these curves based on selected SEC curve parameters. The class creates an object with the given parameters based on a string specifying the name of the curve given by the SEC. The following are the curve parameters used in this project:

- secp160r1
- secp224r1
- secp384r1
- secp521r1
- secp160k1

- secp224k1

The format of the above curve names detail originating agency (sec), the finite field used (p), the curve domain bit-length (160), the curve type as either random or Koblitz (k), and finally the number in sequence of the curve with the given parameters (1).

4.1.2 Creating Protocol Class and Key Establishment

The implemented algorithms share a common class, `Texample`, designed carry out the methods for curve selection, key establishment, encryption, decryption, signature generation, and signature verification. The `establishKeys` method populates the class's fields with a given set of curve parameters and creates the user's public and private keys.

4.1.3 Implementing the Encryption Algorithm

The encryption algorithm was the most difficult portion to implement. While the algorithm specified in the text book [3] provided the fundamentals for a data encryption/decryption scheme using elliptic curves, the details of its implementation was left to be implemented. The design decisions discussed in this report aided in getting the protocol working.

The padding choice for the message to be encoded to a point proved to be a good decision, as problems were encountered while padding the encoded point differently. Converting the two elliptic curve points to a single data structure was another source of conflict. The final implementation simply stored the two points as separate sections of a byte array, which allowed them to be easily deconstructed or printed.

4.1.4 Implementing the Digital Signature Algorithm

The algorithm for providing digital signature capability was the better documented of the two algorithms. The ECDSA algorithm is well published and even has implementation guidelines in many of the standards and best practices documents. The implementation uses the method for typical DSA tailored for elliptic curves. This process is relatively simple and results in a standard security strength (whereas the encryption algorithm *may* have holes). Some of the more difficult tasks in implementing the signatures algorithm were once again signature construction/deconstruction and verification.

4.1.5 Creating Test Parameters and Benchmark Cases

In order to analyze *T*'s performance, equivalent cryptographic schemes had to be benchmarked against the corresponding elliptic curve parameters. To facilitate in the creation of these test cases, a `Generator` class was created to abstract the creation of various schemes such as RSA, Bouncy Castle's ECIES (Elliptic Curve Integrated Encryption Scheme), Bouncy Castle's ECDSA, and AES. The `Drive` class was then created to manage and create specific test cases for use in later analysis.

5 Testing and Performance Tweaking

The test parameters laid out in the implementation phase helped guide the testing phase of this project. This section details the target platforms used in test, the parameters and basis for comparison, test results and analysis, and finally potential performance enhancements.

5.1.1 Target Platforms

The final implementation of the *T* cryptography suite is intended to be platform agnostic and should run on any system with the Java SE 6 JRE installed. The software was not verified against the Java SE 5 JRE or earlier, so results may very. The software platforms that were used for functional verification and benchmarking were Windows XP and the GNU/Linux distribution Ubuntu 7.10 running on the following hardware:

- Windows XP: a Dell D610 laptop with a 1.6 GHz Intel® Pentium® M processor and1 GB of RAM.
- Ubuntu 7.10: a custom built PC with a 2.4 GHz Intel® Core^{TM} 2 E6600 processor, 2 GB of RAM, and 2 GB of swap space.

The performance on these two systems vary greatly, so the performance seen on the Windows XP system will be normalized to correspond to the values seen running under Linux.

5.1.2 Test Parameters

The basis for comparison will be the time of completion between phases. This basis will be used to compare the test parameters that are deemed equivalent by the SEC, provided in Table 3.

Table 3: Key Lengths of Test Parameters

RSA	T/ECIES/ECDSA	AES
1024	160	NA
2048	224	NA
7680	384	192
15360	521	256

For the encryption and signature measurements, the target plaintext will be a 256-bit AES key, which is assumed to be made up of seemingly random data.

5.1.3 Test Results

The following is a section of output captured from an iteration of each test parameter cycled a reasonable number of times:

```
Algorithm | Key size  | Key (s)     | Encrypt (s) | Decrypt (s) | Sign (s)    | Verify (s)  | Total (s)   | # of Runs  | Match? |
Verified?
----------------------------------------------------------------------------------------------------
RSA       | 1024      | 0.14866861 | 0.000787165 | 0.005073835 | 0.00291367 | 0.000303938 | 0.157747218 | 10.0 | true | true
ECIES     | secp160r1 | 0.037831414 | 0.043496187 | 0.009172243 | 0.009008949 | 0.013457076 | 0.11296587 | 10.0 | true | true
T         | secp160r1 | 0.009196458 | 0.018131046 | 0.010205454 | 0.009512227 | 0.011263528 | 0.058308712 | 10.0 | true | true
T         | secp160k1 | 0.007387066 | 0.014567958 | 0.007521267 | 0.00754368 | 0.010226161 | 0.047246132 | 10.0 | true | true

RSA       | 2048      | 1.459228328 | 0.000874385 | 0.02006922 | 0.018024959 | 0.000616225 | 1.498813118 | 10.0 | true | true
ECIES     | secp224r1 | 0.029801847 | 0.014862534 | 0.015279537 | 0.015121733 | 0.01983407 | 0.094899721 | 10.0 | true | true
T         | secp224r1 | 0.014984438 | 0.029776188 | 0.015048209 | 0.014913733 | 0.019477249 | 0.094199818 | 10.0 | true | true
T         | secp224k1 | 0.015230441 | 0.02901234 | 0.014974046 | 0.014646104 | 0.019558718 | 0.093421649 | 10.0 | true | true

RSA       | 7680      | 172.010816078 | 0.006740515 | 0.785718273 | 0.786169087 | 0.006371304 | 173.595815257 | 1.0 | true | true
AES       | 192       | 0.000035961 | 0.000260291 | 0.000135339 | 0 | 0 | 0.000431591 | 100.0 | true | false
ECIES     | secp384r1 | 0.109816227 | 0.056346281 | 0.05228687 | 0.051341556 | 0.070124821 | 0.339915754 | 10.0 | true | true
T         | secp384r1 | 0.049875336 | 0.098457247 | 0.048456601 | 0.048959336 | 0.063808895 | 0.309557415 | 10.0 | true | true

RSA       | 15360     | 1325.514085486 | 0.027067397 | 6.632376421 | 6.353244307 | 0.025803841 | 1338.552577452 | 1.0 | true | true
AES       | 256       | 0.000061361 | 0.000195348 | 0.001140332 | 0 | 0 | 0.001397042 | 100.0 | true | false
ECIES     | secp521r1 | 0.205002787 | 0.101849578 | 0.101433959 | 0.104859625 | 0.135783553 | 0.648929501 | 10.0 | true | true
T         | secp521r1 | 0.110593508 | 0.215714647 | 0.107547962 | 0.105959707 | 0.137998746 | 0.677814569 | 10.0 | true | true
```

It should be noted that signature verification is not done by AES, so the timing data and verified boolean values reflect this shortcoming. When represented graphically, each test parameter shows essentially the same data. Figure 1 below adequately shows how extensive certain operations are for each protocol with test parameters comparable to that of RSA-1024. Notably, the key establishment operation of RSA seems to be extremely time consuming, while the remaining operations are extremely lightweight.

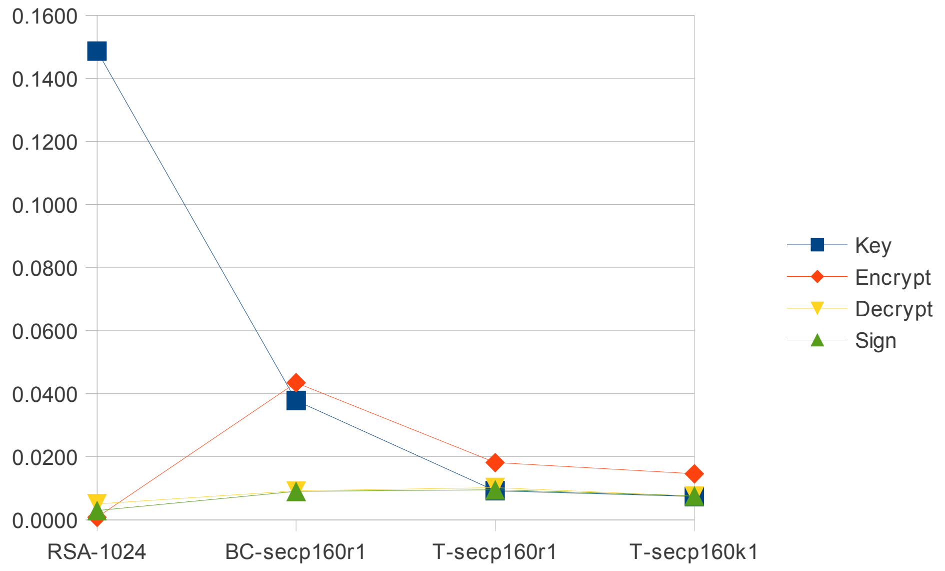

Figure 1: Time in seconds for each operation of each protocol comparable to RSA-1024

The same information is clearly represented in Figure 2. The time spent by each protocol in the various operations

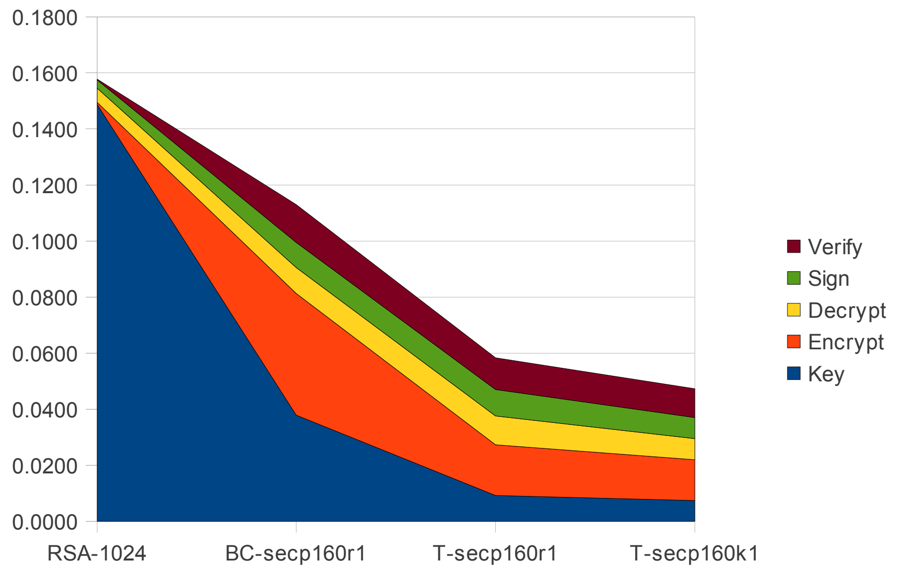

Figure 2: Time in seconds for each protocol comparable to RSA-1024 weighted by operation

As the key lengths increase the RSA protocol becomes more and more computationally heavy when it comes to key establishment. As a result, the remaining graphs will leave out key establishment from the representations. One thing to note is that this will put the *T* protocols at a disadvantage due to their responsiveness to key establishment.

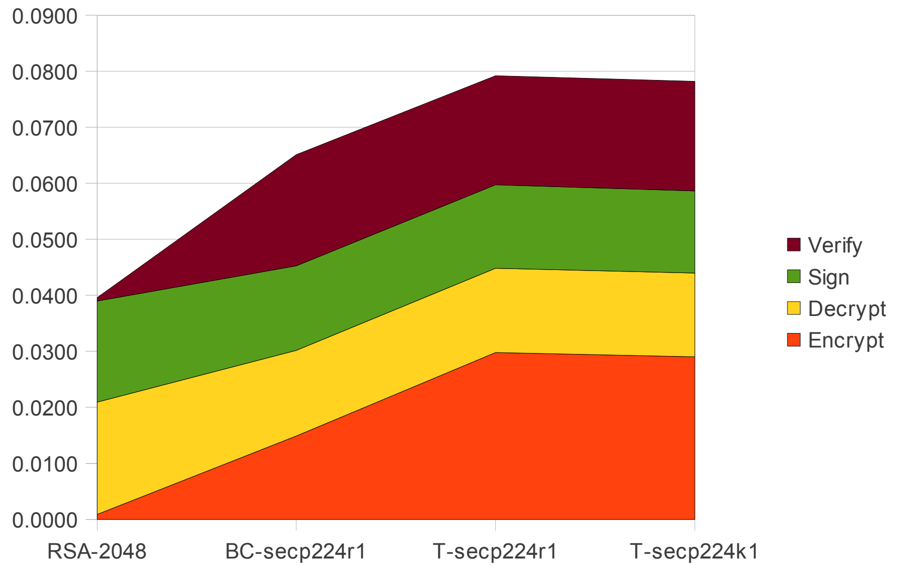

Figure 3: Time in seconds for each protocol comparable to RSA-2048 weighted by operation

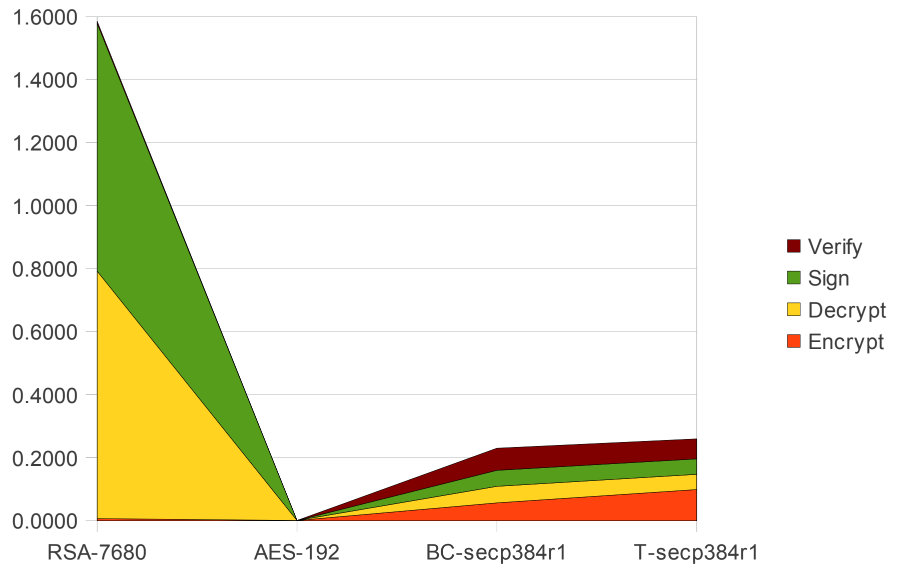

Figure 4: Time in seconds for each protocol comparable to RSA-7680 weighted by operation

Figures 3 and 4 show an increasing level of complexity for the operations of decryption and signature generation under the RSA protocol as the key lengths increase. It is note-worthy that these operations are functions of the private key. Figure 4 also introduces AES into the mix. The inclusion does nothing more than highlight the absurdly fast capabilities of AES and symmetric cryptography in general.

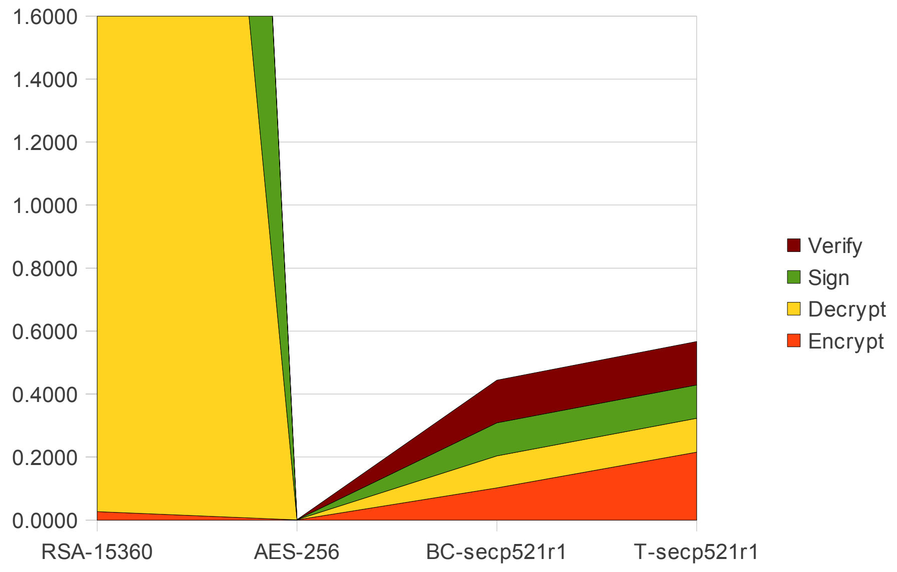

Figure 5: Time in seconds for each protocol comparable to RSA-15360 weighted by operation

The important piece of information to take away from Figure 5 is the how adept the elliptic curve protocols are at scaling as key lengths increase. Even an operation that RSA can complete quickly, like signature verification can't scale nearly as well as the elliptic curve protocols. Figures 6 and 7 illustrate the scalability of the signature verification process in particular.

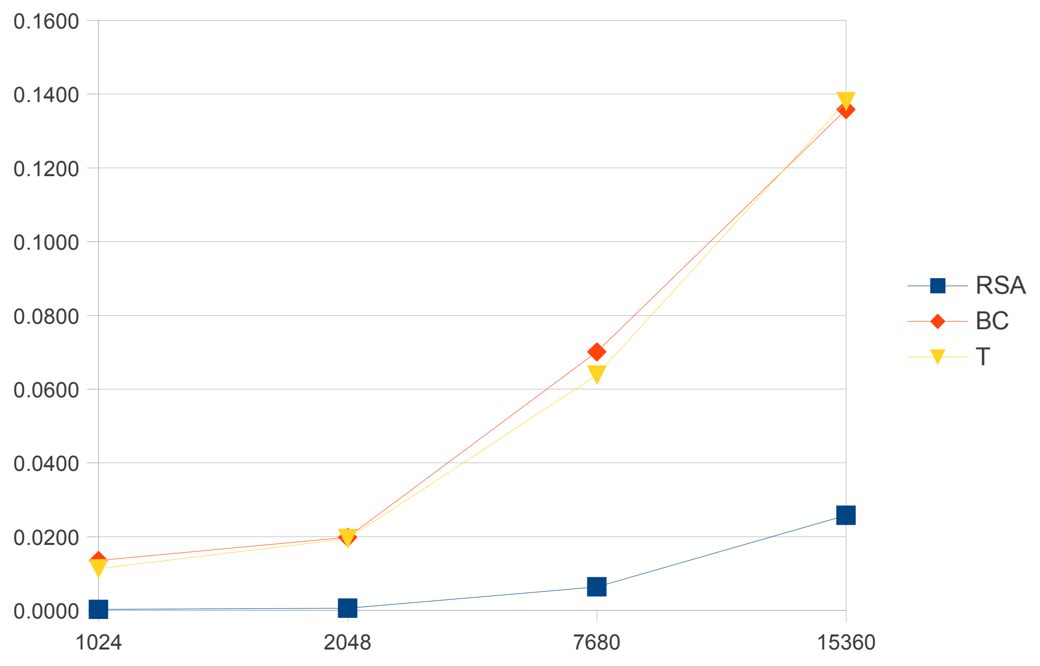

Figure 6: Time in seconds for signature verification

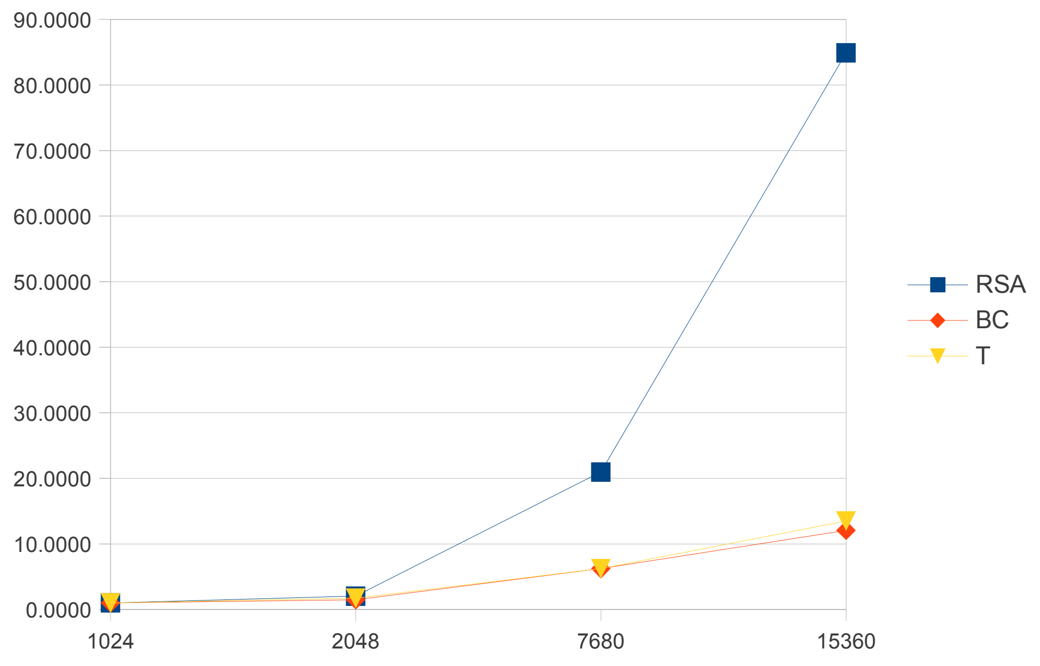

Figure 7: Magnitude of time increase for signature verification

Figures 6 and 7 attempt to show the level of scalability provided by elliptic curve cryptography. Figure 7 is perhaps the more interesting of the two, as it clearly shows the magnitude of time increase for comparably secure key lengths. In reference to 1024-bit RSA, the time to verify a signature using 15360-bit RSA took almost 85 times as long; whereas the comparable elliptic curve protocols increased by a factor of less than 14.

5.1.4 Performance Tweaking

Computationally, the most complex portions of the each algorithm in *T* are the tasks dedicated to random number generation of a certain bit length. This is done once for key establishment (to be used as a key) and again for either encryption or signature generation (to aid in the creation of ciphertext and signatures). There are a number of things that could be done, at the risk of negating a level of security present in the current algorithms. To speed up this process the implementation could:

- find a quicker random number generator, or
- generate both random numbers during key establishment.

Each change has inherent security concerns associated with its implementation. Using a random number generator other than Java's `SecureRandom` might create numbers of uneven distributions and with insecure properties. Secondly, generating both random numbers during key establishment means that the random number introduced for encrypting and signing will be utilized for every pass during the existence of the session. One feature that could be added to avoid this would be a method used to generate a new random variable on demand. However, the use of this method in practice would be scarce and its avoidance would leave the protocols in an insecure state. Therefore, none of these options were added to the suite because of the potential security holes attributed to them.

6 Lessons Learned

Creating protocols using elliptic curve cryptography proved to be an interesting challenge—even without the need of developing the mathematical primitives. The amount of testing needed to secure variable length messages, variation in message content, hashing, padding, data representation, and error handling proved to be greater than originally anticipated. This extra effort made the project all the more valuable. Furthermore, the creation of the implementations themselves helped remove some of the mystique surrounding cryptography, especially software implementations of cryptography. Overall, a lot was learned from this project and the opportunity to further develop the algorithm is something that will definitely be hard to turn down.

7 Conclusions

This report presents an implementation of cryptography based on elliptic curves over prime finite fields. The protocols are based on algorithms supported by research or proposed in academic literature and are all comprised of recommended curve parameters. The implementation makes use of preexisting mathematical primitives provided by the developers of the Bouncy Castle Crypto APIs. The resulting implementation is ideal for mobile device, real-time systems, and systems with portable code requirements.

The test results support the scientific and marketing claims that elliptic curve cryptography is a computationally lightweight protocol that is ideal for mobile devices and thin clients where power consumption and speed requirements are matched with strong cryptographic needs. The scalability of the protocol's performance is far greater than that of RSA or DSA. This will allow future privacy requirements to increase without sacrificing performance.

APPENDIX A: References

[1] Certicom Research, *An Introduction to the Uses of ECC-based Certificates*, <`http://www.certicom.com/index.php?action=res,cc&issue=2-2&&article=1`>. Accessed Apr 22, 2008.

[2] Certicom Research, *SEC 2: Recommended Elliptic Curve Domain Parameters*, Standards for Efficient Cryptography, Version 1.0, Sep. 2000.

[3] Stallings, William, *Cryptography and Network Security Principles and Practices*, Fourth Edition, Prentice Hall, 2006.

[4] Certicom Research, *SEC 1: Elliptic Curve Cryptography*, Standards for Efficient Cryptography, Version 1.0, Sep. 2000.

[5] Anderson-Lee, Jeff, *ECCurve* , <`http://www.cs.berkeley.edu/~jonah/javadoc/org/bouncycastle/math/ec/ECCurve.html`>. Accessed Apr 20, 2008.

[6] Accredited Standards Committee X9 Incorporated , *ANSI/X9 X9.62-2005: Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)*", 2005.

[7] National Institute of Standards and Technology, *FIPS 186-2: Digital Signature Standard*, <`http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf`>. Accessed Apr 21, 2008

[8] Wikipedia, *MD5*, <`http://en.wikipedia.org/wiki/MD5`>. Accessed Apr 21, 2008.

[9] Abzug, Mordechai T, *MD5 Homepage (unofficial)*", <`http://userpages.umbc.edu/~mabzug1/cs/md5/md5.html`>. Accessed Apr 22, 2008.

[10] Schneier, Bruce, *Schneier on Security: SHA-1 Broken*, <http://www.schneier.com/blog/archives/2005/02/sha1_broken.html>. Accessed Apr 22, 2008.

[11] Sun Microsystems, *How to Implement a Provider in the Java™ Cryptography Architecture*, <http://java.sun.com/javase/6/docs/technotes/guides/security/crypto/HowToImplAProvider.html>. Accessed Apr 23, 2008.

APPENDIX B: Source Code

B.1 AESexample.java

```
import java.security.Key;
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;

public class AESexample {

      private String keySize;
      private Key aesKey;


      public AESexample () throws Exception{

      }

      public void establishKeys(String keysize) throws Exception {

          KeyGenerator keyGenSingle = KeyGenerator.getInstance("AES");
          keyGenSingle.init(Integer.parseInt(keysize));
          this.aesKey = keyGenSingle.generateKey();

          this.keySize= Integer.toString(aesKey.getEncoded().length*8);


      }

      public byte[] encrypt(byte[] plainText) throws Exception {

          // get an AES cipher object and print the provider
          Cipher cipher = Cipher.getInstance("AES");

          // encrypt the plaintext using the public key
          cipher.init(Cipher.ENCRYPT_MODE, aesKey);
          return cipher.doFinal(plainText);

      }

      public byte[] decrypt(byte[] cipherText) throws Exception {

          // get an AES cipher object and print the provider
          Cipher cipher = Cipher.getInstance("AES");

          // decrypt the text using the private key
          cipher.init(Cipher.DECRYPT_MODE, aesKey);
          return cipher.doFinal(cipherText);
      }

      public String getKeySize() {
            return keySize;
      }

}
```

B.2 ApprovedCurves.java

```
import org.bouncycastle.math.ec.ECCurve;
import org.bouncycastle.math.ec.ECPoint;
import java.math.BigInteger;

public class ApprovedCurves {
      private BigInteger p;
      private BigInteger a;
      private BigInteger b;
      private BigInteger n;
      private BigInteger h;
      private ECCurve curve;
      private ECPoint G;

      public ApprovedCurves(String named) {

            // SEC equiv RSA 1024 (80 bit)
            if(named == "secp160r1"){
                  this.p = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFF", 16);
                  this.a = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7FFFFFFC", 16);
                  this.b = new
BigInteger("1C97BEFC54BD7A8B65ACF89F81D4D4ADC565FA45", 16);
                  this.n = new
BigInteger("0100000000000000000001F4C8F927AED3CA752257", 16);
                  this.h = BigInteger.valueOf(1);
                  this.curve = new ECCurve.Fp(p, a, b);
                  this.G = curve.decodePoint(new BigInteger("04"
                              +
"4A96B5688EF573284664698968C38BB913CBFC82"
                              +
"23A628553168947D59DCC912042351377AC5FB32", 16).toByteArray());

            // SEC equiv RSA 2048 (112 bit)
            } else if(named == "secp224r1"){
                  this.p = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000000000000000000001",
16);
```

```
                  this.a = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFE",
16);
                  this.b = new
BigInteger("B4050A850C04B3ABF54132565044B0B7D7BFD8BA270B39432355FFB4",
16);
                  this.n = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFF16A2E0B8F03E13DD29455C5C2A3D",
16);
                  this.h = BigInteger.valueOf(1);
                  this.curve = new ECCurve.Fp(p, a, b);
                  this.G = curve.decodePoint(new BigInteger("04"
                      +
"B70E0CBD6BB4BF7F321390B94A03C1D356C21122343280D6115C1D21"
                      +
"BD376388B5F723FB4C22DFE6CD4375A05A07476444D5819985007E34",
16).toByteArray());

            // SEC equiv RSA 7680 (192 bit)
            } else if(named == "secp384r1"){
                  this.p = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFEFFFFFFFF0000000000000000FFFFFFFF", 16);
                  this.a = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFEFFFFFFFF0000000000000000FFFFFFFC", 16);
                  this.b = new
BigInteger("B3312FA7E23EE7E4988E056BE3F82D19181D9C6EFE8141120314088F501
3875AC656398D8A2ED19D2A85C8EDD3EC2AEF", 16);
                  this.n = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC7634D81F43
72DDF581A0DB248B0A77AECEC196ACCC52973", 16);
                  this.h = BigInteger.valueOf(1);
                  this.curve = new ECCurve.Fp(p, a, b);
                  this.G = curve.decodePoint(new BigInteger("04"
                      +
"AA87CA22BE8B05378EB1C71EF320AD746E1D3B628BA79B9859F741E082542A385502F2
5DBF55296C3A545E3872760AB7"
                      +
"3617DE4A96262C6F5D9E98BF9292DC29F8F41DBD289A147CE9DA3113B5F0B8C00A60B1
CE1D7E819D7A431D7C90EA0E5F", 16).toByteArray());

            // SEC equiv RSA 15360 (256 bit)
            } else if(named == "secp521r1"){
                  this.p = new
BigInteger("01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FF", 16);
                  this.a = new
BigInteger("01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FC", 16);
                  this.b = new
BigInteger("0051953EB9618E1C9A1F929A21A0B68540EEA2DA725B99B315F3B8B4899
18EF109E156193951EC7E937B1652C0BD3BB1BF073573DF883D2C34F1EF451FD46B503F
00", 16);
                  this.n = new
BigInteger("01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFA51868783BF2F966B7FCC0148F709A5D03BB5C9B8899C47AEBB6FB71E913864
09", 16);
                  this.h = BigInteger.valueOf(1);
                  this.curve = new ECCurve.Fp(p, a, b);
                  this.G = curve.decodePoint(new BigInteger("04"
                      +
"00C6858E06B70404E9CD9E3ECB662395B4429C648139053FB521F828AF606B4D3DBAA1
4B5E77EFE75928FE1DC127A2FFA8DE3348B3C1856A429BF97E7E31C2E5BD66"
                      +
"011839296A789A3BC0045C8A5FB42C7D1BD998F54449579B446817AFBD17273E662C97
EE72995EF42640C550B9013FAD0761353C7086A272C24088BE94769FD16650",
16).toByteArray());

            // Koblitz equiv RSA 1024 (80 bit)
            } else if(named == "secp160k1"){
                  this.p = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFAC73", 16);
                  this.a = BigInteger.valueOf(0);
                  this.b = BigInteger.valueOf(7);
                  this.n = new
BigInteger("0100000000000000000001B8FA16DFAB9ACA16B6B3", 16);
                  this.h = BigInteger.valueOf(1);
                  this.curve = new ECCurve.Fp(p, a, b);
                  this.G = curve.decodePoint(new BigInteger("04"
                        + "3B4C382CE37AA192A4019E763036F4F5DD4D7EBB"
                        + "938CF935318FDCED6BC28286531733C3F03C4FEE",
16).toByteArray());

            // Koblitz equiv RSA 2048 (112 bit)
            } else if(named == "secp224k1"){
                  this.p = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFE56D",
16);
                  this.a = BigInteger.valueOf(0);
                  this.b = BigInteger.valueOf(5);
                  this.n = new
BigInteger("010000000000000000000000000001DCE8D2EC6184CAF0A971769FB1F7",
16);
                  this.h = BigInteger.valueOf(1);
                  this.curve = new ECCurve.Fp(p, a, b);
                  this.G = curve.decodePoint(new BigInteger("04"
                      +
"A1455B334DF099DF30FC28A169A467E9E47075A90F7E650EB6B7A45C"
                      +
"7E089FED7FBA344282CAFBD6F7E319F7C0B0BD59E2CA4BDB556D61A5",
16).toByteArray());
            }

      }
```

```
	public BigInteger getP() {
		return p;
	}

	public BigInteger getA() {
		return a;
	}

	public BigInteger getB() {
		return b;
	}

	public BigInteger getN() {
		return n;
	}

	public BigInteger getH() {
		return h;
	}

	public ECCurve getCurve() {
		return curve;
	}

	public ECPoint getG() {
		return G;
	}
}
```

B.3 Drive.java

```
import javax.crypto.*;

public class Drive {

	/**
	 * @param args
	 */
	public static void main(String[] args) throws Exception{

		KeyGenerator keyGenSingle = KeyGenerator.getInstance("AES");
		keyGenSingle.init(256);
		byte[] plainText = keyGenSingle.generateKey().getEncoded();

//		  SecureRandom random = new SecureRandom();
//		byte[] plainText = new byte[39];
//		random.nextBytes(plainText);

		Generator G;
		System.out.println("Algorithm | Key size  | Key (s)     |
Encrypt (s) | Decrypt (s) | Sign (s)    | Verify (s)  | Total (s)   | #
of Runs   | Match? | Verified? " );

System.out.println("----------------------------------------------------
---------------------------------------------------");

          G = new Generator ("RSA", "1024", plainText, 10);
          System.out.println(G.getAlgorithm()+"       | "+G.getKeySize()
+"      | "+G.getKeyTimeFormatted()+" | "+G.getEncryptTimeFormatted()+"
| "+G.getDecryptTimeFormatted()+" | "+G.getSignTimeFormatted()+" |
"+G.getVerifyTimeFormatted()+" | "+G.getTotalTimeFormatted()+" |
"+G.getRepeat()+" | "+G.isMatch()+" | "+G.isVerified());
          G = new Generator ("ECIES", "secp160r1", plainText, 10);
          System.out.println(G.getAlgorithm()+"     | "+G.getKeySize()+"
| "+G.getKeyTimeFormatted()+" | "+G.getEncryptTimeFormatted()+" |
"+G.getDecryptTimeFormatted()+" | "+G.getSignTimeFormatted()+" |
"+G.getVerifyTimeFormatted()+" | "+G.getTotalTimeFormatted()+" |
"+G.getRepeat()+" | "+G.isMatch()+" | "+G.isVerified());
          G = new Generator ("T", "secp160r1", plainText, 10);
          System.out.println(G.getAlgorithm()+"         |
"+G.getKeySize()+" | "+G.getKeyTimeFormatted()+" |
"+G.getEncryptTimeFormatted()+" | "+G.getDecryptTimeFormatted()+" |
"+G.getSignTimeFormatted()+" | "+G.getVerifyTimeFormatted()+" |
"+G.getTotalTimeFormatted()+" | "+G.getRepeat()+" | "+G.isMatch()+" |
"+G.isVerified());
          G = new Generator ("T", "secp160k1", plainText, 10);
          System.out.println(G.getAlgorithm()+"         |
"+G.getKeySize()+" | "+G.getKeyTimeFormatted()+" |
"+G.getEncryptTimeFormatted()+" | "+G.getDecryptTimeFormatted()+" |
"+G.getSignTimeFormatted()+" | "+G.getVerifyTimeFormatted()+" |
"+G.getTotalTimeFormatted()+" | "+G.getRepeat()+" | "+G.isMatch()+" |
"+G.isVerified());
          System.out.println("");
          G = new Generator ("RSA", "2048", plainText, 10);
          System.out.println(G.getAlgorithm()+"       | "+G.getKeySize()
+"      | "+G.getKeyTimeFormatted()+" | "+G.getEncryptTimeFormatted()+"
| "+G.getDecryptTimeFormatted()+" | "+G.getSignTimeFormatted()+" |
"+G.getVerifyTimeFormatted()+" | "+G.getTotalTimeFormatted()+" |
"+G.getRepeat()+" | "+G.isMatch()+" | "+G.isVerified());
          G = new Generator ("ECIES", "secp224r1", plainText, 10);
          System.out.println(G.getAlgorithm()+"     | "+G.getKeySize()+"
| "+G.getKeyTimeFormatted()+" | "+G.getEncryptTimeFormatted()+" |
"+G.getDecryptTimeFormatted()+" | "+G.getSignTimeFormatted()+" |
"+G.getVerifyTimeFormatted()+" | "+G.getTotalTimeFormatted()+" |
"+G.getRepeat()+" | "+G.isMatch()+" | "+G.isVerified());
          G = new Generator ("T", "secp224r1", plainText, 10);
          System.out.println(G.getAlgorithm()+"         |
"+G.getKeySize()+" | "+G.getKeyTimeFormatted()+" |
"+G.getEncryptTimeFormatted()+" | "+G.getDecryptTimeFormatted()+" |
"+G.getSignTimeFormatted()+" | "+G.getVerifyTimeFormatted()+" |
"+G.getTotalTimeFormatted()+" | "+G.getRepeat()+" | "+G.isMatch()+" |
"+G.isVerified());
          G = new Generator ("T", "secp224k1", plainText, 10);
          System.out.println(G.getAlgorithm()+"         |
"+G.getKeySize()+" | "+G.getKeyTimeFormatted()+" |
"+G.getEncryptTimeFormatted()+" | "+G.getDecryptTimeFormatted()+" |
"+G.getSignTimeFormatted()+" | "+G.getVerifyTimeFormatted()+" |
"+G.getTotalTimeFormatted()+" | "+G.getRepeat()+" | "+G.isMatch()+" |
"+G.isVerified());
          System.out.println("");
          G = new Generator ("RSA", "7680", plainText, 1);
          System.out.println(G.getAlgorithm()+"       | "+G.getKeySize()
+"      | "+G.getKeyTimeFormatted()+" | "+G.getEncryptTimeFormatted()+"
| "+G.getDecryptTimeFormatted()+" | "+G.getSignTimeFormatted()+" |
"+G.getVerifyTimeFormatted()+" | "+G.getTotalTimeFormatted()+" |
"+G.getRepeat()+" | "+G.isMatch()+" | "+G.isVerified());
          G = new Generator ("AES", "192", plainText, 100);
          System.out.println(G.getAlgorithm()+"       | "+G.getKeySize()
+"       | "+G.getKeyTimeFormatted()+" | "+G.getEncryptTimeFormatted()
+" | "+G.getDecryptTimeFormatted()+" | "+G.getSignTimeFormatted()+" |
"+G.getVerifyTimeFormatted()+" | "+G.getTotalTimeFormatted()+" |
"+G.getRepeat()+" | "+G.isMatch()+" | "+G.isVerified());
          G = new Generator ("ECIES", "secp384r1", plainText, 10);
          System.out.println(G.getAlgorithm()+"     | "+G.getKeySize()+"
| "+G.getKeyTimeFormatted()+" | "+G.getEncryptTimeFormatted()+" |
"+G.getDecryptTimeFormatted()+" | "+G.getSignTimeFormatted()+" |
"+G.getVerifyTimeFormatted()+" | "+G.getTotalTimeFormatted()+" |
"+G.getRepeat()+" | "+G.isMatch()+" | "+G.isVerified());
          G = new Generator ("T", "secp384r1", plainText, 10);
          System.out.println(G.getAlgorithm()+"         |
"+G.getKeySize()+" | "+G.getKeyTimeFormatted()+" |
"+G.getEncryptTimeFormatted()+" | "+G.getDecryptTimeFormatted()+" |
"+G.getSignTimeFormatted()+" | "+G.getVerifyTimeFormatted()+" |
"+G.getTotalTimeFormatted()+" | "+G.getRepeat()+" | "+G.isMatch()+" |
"+G.isVerified());
          System.out.println("");
          G = new Generator ("RSA", "15360", plainText, 1);
          System.out.println(G.getAlgorithm()+"       | "+G.getKeySize()
+"     | "+G.getKeyTimeFormatted()+" | "+G.getEncryptTimeFormatted()+"
| "+G.getDecryptTimeFormatted()+" | "+G.getSignTimeFormatted()+" |
"+G.getVerifyTimeFormatted()+" | "+G.getTotalTimeFormatted()+" |
"+G.getRepeat()+" | "+G.isMatch()+" | "+G.isVerified());
          G = new Generator ("AES", "256", plainText, 100);
          System.out.println(G.getAlgorithm()+"       | "+G.getKeySize()
+"       | "+G.getKeyTimeFormatted()+" | "+G.getEncryptTimeFormatted()
+" | "+G.getDecryptTimeFormatted()+" | "+G.getSignTimeFormatted()+" |
"+G.getVerifyTimeFormatted()+" | "+G.getTotalTimeFormatted()+" |
"+G.getRepeat()+" | "+G.isMatch()+" | "+G.isVerified());
          G = new Generator ("ECIES", "secp521r1", plainText, 10);
          System.out.println(G.getAlgorithm()+"     | "+G.getKeySize()+"
| "+G.getKeyTimeFormatted()+" | "+G.getEncryptTimeFormatted()+" |
"+G.getDecryptTimeFormatted()+" | "+G.getSignTimeFormatted()+" |
"+G.getVerifyTimeFormatted()+" | "+G.getTotalTimeFormatted()+" |
"+G.getRepeat()+" | "+G.isMatch()+" | "+G.isVerified());
          G = new Generator ("T", "secp521r1", plainText, 10);
          System.out.println(G.getAlgorithm()+"         |
"+G.getKeySize()+" | "+G.getKeyTimeFormatted()+" |
"+G.getEncryptTimeFormatted()+" | "+G.getDecryptTimeFormatted()+" |
"+G.getSignTimeFormatted()+" | "+G.getVerifyTimeFormatted()+" |
"+G.getTotalTimeFormatted()+" | "+G.getRepeat()+" | "+G.isMatch()+" |
"+G.isVerified());
          System.out.println("");

      }

}
```

B.4 ECIESexample.java

(also includes ECDSA from Bouncy Castle)

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.Signature;
import java.security.SignatureException;
import java.security.spec.ECGenParameterSpec;
import javax.crypto.Cipher;
import org.bouncycastle.jce.spec.IEKeySpec;
import org.bouncycastle.jce.spec.IESParameterSpec;

public class ECIESexample {
    private SecureRandom random;
    private int keySize;
    private KeyPair akey;
    private KeyPair bkey;

      public ECIESexample () throws Exception{
            this.random = new SecureRandom();
      }

      public void establishKeys(String keysize) throws Exception {

          ECGenParameterSpec     ecGenSpec = new
ECGenParameterSpec(keysize);
          KeyPairGenerator keyGen = KeyPairGenerator.getInstance("EC");

          keyGen.initialize(ecGenSpec, random);

          this.akey = keyGen.generateKeyPair();
          this.bkey = keyGen.generateKeyPair();
            this.keySize = Integer.valueOf(
(ecGenSpec.getName().substring(4, 7)) ).intValue();
      }


      public byte[] encrypt(byte[] plainText) throws Exception {

          // get ECIES cipher objects
          Cipher acipher = Cipher.getInstance("ECIES");

          //  generate derivation and encoding vectors
        byte[]  d = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8 };
        byte[]  e = new byte[] { 8, 7, 6, 5, 4, 3, 2, 1 };
        IESParameterSpec param = new IESParameterSpec(d, e, 256);

        // encrypt the plaintext using the public key
          acipher.init(Cipher.ENCRYPT_MODE, new
IEKeySpec(akey.getPrivate(), bkey.getPublic()), param);
          return acipher.doFinal(plainText);
      }

      public byte[] decrypt(byte[] cipherText) throws Exception {

          // get ECIES cipher objects
          Cipher bcipher = Cipher.getInstance("ECIES");

          //  generate derivation and encoding vectors
        byte[]  d = new byte[] { 1, 2, 3, 4, 5, 6, 7, 8 };
        byte[]  e = new byte[] { 8, 7, 6, 5, 4, 3, 2, 1 };
        IESParameterSpec param = new IESParameterSpec(d, e, 256);

        // decrypt the text using the private key
          bcipher.init(Cipher.DECRYPT_MODE, new
IEKeySpec(bkey.getPrivate(), akey.getPublic()), param);
          return bcipher.doFinal(cipherText);
      }

      public byte[] sign(byte[] plainText) throws Exception {

            Signature sig = Signature.getInstance("SHA1WithECDSA");
            sig.initSign(akey.getPrivate());
            sig.update(plainText);
            return sig.sign();
      }

      public boolean verify(byte[] plainText, byte[] signature)
throws Exception {

            Signature sig = Signature.getInstance("SHA1WithECDSA");
            sig.initVerify(akey.getPublic());
            sig.update(plainText);
            try {
                  if (sig.verify(signature)) {
                        return true;
                  }     else return false;
            } catch (SignatureException se) {
                  System.out.println( "Signature failed" );
            }
            return false;
      }

      public int getKeySize() {
            return keySize;
      }

}
```

B.5 Generator.java

```
import java.text.DecimalFormat;

public class Generator {
      private double totalTime;
      private double keyTime;
      private double encryptTime;
      private double decryptTime;
      private double signTime;
      private double verifyTime;
      private String keySize;
      private boolean match;
      private boolean verified;
      private String algorithm;
      private double repeat;

      public Generator (String alg, String keysize, byte[] plainText,
double repeat) throws Exception {

            this.algorithm=alg;
            double ktime = 0;
            double etime = 0;
            double dtime = 0;
            double stime = 0;
            double vtime = 0;
            double delta = 0;
            this.match=true;
            this.verified=true;
            this.repeat = repeat;
            double DIV = 1000000000;

            if (alg == "RSA"){
                  for (int i=0; i<repeat; i++){
                      RSAexample Test = new RSAexample();

                      delta = System.nanoTime();
                      Test.establishKeys(keysize);
                      delta = (System.nanoTime() - delta)/DIV;
                      ktime += delta;

                      delta = System.nanoTime();
                      byte[] cipherText = Test.encrypt(plainText);
                      delta = (System.nanoTime() - delta)/DIV;
                      etime += delta;
                      delta = System.nanoTime();
                      byte[] decryptedText = Test.decrypt(cipherText);
                      delta = (System.nanoTime() - delta)/DIV;
                      dtime += delta;

                      delta = System.nanoTime();
                      byte[] signature = Test.sign(plainText);
                      delta = (System.nanoTime() - delta)/DIV;
                      stime += delta;

                      delta = System.nanoTime();
                      if ( !(Test.verify(plainText,signature)) ){
                        this.verified=false;
                      }

                      delta = (System.nanoTime() - delta)/DIV;
                      vtime += delta;

                        if (!(java.util.Arrays.equals(plainText,
decryptedText))){
                              this.match=false;
                        }
                  }

            } else if (alg == "ECIES"){
                  for (int i=0; i<repeat; i++){
                      ECIESexample Test = new ECIESexample();

                      delta = System.nanoTime();
                      Test.establishKeys(keysize);
                      delta = (System.nanoTime() - delta)/DIV;
                      ktime += delta;

                      delta = System.nanoTime();
                      byte[] cipherText = Test.encrypt(plainText);
                      delta = (System.nanoTime() - delta)/DIV;
                      etime += delta;

                      delta = System.nanoTime();
                      byte[] decryptedText = Test.decrypt(cipherText);
                      delta = (System.nanoTime() - delta)/DIV;
                      dtime += delta;

                      delta = System.nanoTime();
                      byte[] signature = Test.sign(plainText);
                      delta = (System.nanoTime() - delta)/DIV;
                      stime += delta;

                      delta = System.nanoTime();
                      if ( !(Test.verify(plainText,signature)) ){
                        this.verified=false;
                      }
                      delta = (System.nanoTime() - delta)/DIV;
                      vtime += delta;

                        if (!(java.util.Arrays.equals(plainText, 
decryptedText))){
                              this.match=false;
                        }
                  }

            } else if (alg == "AES"){
                  for (int i=0; i<repeat; i++){
                      AESexample Test = new AESexample();

                      delta = System.nanoTime();
                      Test.establishKeys(keysize);
                      delta = (System.nanoTime() - delta)/DIV;
                      ktime += delta;

                      delta = System.nanoTime();
                      byte[] cipherText = Test.encrypt(plainText);
                      delta = (System.nanoTime() - delta)/DIV;
                      etime += delta;

                      delta = System.nanoTime();
                      byte[] decryptedText = Test.decrypt(cipherText);
                      delta = (System.nanoTime() - delta)/DIV;
                      dtime += delta;

                      this.verified=false;
                      stime=0;
                      vtime=0;

                        if (!(java.util.Arrays.equals(plainText, 
decryptedText))){
                              this.match=false;
                        }
                  }
            } else if (alg == "T"){
                  for (int i=0; i<repeat; i++){
                      Texample Test = new Texample();

                      delta = System.nanoTime();
                      Test.establishKeys(keysize);
                      delta = (System.nanoTime() - delta)/DIV;
                      ktime += delta;

                      delta = System.nanoTime();
                      byte[] cipherText = Test.encrypt(plainText);
                      delta = (System.nanoTime() - delta)/DIV;
                      etime += delta;

                      delta = System.nanoTime();
                      byte[] decryptedText = Test.decrypt(cipherText);
                      delta = (System.nanoTime() - delta)/DIV;
```

```
                        dtime += delta;

                        delta = System.nanoTime();
                        byte[] signature = Test.sign(plainText);
                        delta = (System.nanoTime() - delta)/DIV;
                        stime += delta;

                        delta = System.nanoTime();
                        if ( !(Test.verify(plainText,signature)) ){
                          this.verified=false;
                        }

                        delta = (System.nanoTime() - delta)/DIV;
                        vtime += delta;

                          if (!(java.util.Arrays.equals(plainText,
decryptedText))){
                                this.match=false;
                          }
                  }
            } else {
                  this.keySize = "";
                  this.match = false;
                  this.verified = false;
                  this.algorithm = "NONE";
            }
            this.totalTime = (ktime + etime + dtime + stime +
vtime)/repeat;
            this.keyTime = ktime/repeat;
            this.encryptTime = etime/repeat;
            this.decryptTime = dtime/repeat;
            this.signTime = stime/repeat;
            this.verifyTime = vtime/repeat;
            this.keySize = keysize;

      }

      public String getTotalTimeFormatted() {
            DecimalFormat f = new DecimalFormat("#.#########");
            return f.format(totalTime);
      }

      public double getTotalTime() {
            return totalTime;
      }

      public String getKeyTimeFormatted() {
            DecimalFormat f = new DecimalFormat("#.#########");
            return f.format(keyTime);
      }

      public double getKeyTime() {
            return keyTime;
      }
```

```
	public String getEncryptTimeFormatted() {
		DecimalFormat f = new DecimalFormat("#.#########");
		return f.format(encryptTime);
	}

	public double getEncryptTime() {
		return encryptTime;
	}

	public String getDecryptTimeFormatted() {
		DecimalFormat f = new DecimalFormat("#.#########");
		return f.format(decryptTime);
	}
	public double getDecryptTime() {
		return decryptTime;
	}

	public String getSignTimeFormatted() {
		DecimalFormat f = new DecimalFormat("#.#########");
		return f.format(signTime);
	}

	public double getSignTime() {
		return signTime;
	}

	public String getVerifyTimeFormatted() {
		DecimalFormat f = new DecimalFormat("#.#########");
		return f.format(verifyTime);
	}

	public double getVerifyTime() {
		return verifyTime;
	}

	public String getKeySize() {
		return keySize;
	}

	public double getRepeat() {
		return repeat;
	}

	public boolean isMatch() {
		return match;
	}

	public boolean isVerified() {
		return verified;
	}

	public String getAlgorithm() {
		return algorithm;
```

```
        }
}
```

B.6 RSAexample.java

```
public class RSAexample {

        private String keySize;
        private KeyPair key;

        public RSAexample () throws Exception{

        }

        public void establishKeys(String keysize) throws Exception {

             KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");

             keyGen.initialize(Integer.parseInt(keysize));

             this.key = keyGen.generateKeyPair();
             this.keySize =
Integer.toString(((RSAKey)key.getPublic()).getModulus().bitLength());

        }

        public byte[] encrypt(byte[] plainText) throws Exception {

             // get an RSA cipher object and print the provider
             Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");

             // encrypt the plaintext using the public key
             cipher.init(Cipher.ENCRYPT_MODE, key.getPublic());
             return cipher.doFinal(plainText);

        }

        public byte[] decrypt(byte[] cipherText) throws Exception {

             // get an RSA cipher object and print the provider
             Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");

             // decrypt the text using the private key
             cipher.init(Cipher.DECRYPT_MODE, key.getPrivate());
             return cipher.doFinal(cipherText);
        }

        public byte[] sign(byte[] plainText) throws Exception {

               Signature sig = Signature.getInstance("SHA1WithRSA");
               sig.initSign(key.getPrivate());
```

```
            sig.update(plainText);
            return sig.sign();
      }

      public boolean verify(byte[] plainText, byte[] signature)
throws Exception {

            Signature sig = Signature.getInstance("SHA1WithRSA");
            sig.initVerify(key.getPublic());
            sig.update(plainText);
            try {
                  if (sig.verify(signature)) {
                        return true;
                  }     else return false;
            } catch (SignatureException se) {
                  System.out.println( "Signature failed" );
            }
            return false;
      }

      public String getKeySize() {
            return keySize;
      }

}
```

B.7 Texample.java

```
import java.security.SecureRandom;
import org.bouncycastle.math.ec.ECCurve;
import org.bouncycastle.math.ec.ECPoint;
import org.bouncycastle.math.ec.ECAlgorithms;
import java.math.BigInteger;
import java.security.*;


public class Texample {
      private int keySize;
    private SecureRandom random;
    private BigInteger priv;
    private ECPoint pub;
    private BigInteger n;
    private BigInteger p;
    private ECPoint G;
    private ECCurve curve;

      public Texample () throws Exception {
            this.random = new SecureRandom();
      }

      public void establishKeys(String seccurve){
            ApprovedCurves c = new ApprovedCurves(seccurve);
            this.n = c.getN();
            this.p = c.getP();
            this.G = c.getG();
            this.curve = c.getCurve();

        // dA, dB are private
        // QA, QB are public
        BigInteger dB = new BigInteger (n.bitLength()-1,
random).add(BigInteger.valueOf(1));
        ECPoint QB = G.multiply(dB);
            this.keySize = n.bitLength();
            this.priv = dB;
            this.pub = QB;
      }

      public byte[] encrypt(byte[] plainText) throws Exception {

            BigInteger rA=BigInteger.valueOf(0);
            do{
                  rA = new BigInteger (p.bitLength()-1, random);
            }while(rA.equals(BigInteger.valueOf(0)));
        ECPoint RA = G.multiply(rA);

        // pad plaintext...
        int padBytes = ((int)Math.ceil((double)p.bitLength()/8)*2 -
(plainText.length));
        byte[] paddedPlainText = new
byte[plainText.length+padBytes+1];
        java.util.Arrays.fill(paddedPlainText, (byte)0x00);
        try {
            // encoding
            paddedPlainText[0] = (byte)(0x04);
            System.arraycopy(plainText, 0, paddedPlainText,
paddedPlainText.length-plainText.length-1, plainText.length);
        } catch (Exception e) {
            System.err.println("Data must not be longer than "+
((p.bitLength()*2)-1)/8+" bytes");
                throw e;
        }

        int x = padBytes;
        paddedPlainText[paddedPlainText.length-1] = (byte)((x & 0xff));

        ECPoint CM1 = RA;
        // (QB*rA) + pt
        ECPoint CM2 =
(this.pub.multiply(rA).add(curve.decodePoint(paddedPlainText)));

        byte[] cipherText = new
byte[CM1.getEncoded().length+CM2.getEncoded().length];

        System.arraycopy(CM1.getEncoded(), 0, cipherText, 0,
CM1.getEncoded().length);
```

```
          System.arraycopy(CM2.getEncoded(), 0, cipherText,
CM1.getEncoded().length, CM2.getEncoded().length);
          return cipherText;
      }

      public byte[] decrypt(byte[] cipherText) throws Exception {

            byte[] cipherText1 = new byte[cipherText.length/2];
          System.arraycopy(cipherText, 0, cipherText1, 0,
cipherText.length/2);
            byte[] cipherText2 = new byte[cipherText.length/2];
          System.arraycopy(cipherText, cipherText.length/2, cipherText2,
0, cipherText.length/2);

            ECPoint CM1 = (curve.decodePoint(cipherText1));
            ECPoint CM2 = (curve.decodePoint(cipherText2));

            // pt = pt + (QB*rA) - RA*dB
            // QB*rA = RA*dB
          ECPoint Pm = CM2.subtract(CM1.multiply(this.priv));
          byte[] paddedDecryptedText = Pm.getEncoded();

          // strip decryptedtext
          // cut first byte specifying the encoding...
        byte[] decryptedText = new byte[paddedDecryptedText.length-1-
(paddedDecryptedText[paddedDecryptedText.length-1] & 0xff)];
        System.arraycopy(paddedDecryptedText,
paddedDecryptedText.length-decryptedText.length-1, decryptedText, 0,
decryptedText.length);
        return decryptedText;
      }

      public byte[] sign(byte[] plainText) throws Exception {

            BigInteger rA=BigInteger.valueOf(0);
            do{
                  rA = new BigInteger (p.bitLength()-1, random);
            }while(rA.equals(BigInteger.valueOf(0)));
            ECPoint RA = G.multiply(rA);

            BigInteger r = RA.getX().toBigInteger().mod(n);

        MessageDigest messageDigest = MessageDigest.getInstance("SHA-
1");
        messageDigest.update(plainText);
        BigInteger m = new BigInteger(messageDigest.digest());
        // (rA^-1)*(m+(priv*r))
        BigInteger s =
((rA.modInverse(n).multiply(m.add(priv.multiply(r)))));
        byte[] sbyte = s.toByteArray();
        byte[] rbyte = r.toByteArray();
        int maxlength;
        int roffset;
        int soffset;
        if (rbyte.length>sbyte.length){
            maxlength=rbyte.length;
            soffset=rbyte.length-sbyte.length;
            roffset=0;

        } else {
            maxlength=sbyte.length;
            roffset=sbyte.length-rbyte.length;
            soffset=0;
            }
        byte[] signature = new byte[maxlength*2];
        System.arraycopy(rbyte,0,signature,roffset,maxlength-roffset);
        System.arraycopy(sbyte,0,signature,maxlength+soffset,maxlength-
soffset);
        return signature;
      }

      public boolean verify(byte[] plainText, byte[] signature)
throws Exception {

        MessageDigest messageDigest = MessageDigest.getInstance("SHA-
1");
        messageDigest.update(plainText);
        BigInteger m = new BigInteger(messageDigest.digest());
            byte[] rbyte = new byte[signature.length/2];
          System.arraycopy(signature, 0, rbyte, 0, signature.length/2);
            byte[] sbyte = new byte[signature.length/2];
          System.arraycopy(signature, signature.length/2, sbyte, 0,
signature.length/2);
          BigInteger r = new BigInteger(rbyte);
          BigInteger s = new BigInteger(sbyte);
          ECPoint rs = ECAlgorithms.sumOfTwoMultiplies(G,
((s.modInverse(n)).multiply(m)).mod(n), pub,
((s.modInverse(n)).multiply(r)).mod(n));
          BigInteger rp = rs.getX().toBigInteger().mod(n);
          return (rp.equals(r));
      }

      public int getKeySize() {
            return keySize;
      }

}
```