AN ABSTRACT OF THE THESIS OF

Rhea Stadick for the degree of Honors Baccalaureate of Science in Computer Science presented on June 06, 2005. Title: A Java Implementation of the Elliptic Curve Digital Signature Algorithm Using NIST Curves Over GF(p).

Abstract approved:

_____

Çetin K. Koç

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a public key cryptosystem used for creation and verification of digital signatures in electronic documents. In this thesis, we created a Java applet that provides the functionality of the ECDSA using all of the NIST elliptic curves over GF(p). This applet was embedded into a website and tailored for public use across an online network. We show that Java can provide an effective platform for the ECDSA in combination with imported packages and classes. The implementation details of our ECDSA applet are discussed, followed by an analysis of the code functions.

A Java Implementation of the Elliptic Curve Digital Signature Algorithm Using NIST Curves Over GF(p)

by

Rhea Stadick

A THESIS

submitted to

Oregon State University

University Honors College

In partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science (Honors Scholar)

Presented June 06, 2005
Commencement June 2006

Honors Baccalaureate of Science in Computer Science thesis of Rhea Stadick presented on June 06, 2005.

APPROVED:

_____

Mentor, representing Electrical Engineering and Computer Science

_____

Committee Member, representing Mathematics

_____

Committee Member, representing Electrical Engineering and Computer Science

_____

Chair, Department of Electrical Engineering and Computer Science

_____

Dean, University Honors College

I understand that my thesis will become part of the permanent collection of Oregon State University, University Honors College. My signature below authorizes release of my project to any reader upon request.

_____

Rhea Stadick, Author

ACKNOWLEDGEMENTS

I would like to give special thanks to Dr. Koç for his guidance and sponsorship. I would also like to thank Dr. Schmidt and Colin Van Dyke for taking the time to participate in my honors thesis committee.

I also thank Avantika Mathur for her support, patience, and friendship during the course of this project.

Rhea Stadick

Corvallis, OR

TABLE OF CONTENTS

Page

1. INTRODUCTION 1

2. INTRODUCTION TO CRYPTOGRAPHY 3

2.1. SYMMETRIC CRYPTOSYSTEMS 3

2.2. ASYMMETRIC CRYPTOSYSTEMS 4

2.3. PUBLIC KEY CRYPTOGRAPHY 4

2.3.1. Discrete Logarithm Problem 5

2.3.2. Elliptic Curve Discrete Logarithm Problem 6

2.4. ELLIPTIC CURVE CRYPTOGRAPHY 7

2.4.1. Utilization of Elliptic Curve Cryptography 7

2.4.2. Elliptic Curves Over Finite Fields 7

2.4.3. Field Overview of GF(2^k) 8

2.4.4. Field Overview of GF(p) 9

2.4.5. Arithmetic Operations on Elliptic Curves over GF(p) 9

2.4.5.1. Addition of Distinct Points 10

2.4.5.2. Doubling of a Point 10

2.4.5.3. Scalar Multiplication of Points 11

2.5. DIGITAL SIGNATURES 11

2.5.1. Functionality and Intent of the Digital Signature Algorithm 12

2.5.2. Signature Generation Using DSA 12

2.5.3. Signature Verification Using DSA 13

2.6. ECDSA 14

2.6.1. Security of ECDSA 14

2.6.2. Key Pair Generation Using ECDSA 15

2.6.3. Signature Generation Using ECDSA 15

2.6.4. Signature Verification Using ECDSA 16

3. JAVA APPLET IMPLEMENTATION OF ECDSA 17

3.1. JAVA OVERVIEW 17

3.2. ECDSA APPLET USAGE 19

3.2.1 Elliptic Curve Selection 21

3.2.2 Applet Key Generation 21

3.2.3 Applet Signature Generation 22

3.2.4 Applet Signature Verification 22

3.2.5 Java Classes Used 22

3.3. IMPEMENTATION DETAILS 23

3.3.1. Packages Used 24

3.3.2. Global Curve Values 26

3.3.3. Applet User Interface and Components 26

3.3.4. ItemListeners and ActionListeners 27

3.3.4.1. The actionPerformed() Method 27

3.3.5. Verify, Sign, and Arithmetic Method Summaries 30

3.3.6. Additional Elliptic Curve Classes 33

3.3.6.1. ECPoint Class 33

TABLE OF CONTENTS (Continued)

3.3.6.2. EllipticCurve Class 35
3.3.7 Class Function Flow 35

3.4. ANALYSIS AND IMPROVEMENTS **38**
3.4.1. Timing of Functions 38
3.4.2. Runtime Analysis 40
3.4.3. Suggested Improvements 40

4. CONCLUSION ***41***

5. BIBLIOGRAPHY ***43***

APPENDICES ***44***

LIST OF TABLES

Table **Page**

Table 1: ECDSA Imported Packages 25

Table 2: Summary of handling applet events 29

Table 3: ECDSA method summaries 32

Table 4: ECPoint class constructor summary 34

Table 5: ECPoint class method summaries 34

Table 6: EllipticCurve class constructor summary 35

Table 7: EllipticCurve class method summaries 35

Table 8: Average function runtimes by key size 39

LIST OF FIGURES

Figure **Page**

Figure 1: Applet interface 20

Figure 2: Flow of function usage 37

Figure 3: Depiction of ECDSA method runtimes as a function of the key size 39

LIST OF APPENDICES

Appendix **Page**

A. Timing results of function analysis 45

B. NIST curve values over GF(p) 46

C. ECDSA applet source code 48

A Java Implementation of the Elliptic Curve Digital Signature Algorithm Using NIST Curves Over GF(p)

1. INTRODUCTION

Digital transactions have become commonplace, and in some cases inextricably linked to modern life. This technological dependency requires that information be unaltered and confidential. Before the age of web environments, large distributed networks, and the proliferation of electronic data, cryptosystems required only symmetric algorithms. Individuals could maintain the privacy of information with one secret key. With the advent of the Internet era, many obstacles to trust and security appeared. Information security concerns were introduced such as proving the identity of an individual, identifying unwanted modification of data, and securing information without complex key exchanges. The invention of asymmetric algorithms in the mid-1970's provided the answers to these issues.

Public key cryptosystems based on asymmetric design are widely used today and many employ the intractability of the discrete logarithm problem. Other public key cryptosystems are based on the computational complexity of the elliptic curve discrete logarithm problem. These cryptosystems are known as elliptic curve cryptography (ECC) which utilizes elliptic curves defined over prime and binary finite fields GF(p) and $GF(2^k)$, respectively.

Despite its recent emergence, elliptic curve cryptography is being studied as a viable replacement for well-known and implemented public key cryptosystems. The hard

problem that ECC is based on gives it a greater strength-per-key-bit making it ideal for resource-constrained systems. The Elliptic Curve Digital Signature Algorithm (ECDSA), the elliptic curve analogue of the Digital Signature Algorithm (DSA), is a popular example of this. ECDSA combines the additional information security services of non-repudiation, authenticity, and integrity that digital signatures offer, with greater protection for a given key size compared with DSA.

This thesis focuses on the implementation of ECDSA in a publicly available format. In order to satisfy this ease of use criterion, we chose to create a Java applet where anyone can perform the three components of ECDSA – key generation, signature generation, and signature verification. We chose to work over all prime curves that are recommended by the National Institute of Science and Technology (NIST). This provides users with the ability to select their preferred level of security when digitally signing a message.

2. INTRODUCTION TO CRYPTOGRAPHY

Cryptography has been an important tool in securing information transactions for thousands of years. It was originally intended to disguise messages so that adversaries could not acquire or alter sensitive information. Cryptography involves encryption, which is the conversion of plaintext to an unreadable format known as ciphertext. If the encryption is secure, only an entity with the secret key can decrypt the ciphertext and convert it back to plaintext. Since its origination, cryptography has been divided to include two types of cryptosystems: symmetric and asymmetric. These terms define whether the cryptosystem uses a single key or a pair of keys for encryption and decryption. Cryptography has also been expanded to provide the following information security requirements:

1. **Non-repudiation**: Preventing an entity from denying previous commitments or actions.
2. **Integrity**: Ensuring no unauthorized alteration of data.
3. **Authentication**: Verifying an entity's identity
4. **Confidentiality**: Protecting the data from all but the intended receiver.

2.1. SYMMETRIC CRYPTOSYSTEMS

Symmetric algorithms share the same key for encryption and decryption. Keeping this single key private is essential to the security of the algorithms. These traditional cryptosystems suffer from many disadvantages including key exchange and key

management (when large groups must manage many pairs of keys). Despite this security risk, they do offer the advantages of having a small key length and consuming a low amount of computing power. Examples of well-known symmetric algorithms include the Data Encryption Algorithm (DEA) defined by the Data Encryption Standard (DES), and Triple-DES.

2.2. ASYMMETRIC CRYPTOSYSTEMS

Asymmetric cryptosystems are better suited for real-world environments where the secure transfer of the secret key is not essential to the security of the system. These algorithms use a pair of keys – one private key typically used for decryption and one public key typically used for encryption. This allows for anyone who knows an entity's public key to send them an encrypted message with assurance that only that entity can decrypt it with their secret private key. The security and utility of public key cryptosystems is aided by their asymmetric design. Well-known asymmetric algorithms include DSA, RSA, and ELGAMAL.

2.3. PUBLIC KEY CRYPTOGRAPHY

The proposal of public key cryptosystems in 1976 by Whitfield Diffie and Martin Hellman introduced a revolutionary way to address modern security issues such as key management, authentication, non-repudiation, and signatures in a digital environment.

All cryptosystems are secure only if the difficulty of the mathematical problem that they are based on is determined to be hard. Public key cryptosystems are based on the intractability of one of three problems. These problems and the cryptosystems based on them are:

1. The Integer Factorization Problem; RSA
2. The Discrete Logarithm Problem; DSA, Diffie-Hellman, ElGamal
3. The Elliptic Curve Discrete Logarithm Problem; ECDSA, ECDH

The discrete logarithm problem (DLP) and the elliptic curve discrete logarithm problem (ECDLP) are discussed in the following sections.

2.3.1. Discrete Logarithm Problem

In 1985, Taher ElGalmal first proposed the use of the DLP for public-key cryptosystems and digital signature schemes. The discrete logarithm problem, like the factoring problem, is said to be difficult in addition to being the hard direction of a one-way function [9]. It involves the mathematical structure of groups, of which the simplest definition is a nonempty set S together with a binary operation *. Let g be an element of group G. If the order n of element g is also the order of the group G, then g is referred to as a generator of G. This means that repeated exponentiation of g ($g*g*g$) will yield all elements of G. Cryptography commonly uses the group Z_p*, a finite and multiplicative group of integers modulo a prime number p. The discrete logarithm problem is characterized by these elements under the following condition:

1. Given a generator g of the multiplicative group Z_p^*, a prime integer p, and another element $h \in Z_p^*$, find the unique integer **j** in the interval [0, p-1] such that $h \equiv g^j$ **(**mod p**)**.

The inverse operation of exponentiation on g^j mod p can be computed efficiently, but it is not practical to calculate g^j mod p for large values. This is why the DLP is a one-way function and considered a hard problem.

2.3.2. Elliptic Curve Discrete Logarithm Problem

Based on the infeasibility of computing discrete logs on elliptic curves over finite fields, the Elliptic Curve Discrete Logarithm Problem is the security behind elliptic curve cryptography. Cryptosystems based on the computational complexity of this mathematical problem include ECDSA, ECElGamal, and Elliptic Curve Diffie-Hellman (ECDH). ECDLP is similar to the aforementioned Discrete Logarithm Problem and can be thought of as an analogue to DLP. In the ECDLP, however, the subgroup Z_p^* is replaced by the group of points on an elliptic curve over a finite field. In addition, unlike the DLP and the integer factorization problem, no subexponential-time algorithm is known for the ECDLP. ECDLP is considered to be significantly harder than the DLP, thus giving elliptic curve cryptosystems a greater strength-per-key-bit than their non-analogue discrete logarithm counterparts.

2.4. ELLIPTIC CURVE CRYPTOGRAPHY

Although elliptic curves have been studied for over 150 years, they weren't applied to cryptography until very recently. In 1985, Neal Koblitz and Victor Miller independently proposed the use of elliptic curve points over a finite field for cryptosystems. Since then, several standards have accepted the use of elliptic curve cryptosystems. Interest in elliptic curve cryptography is due to the determination that is based on a harder mathematical problem than other cryptosystems. Because of this, ECC can offer certain advantages over non-elliptic curve schemes. For effective use in cryptosystems, elliptic curves are defined over finite fields that require unique arithmetic operations.

2.4.1. Utilization of Elliptic Curve Cryptography

Elliptic curve cryptography's strengths make it most suitable for resource-constrained systems. ECC provides greater security for a given key size and can be efficiently and compactly implemented. These attributes make it well suited for systems with constraints on processor speed, security, heat production, power consumption, bandwidth, and memory. Cell phones, PDAs, wireless devices, laptops, and smartcards are applications that benefit from elliptic curve cryptosystems.

2.4.2. Elliptic Curves Over Finite Fields

Elliptic curve cryptography uses elliptic curves to find points on an ellipse. These points are used for keys within a discrete range. This means that all numbers must be finite and

positive integers from 0 to a specified number used. The field constraints of elliptic curves used in cryptosystems prevent problems such as infinitely repeating fractions and round-off errors.

Of the different fields that elliptic curves reside in, GF(p) prime fields and GF(2^k) binary polynomial fields are most effective for cryptographic implementations. Of the two, GF(2^k) is more popular due to available space and time-efficient implementations of elliptic curve arithmetic in GF(2^k).

2.4.3. Field Overview of GF(2^k)

The characteristic two finite field, F(2^k), referred to as a *binary field*, contains 2^k elements for the degree *k* of the field. Bit strings of length *k* make up the elements of the field. All arithmetic in F(2^k) is done in terms of operations on the bits. A basis, either polynomial or normal, is chosen to interpret the bit strings. The correct arithmetic operations are then chosen depending on which basis is used.

The elliptic curve group over GF(2^k) consists of the point at infinity ∞, and all points (x,y), where $x,y \in F(2^k)$, that satisfy the following equation of the form

$$y^2 + xy = x^3 + ax^2 + b,$$

where $b \neq 0$ and $a,b \in F(2^k)$.

2.4.4. Field Overview of GF(p)

The finite field F(p), referred to as a *prime field*, consists of integers in the interval [1,*p*-1] where *p* is a prime number. The arithmetic of this field is modulo *p* so that any calculation results will fall into the finite space defined. An elliptic curve over Galois field GF(p) is defined by the following equation:

$$y^2 = x^3 + ax + b$$

In order to form a group on the elliptic curve consisting of integers, $x^3 + ax + b$ can contain no repeated factors. This can be guaranteed by ensuring that $4a^3 + 27b^2 \pmod{p} \neq 0$. All points (x,y) that satisfy the equation $y^2 = x^3 + ax + b$ where $x,y \in$ GF(p) , together with an extra point ∞, create a group on the curve.

2.4.5. Arithmetic Operations on Elliptic Curves over GF(p)

Inversion on the field, denoted as a^{-1}, is performed modulo *p* such that $(a^{-1}*a) \bmod p = 1$. Addition of points within an elliptic curve group will give another point on the curve, and all multiples of points within the group are also contained on the elliptic curve. There are three rules that addition of points within an elliptic curve group adhere to:

1. $\infty + \infty = \infty$

2. $(x,y) + \infty = (x,y)$

3. $(x,y) + (x, -y) = \infty$

Aside from these rules, addition, doubling, and scalar multiplication of points on elliptic curves over GF(p) are calculated by the following formulas.

2.4.5.1. Addition of Distinct Points

Let two points on the curve $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ and their sum are $R = (x_3, y_3)$. P and Q are distinct if P and $-Q$ are not the same ($x_1 \neq x_2$). The addition of the points, $P + Q = R$ is defined as:

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$$

$$\lambda = (y_2 - y_1)(x_1 - x_1)^{-1}$$

$$x_3 = \lambda^2 - x_1 - x_2$$

$$y_3 = \lambda(x_1 - x_3) - y_1$$

2.4.5.2. Doubling of a Point

Let point $P = (x_1,x_2)$ exist on the curve where $x_1 \neq 0$. The doubling of the point, $2P = R$ is defined as:

$$(x_1, y_1) + (x_1, y_1) = (x_3, y_3)$$

$$\lambda = (3x_1^2 + a)(2y_1)^{-1}$$

$$x_3 = \lambda^2 - 2x_1$$

$$y_3 = \lambda(x_1 - x_3) - y_1$$

2.4.5.3. Scalar Multiplication of Points

Let *P* be a point and *d* be a bit string representation of an integer. In order to compute point $Q = dP$, addition chains are used. A common addition chain is the binary method, which uses addition and doubling of points. The binary method for scalar multiplication of a point, $dP = Q$, follows this algorithm (depicted in pseudocode);

```
if d_{n-1} = 1, then Q := P else Q: = ∞
for i = n-2 to 0
        Q := Q + Q
        if d_i = 1 then Q:= Q + P
return Q
```

2.5. DIGITAL SIGNATURES

Digital Signatures are used as a common and effective tool in information security. They can be thought of as a digital counterpart to handwritten signatures, but they offer much more as an encryption technique. The use of digital signatures can offer a high degree of protection for many digital transactions and documents. A digital signature satisfies the information security requirements of non-repudiation, integrity, and authentication.

2.5.1. Functionality and Intent of the Digital Signature Algorithm

The Digital Signature Standard (DSS) as defined by NIST (FIPS 186) in 1994, specifies DSA as an accepted algorithm to generate and verify digital signatures. DSA is an asymmetric encryption standard whose basic components are key generation, signature generation, and signature verification.

According to the DSS, the purpose of the Digital Signature Algorithm is to provide the capability of generating and verifying signatures to the extent that the identity of the signatory and the integrity of the data can be verified.

2.5.2. Signature Generation Using DSA

During the first stage of signature generation, the data message is condensed into what is called a message digest using a secure one-way hash function specified by the Secure Hash Standard (SHS), FIPS 180. This hash algorithm provides another layer of security because it is not possible to reverse the hash and determine its contents. The integrity of the data is also maintained because a hacker cannot alter the message even by a single character without invalidating the signature.

The DSA sign operation is then performed on the resulting message digest $H(M)$ using the signatory's private key a and the established public key (p, q, α, y). This returns a secure digital signature that can be used to authenticate and ensure integrity of the message. Following are examples of signature generation by sender A and signature verification by receiver B of a message.

DSA signature generation

Sender A computes the signature of message M**:**

1. Select random integer k where $1 \leq k \leq q$-1
2. Compute $r = (\alpha^k \bmod p) \bmod q$
3. Compute $k^{-1} \bmod q$.
4. Compute $s = k^{-1}\{H(M) + ar\} \bmod q$
5. If $s = 0$, return to step 1
6. The signature for message M consists of the computed integers (r,s)

2.5.3. Signature Verification Using DSA

During verification, the original message is received and then, applying the same secure hash function used during signing, the message digest is obtained. The public key, message digest and the provided digital signature are then used with the DSA verify operation to determine authenticity of the signature.

DSA signature verification

Receiver B performs the following steps to verify the A's signature on message M.

1. Obtain the public key (p, q, α, y) of sender A.
2. Verify that $0 < r < q$ and $0 < s < q$, otherwise reject the signature
3. Compute $w = s^{-1} \bmod q$ and $H(M)$
4. Compute $u1 = wH(M) \bmod q$ and $u2 = rw \bmod q$

5. Compute $v = (\alpha^{u1} y^{u2} \bmod p) \bmod q$
6. The signature of A for message M is verified by receiver B if $v = r$.

2.6. ECDSA

The elliptic curve digital signature algorithm is the elliptic curve analogue of DSA and serves the same purposes of key generation, signature generation, and signature verification. ECDSA was first proposed in 1992 by Scott Vanstone in response to NIST's proposal of DSS. It was later accepted in 1998 as an ISO standard (ISO 14888-3), as an ANSI standard (ANSI X9.62) in 1999, and as an IEEE standard (IEEE 1363-2000) and as a NIST standard (FIPS 186-2) in 2000.

2.6.1. Security of ECDSA

The generation of the public key in ECDSA involves computing the point, Q, where $Q = dP$. In order to crack the elliptic curve key, adversary Eve would have to discover the secret key d. Given that the order of the curve E is a prime number n, then computing d given dP and P would take roughly $2^{n/2}$ operations [1]. For example, if the key length n is 192 bits (the smallest key size that NIST recommends for curves defined over GF(p)), then Eve will be required to compute about 2^{96} operations. If Eve had a super computer and could perform one billion operations per second, it would take her around two and a half trillion years to find the secret key. This is the elliptic curve discrete logarithm problem behind ECDSA.

2.6.2. Key Pair Generation Using ECDSA

Let A be the signatory for a message *M*. Entity A performs the following steps to generate a public and private key:

1. Select an elliptic curve *E* defined over a finite field F_p such that the number of points in $E(F_p)$ is divisible by a large prime *n*.
2. Select a base point, *P*, of order n such that $P \in E(F_p)$
3. Select a unique and unpredictable integer, *d*, in the interval [1, *n*-1]
4. Compute $Q = dP$
5. Sender A's private key is *d*
6. Sender A's public key is the combination (*E*, *P*, *n*, *Q*)

2.6.3. Signature Generation Using ECDSA

Using A's private key, A generates the signature for message *M* using the following steps:

1. Select a unique and unpredictable integer *k* in the interval [1,*n*-1]
2. Compute $kP = (x_1,y_1)$, where x_1 is an integer
3. Compute $r = x_1 \bmod n$; If $r = 0$, then go to step 1
4. Compute $h = H(M)$, where *H* is the Secure Hash Algorithm (SHA-1)
5. Compute $s = k^{-1}\{h + dr\} \bmod n$; If $s = 0$, then go to step1
6. The signature of A for message *M* is the integer pair (*r*,*s*)

2.6.4. Signature Verification Using ECDSA

The receiver B can verify the authenticity of A's signature (r,s) for message M by performing the following:

1. Obtain signatory A's public key (E, P, n, Q)
2. Verify that values r and s are in the interval [1,n-1]
3. Compute $w = s^{-1} \bmod p$
4. Compute $h = H(M)$, where H is the same secure hash algorithm used by A
5. Compute $u_1 = hw \bmod n$
6. Compute $u_2 = rw \bmod n$
7. Compute $u_1P + u_2Q = (x_0,y_0)$
8. Compute $v = x_0 \bmod n$
9. The signature for message M is verified only if $v = r$

3. JAVA APPLET IMPLEMENTATION OF ECDSA

Taking advantage of the platform-independence and web integration that Java applets offer, we created a publicly accessible website where anyone can perform the functions of the Elliptic Curve Digital Signature Algorithm. In this way, ECDSA is shown to be easily implemented in a context where it is most useful – in a network environment.

3.1. JAVA OVERVIEW

The Java platform and programming language was first announced only a decade ago in 1995 by Sun Microsystems. Java helped realize the impact of the network age by providing a platform-independent way to integrate programming and the web. Applets, which can easily be embedded into websites, are a key link in this integration. With the growth of the Internet, Java has become the language of choice in web application development and enterprise solutions that require component communication over distributed systems.

Networking and e-commerce provide a fast and global medium in which communications and transactions can take place rapidly and efficiently. Unfortunately, the public nature of networks has made them highly susceptible to cyber attacks and fraud. The importance of maintaining uncompromised sites on the Internet has become a major issue for individuals, government agencies, and commercial organizations. These entities have much more to lose if their sites are compromised due to their dependency on the web, and there is no shortage of adversarial attacks on them. Many public key

cryptosystems have found a natural niche by filling these network security voids, and have since become essential to data protection in e-commerce and communications.

While Java has always been designed with security in mind, the release of Java Development Kit (JDK) 1.4 now integrates cryptographic functionality into its core package. This release introduces the Java Cryptography Extension (JCE), a set of packages that provide a framework and implementations for encryption (asymmetric, symmetric, block and stream ciphers), key generation and agreement, and Message Authentication Code (MAC) algorithms. This functionality is provided on top of implementations and interfaces for digital signatures and message digests that already exist in the Java 2 platform. Various algorithms are supported such as DES, AES, and the Diffie-Hellman key agreement. In addition, JDK 1.4 also provides Java Authentication and Authorization Services (JAAS), and Java Secure Socket Extension (JSSE). In version 1.5, elliptic curves and elliptic point classes help support elliptic curve cryptosystems. While these solutions exist outside of Java, they are not offered in a compact, fully integrated, and portable package that the newest versions of JDK offer. The JCE has also been designed to remove the math and complexity from the cryptographic algorithms to allow for a faster and more accurate distributed system development time.

The Internet has a long way to go in solving its security problems. The use of cryptography in conjunction with a low-cost, portable, network-friendly, and platform-independent language such as Java can be a valuable solution to many of these security issues.

3.2. ECDSA APPLET USAGE

The purpose of using a Java applet is to provide a familiar and easily accessible medium for users to sign and verify messages using the elliptic curve digital signature algorithm. By using a Java applet, our implementation can be embedded into a website and made available for all public use. The ECDSA applet contains three parts: key generation, signature generation and signature verification. Let two entities, Alice and Bob, be users of our applet. The user interface is show below, followed by an outline of typical usage of the applet.

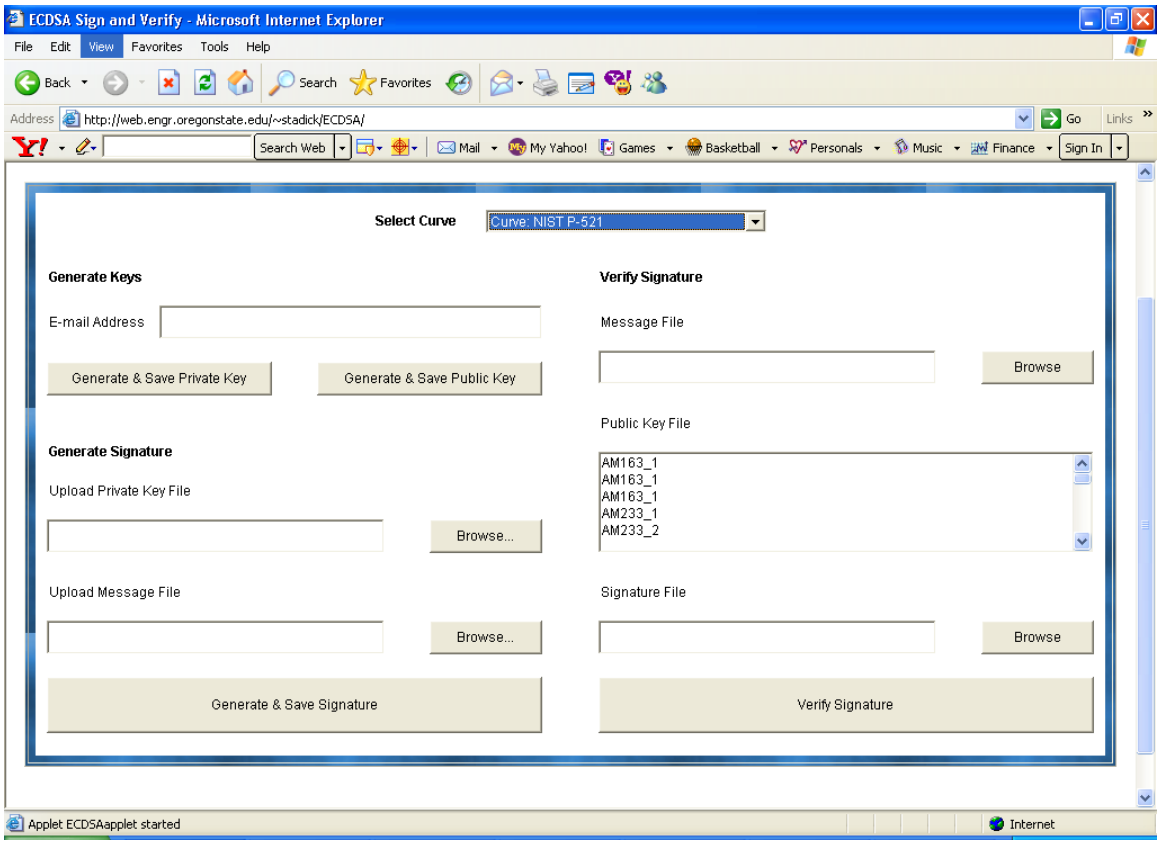

Figure 1: Applet interface

3.2.1 Elliptic Curve Selection

The first step in using the applet is to specify the elliptic curve that all ECDSA arithmetic will be performed over. The ECDSA applet allows selection of five different curves defined over GF(p). The user has the option of selecting one of the following NIST curves via a dropdown list: P-192, P-224, P-256, P-384, and P-512. The names of these curves reference the bit length of the keys and are listed by increasing strength. NIST curves are used because they are recommended as a secure standard under FIPS 186-2.

3.2.2 Applet Key Generation

After selecting an elliptic curve defined over a given field, the ECDSA applet provides the ability to generate a private and public key. First, the user must enter their email address. What the user enters will be their unique identifier. Their public key will be publicly available on the site and listed under this identifier. In addition, their private key will be saved in a file named after their ID. For example, let Alice enter her email address as *Alice@ecdsa*. Alice uses the applet to generate a private key. Her private key will be saved on her local file system under the name *Alice@ecdsa.privateKey*. Alice then generates her public key. Her public key will be saved on a server repository. This allows the applet to store her public key and make it available for all users. In order to retrieve Alice's public key, a user must know Alice's email address. They can then select this address in the Signature Verification portion of the applet when it is necessary to verify Alice's signature for a given message.

3.2.3 Applet Signature Generation

After the user has created both a private key and public key they can generate a signature for any message that is stored on their local file system. The user must upload their private key and message to be signed. The applet then generates a signature for the uploaded message. The signature is saved on the user's local file system under their specified email address name. If Alice has performed these steps, she can send Bob her message with her signature to ensure authenticity.

3.2.4 Applet Signature Verification

If Bob knows Alice's email address, then he can verify her signature on a given message *m*. First, Bob uploads message *m* to the applet. Knowing Alice's email address, he can select it from the list of user's public keys stored on the server repository. He then uploads Alice's signature for message *m*. The applet determines if the signature is valid given Alice's public key. A message is displayed that the signature is valid or invalid.

3.2.5 Java Classes Used

For accuracy purposes, existing Java classes were used whenever possible to perform computations of the large integers. The mathematical operations involved in point arithmetic were manually coded because there are no available Java methods to perform these computations. The algorithms used during implementation for point addition, doubling of a point, and scalar multiplication are discussed in section 2.4.5.

For the characteristic mathematical operations of the primary field, the Java *BigInteger* class was used. The *BigInteger* class contains functions for modular inversion, addition, multiplication, and modular arithmetic of large integers. In addition, the *BigInteger* method *SecureRandom* was utilized in order to securely generate the large random numbers necessary for key generation and signature generation.

To create the secure hash of a message, we used the Java *Security* class, and specifically the *MessageDigest* functions. Although SHA-1 is the only hash function defined for use with ECDSA by NIST, we elected to use SHA-512 in our implementation. During the course of programming the applet, SHA-1 was believed to be broken. Due to the newly introduced security risk of this hash algorithm and the announcement by NIST to phase out SHA-1 in favor of larger and stronger hash functions (SHA-224, SHA-256, SHA-384, and SHA-512), we felt that SHA-512 should be used [7]. This does not compromise the security of the ECDSA implementation.

3.3. IMPEMENTATION DETAILS

The ECDSA Applet is composed of one main class: the *ECDSA* class. This class implements the applet and performs all of the arithmetic computations and ECDSA functionality that the user requests when signing a message on the applet or verifying a signature. The user can select from a list of curves with various key sizes to sign their message. The source code for the ECDSA applet can be found in Appendix C. Following is a summary of the functions and packages used to implement the *ECDSA* class.

3.3.1. Packages Used

The table that follows is the list of packages and the specific classes imported and used in the *ECDSA* class, followed by a short description of their purpose.

Imported Packages	
java.applet	Necessary to create the applet and allow the applet to communicate with the website.
java.awt	Contains classes necessary for creating the user interface and components.
java.awt.event	Provides functionality to handle usage of the applet by users.
java.math.BigInteger	Contains classes that allow for arithmetic computations of very large numbers.
java.io	Allows for input and output from the applet to the user's file system or host server.
java.io.DataInputStream	Provides ability to write data to file on the host server in stream form.
java.io.IOException	Necessary to catch in case of I/O error.
java.util.StringTokenizer	Used to parse through strings.
java.io.UnsupportedEncodingException	Must be thrown when creating the message digest.
java.net	Used when communicating with host server via a URL.
import java.security	Contains security classes in the Java framework. Used in creating the message digest and secure random numbers.
import javax.swing	Provides a user-friendly way to access and browse local file system.

Table 1: ECDSA Imported Packages

3.3.2. Global Curve Values

The parameters for the NIST elliptic curves over GF(p) can be found in Appendix B. These parameters include the order *r*, base point coordinates (*x*,*y*) (where $x,y \in GF(p)$ and the point is of order *r*), the n-bit prime modulus *p*, the coefficient *b*, and the coefficient *a* (which always has a value of –3 for efficiency purposes). All of these parameters satisfy the equation $y^2 = x^3 + ax + b$. These parameters were hard-coded into the applet. They were then put into an array so that when the user selected a particular curve from the drop-down list in the applet, all of selected curve's parameters can be easily referenced.

3.3.3. Applet User Interface and Components

All of the applet components are first instantiated. The applet is then initialized in the **init()** method. The **init()** method initializes all of the components by calling **initComponents()**. The sizes and locations of all the applet components are set in this method and then added to the applet. The **initComponents()** method also calls the **displayPublicKeys()** method which reads the ID names (email addresses) of all of the public keys that have been created in the applet. These IDs are stored in a file on the server that hosts the applet. They are then displayed in the signature verification section of the applet under available public keys. The last thing that the **initComponents()** method does is to add an ActionListener to each button in the applet and to the public key list. It also adds an ItemListener to the drop down list of NIST curves to use.

3.3.4. ItemListeners and ActionListeners

The *ItemListen* and *ActionListen* classes implement the *ItemListener* and *ActionListener* classes, respectively. These two classes handle all events in the applet that deal with components. All button presses and list selections are handled differently and the listeners that were added to each component will call these classes to properly deal with an event when it happens.

When a user selects a curve to use in signature generation or verification, the *ItemListen* class will receive the selected index in the drop down list. This translates to the array index where the curve parameters of the selected curve are stored. A new elliptic curve is created using the *EllipticCurve* class with these curve parameters taken from the arrays. The *ActionListen* class handles all other component events and is much longer. It has only one method – **actionPerformed()** – that specifies different actions for all events.

3.3.4.1. The actionPerformed() Method

The following is a breakdown of how the **actionPerformed()** method handles each event.

Generate and Save Private Key button pressed	A random big integer is generated that will serve as the private key for the user. The Java *SecureRandom* class is used to create the key. Modulo of the order is performed on the private key. A popup window in the applet appears for the user so that they can select the location on their file system where the private key will be stored. The private key is saved under user's email address (ID). The users ID, curve selection, date, and private key will be stored in this file.
Generate and Save Public Key button pressed	The public key is created using the **multPoint()** method with the user's private key and base point as parameters. A php file that is stored on the host server is then called. This script will add to two files on the server – one that will save the list of public key IDS and another that will save the IDs along with the public key values associated with that user.
Upload Private Key browse button pressed	User uploads their private key by searching through local file system.
Upload Message browse button pressed	User uploads their message to be signed by searching through local file system.
Generate Signature button pressed	The applet calls the **sign()** method and generates a signature for the uploaded

	message. The user selects the location to save their signature file. The applet creates a file (named with the user's email address) and the ID, curve selection, date, and signature is stored in the file.
Upload Message for Verification button pressed	User uploads their message that was signed by searching through local file system.
Upload Signature for Verification button pressed	User uploads their signature to be verified by searching through local file system.
Verify Signature button pressed	The **verify()** method is called to verify the uploaded signature using the selected public key. A message is displayed stating whether the signature is valid or invalid.

Table 2: Summary of handling applet events

3.3.5. Verify, Sign, and Arithmetic Method Summaries

The cryptographic section of the *ECDSA* class is found in the **verify()**, **sign()**, and arithmetic methods. This is the summary of the major methods used to perform the ECDSA computations. The inputs, outputs, and provided functionality of each method are explained in the following table.

Method Summary	
BigInteger[]	**sign**() Follows the ECDSA algorithm for signature generation. The user's uploaded message file and private key are used to create the signature. A message digest is created using the **sha512()** method. For point multiplication, the **multiPoint()** method is used. All other arithmetic is performed using Java methods provided by the BigInteger class (modular arithmetic, subtraction, multiplication, modular inversion). The returned BigInteger array contains the signature [*r*,*s*]
String	**verify**() Follows the ECDSA algorithm for signature verification. This method calls the **getPubKey()** method which reads from a file located on the host server that contains all of the public keys. This method will find the selected public key and return the values. Point arithmetic performed using the **multipoint()** and **addPoints()** methods. All other arithmetic is done using the Java BigInteger class methods. The returned String indicates whether the signature is valid or invalid.
ECPoint	**addPoints**(ECPoint p1, ECPoint p2) Addition or doubling of points p1 and p2 is performed following the algorithms lined out in sections 2.4.5.1 and 2.4.5.2. All arithmetic uses the Java BigInteger class methods. The returned ECPoint is the sum of p1 and p2.

ECPoint	**multPoint(**BigInteger s, ECPoint p) Follows the binary method for scalar multiplication of appoint outlined in section 2.4.5.3. The addPoints() method is used for the necessary point addition. The returned ECPoint is the sum of p added s times.
BigInteger	**sha512(File file)** Uses the Secure Hash Algorithm 512 to generate a message digest of a given message contained in file. This method uses the Java MessageDigest class to create the hash. The returned BigInteger contains the message digest value.

Table 3: ECDSA method summaries

3.3.6. Additional Elliptic Curve Classes

Two elliptic curve classes were created within the main *ECDSA* class. They are the *ECPoint* class and the *EllipticCurve* class. These classes provide the ability to create an elliptic curve point and elliptic curve, respectively, as well as the ability to easily access these values. It should be noted that the functionality of both of these classes are available in Java SDK 2 version 1.5. Our system was constrained to use of Java v.1.4.2, but we modeled our classes after the Java 1.5 class to have the same names, method names, and close to the same arguments so that they can fairly easily be swapped.

3.3.6.1. ECPoint Class

Our *ECPoint* class has the same methods as the Java class but the constructor is slightly different. A third value was added to indicate if the point was at infinity. The **equals()** method also compares the point to another *ECPoint* instead of an object (as in the Java version). The following details the *ECPoint* methods.

Constructor Summary
ECPoint(BigInteger x, BigInteger y) Creates an ECPoint from the specified affine x-coordinate x and affine y-coordinate y.

Table 4: ECPoint class constructor summary

Method Summary	
boolean	**equals**(ECPoint pt) Compares this elliptic curve point for equality with the specified point.
BigInteger	**getAffineX**() Returns the affine x-coordinate x.
BigInteger	**getAffineY**() Returns the affine y-coordinate y.

Table 5: ECPoint class method summaries

3.3.6.2. EllipticCurve Class

Our *EllipticCurve* class differs from the Java one in that the constructor doesn't take a third field argument. Below is the *EllipticCurve* class methods summary.

Constructor Summary
EllipticCurve(BigInteger a, BigInteger b) Creates an elliptic curve with the coefficients a and b.

Table 6: EllipticCurve class constructor summary

Method Summary	
BigInteger	**getA**() Returns the first coefficient a of the elliptic curve.
BigInteger	**getB**() Returns the second coefficient b of the elliptic curve.

Table 7: EllipticCurve class method summaries

3.3.7 Class Function Flow

The main functions of the *ECDSA* class are the signature generation and verification methods. These two methods call on the same functions that were created to perform secure hashing of a message, point multiplication, and point addition. All other methods

that the **sign()** and **verify()** class utilize for computation are Java provided. The following flowchart represents the interactions between all of these functions.

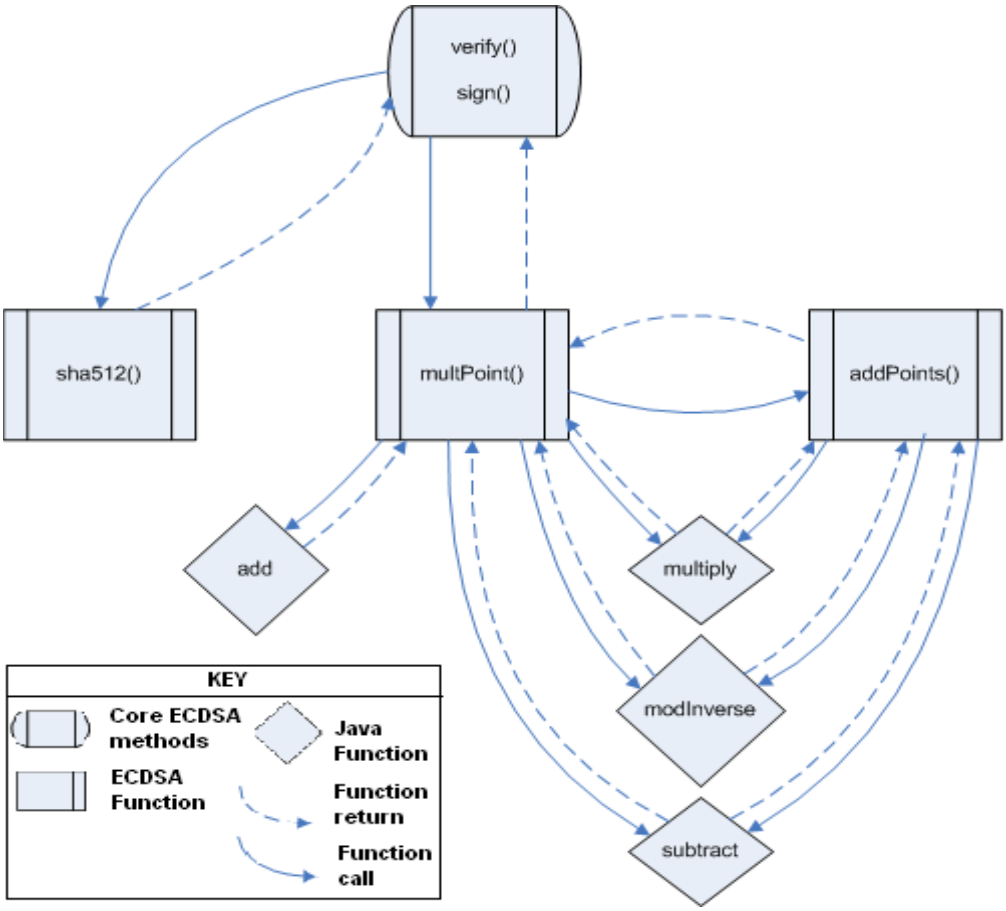

Figure 2: Flow of function usage

3.4. ANALYSIS AND IMPROVEMENTS

Each of the functions in the *ECDSA* class was analyzed for its timing performance compared with the key size being used for signature generation and verification. Although this applet was not optimized for efficiency, identification of the slow functions and their time complexities can aid further investigation into faster implementations of ECDSA in Java.

3.4.1. Timing of Functions

Five timing trials were conducted for the **sign()**, **verify()**, **multPoint()**, and **addPoints()** functions over each curve (of various key lengths) using a 32 KB message. The exact values obtained for the trials can be found in Appendix A. It was found that the **addPoints()** method was too fast to be timed in milliseconds and so was considered negligible in the overall efficiency of the class. The **sign()**, **verify()**, and **multipoint()** functions were identified as the core features that can be optimized. Following is the table of the average runtimes of each of the three functions and the graph representation of these runtimes as a function of the key size.

Key Size	sign()	verify()	multPoint()
521(bits)	308 (ms)	2653(ms)	637(ms)
384	225	1412	293
256	100	697	109
224	87	591	85
192	75	478	63

Table 8: Average function runtimes by key size

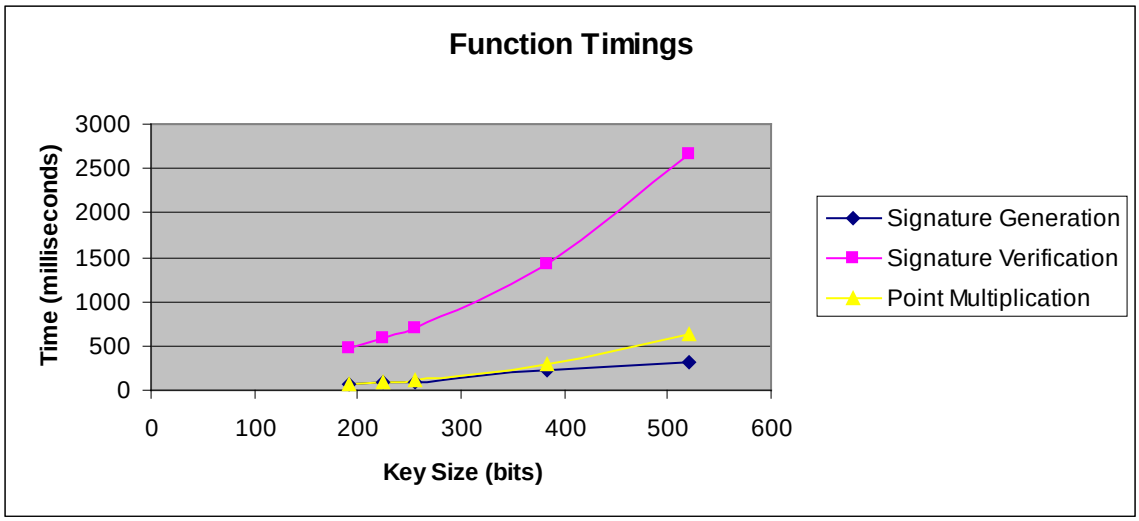

Figure 3: Depiction of ECDSA method runtimes as a function of the key size

3.4.2. Runtime Analysis

Based on the average runtime for each key size, the time complexity run time for the signature generation, verification, and point multiplication is the same – $O(n^2)$. The reason for such a high time order in the **sign()** and **verify()** methods is due to the **multPoint()** function that they both use.

3.4.3. Suggested Improvements

Most of the arithmetic operations in the ECDSA applet are performed using Java *BigInteger* functions and are as efficient as they can be when run on a Java platform. Therefore, the remaining arithmetic operations of either the **addPoints()** or the **multPoint()** functions must be the bottlenecks. The **addPoints()** function was found to have a very fast runtime compared to **multPoint()**, although **multPoint()** performs several point addition calculations. While a more precise timing system might fight that the **addPoints()** method does in fact have a relevant runtime, optimization of the **multPoint()** algorithm would have the largest impact. A more efficient point multiplication algorithm would also reduce the runtime of both the **sign()** and **verify()** methods by the same rate since they both depend on the **multPoint()** function and have the same time order.

4. CONCLUSION

Digital signatures provide cryptographic services that have become a necessity in data and network security. They offer non-repudiation, confidentiality, authentication, and integrity in a format that easily extends to the digital age. As systems become faster and smaller, they will need to maintain their security while using minimal resources. The elliptic curve digital signature algorithm provides the same functionality and security as the standard digital signature algorithm, but delivers it in a smaller key size. This makes it ideal for resource-constrained systems and network technology.

The Java language offers a platform to deliver the functionality of ECDSA in a web-based context. Leveraging off of this ability to provide a high level of security for data transfer in a public interface, we created a Java applet that implements ECDSA. This applet was embedded in a webpage so that it could be easily accessed by anyone on the Internet.

The ECDSA applet was created with the idea of being highly functional to a distributed community. User Alice can access the public site, generate a key, sign a document, and post her public key for user Bob to employ when verifying Alice's signature. Efficient implementation of the ECDSA on the Java platform was not considered as a part of this exercise. Java provided methods were used whenever possible to aid implementation and speed, but the algorithms in our functions were not optimized. Analysis of the function runtimes found that the time order of the three main functions that provided the ECDSA services were high. Two of these functions relied on

the third function, which performed point multiplication. This lends to the conclusion that optimization of this base function would improve the entire system. Further research can be done to determine a faster implementation of ECDSA in a Java environment.

5. BIBLIOGRAPHY

[1] Engelfriet, Arnoud. Elliptic Curve Cryptography. 1 Jan. 2004. 4 Apr. 2004. Ius Mentis. <http://www.iusmentis.com/technology/encryption/elliptic-curves/>

[2] Gutschmidt, Thomas. "An Overview of Cryptography in Java, Part 2: Provider History." JuniperWeb. 4 April 2005. <http://www.developer.com/java/ent/article.php/641531>

[3] Helton, Rich, and Johennie Helton. Java Security Solutions. Indianapolis: Wiley Publishing. 2002.

[4] Johnson, Don B., Alfred J. Menezes, Scott Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). Canada: Certicom. 2001. Available at <http://www.comms.scitech.susx.ac.uk/fft/crypto/ecdsa.pdf>

[5] Koc, Cetin Kaya. Elliptic Curve Cryptosystems. Posted Notes. Oregon State University. 2005. Available at <http://islab.oregonstate.edu/koc/ece575/notes/ec1.pdf>

[6] Mohapatra, Pradosh Kumar. "Public Key Cryptography." ACM Crossroads Sep. 2000. 10 Apr. 2005. < http://www.acm.org/crossroads/xrds7-1/crypto.html>

[7] NIST. NIST Brief Comments on Recent Cryptanalytic Attacks on SHA-1. 22 Feb. 2005. Computer Security Division. 27 Feb. 2005.

[8] Qiu, Qizhi and Qianxing Xiong. Research on Elliptic Curve Cryptography. The 8th International Conference on Computer Supported Cooperative Work in Design. Cincinnati. 2004.

[9] RSA Laboratories. Frequently Asked Questions About Today's Cryptography. 1998. 4 Apr. 2005. Bedford: RSA Laboratories. < http://www.rsasecurity.com/rsalabs/node.asp?id=2152>

[10] Sanchez-Vila, C, R. Sanchez-Reillo, M. de Miguel de Santos. Elliptic Curve Cryptography on Constraint Environments. The 38th Annual 2004 International Carnahan Conference. Spain: Security Technology. 2004.

[11] SkillSoft. Cryptography Protocols and Algorithms. Nashua: SkillSoft Press. 2003

[12] United States. National Institute of Standards and Technology. Digital Signature Standard (DSS). Federal Processing Standards Publication 186. Washington: FIPS. 1994.

APPENDICES

Appendix A: Timing results of function analysis

Note: All time is displayed in milliseconds

Appendix A. Table 1. Results of time analysis using various curves

Curve	Method	Test1	Test2	Test3	Test4	Test5	Average
P-521	Sign	375	375	360	359	375	308
	Verify	2625	2672	2672	2641	2656	2653
	multPoints	640	657	641	625	625	637
	addPoints	0	0	0	0	0	0
	SHA	15	15	32	31	31	25

Curve	Method	Test1	Test2	Test3	Test4	Test5	Average
P-384	Sign	203	203	313	203	203	225
	Verify	1390	1422	1422	1422	1406	1412.4
	multPoints	281	296	296	297	297	293.4
	addPoints	0	0	0	0	0	0
	SHA	31	31	31	31	31	31

Curve	Method	Test1	Test2	Test3	Test4	Test5	Average
P-256	Sign	110	110	94	94	94	100.4
	Verify	687	687	687	719	704	696.8
	multPoints	110	109	110	109	109	109.4
	addPoints	0	0	0	0	0	0
	SHA	31	15	16	31	15	21.6

Curve	Method	Test1	Test2	Test3	Test4	Test5	Average
P-224	Sign	78	78	94	94	93	87.4
	Verify	578	594	593	594	594	590.6
	multPoints	94	78	79	78	94	84.6
	addPoints	0	0	0	0	0	0
	SHA	15	15	31	32	31	24.8

Curve	Method	Test1	Test2	Test3	Test4	Test5	Average
P-192	Sign	78	79	63	78	78	75.2
	Verify	469	484	469	484	485	478.2
	multPoints	63	62	63	62	63	62.6
	addPoints	0	0	0	0	0	0
	SHA	15	32	16	31	31	25

Appendix B: NIST curve values over GF(p)

Values for the prime modulus *p* and order *r* are given in decimal form. Coefficient *b* and base point coordinate values *G x* and *G y* are given in hexadecimal form.

Curve P-192

p = 6277101735386680763835789423207666416083908700390324961279
r = 6277101735386680763835789423176059013767194773182842284081
b = 64210519 e59c80e7 0fa7e9ab 72243049 feb8deec c146b9b1
G x = 188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd 82ff1012
G y = 07192b95 ffc8da78 631011ed 6b24cdd5 73f977a1 1e794811

Curve P-224

p = 26959946667150639794667015087019630673557916260026308143510066298881
r = 26959946667150639794667015087019625940457807714424391721682722368061
b = b4050a85 0c04b3ab f5413256 5044b0b7 d7bfd8ba 270b3943 2355ffb4
G x = b70e0cbd 6bb4bf7f 321390b9 4a03c1d3 56c21122 343280d6 115c1d21
G y = bd376388 b5f723fb 4c22dfe6 cd4375a0 5a074764 44d58199 85007e34

Curve P-256

p = 115792089210356248762697446949407573530086143415290314195\
533631308867097853951
r = 115792089210356248762697446949407573529996955224135760342\
2259061068512044369
b = 5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e 27d2604b
G x = 6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0 f4a13945 d898c296
G y = 4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece cbb64068 37bf51f5

Curve P-384

p = 3940200619639447921227904010014361380507973927046544666794\
829340424572177149687032904726608825893800186160697311231 9
r = 3940200619639447921227904010014361380507973927046544666794\
690527962765939911326356939895630815229491355443365394264 3
b = b3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112 0314088f
5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef

G x = aa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62 8ba79b98 59f741e0 82542a38 5502f25d bf55296c 3a545e38 72760ab7
G y = 3617de4a 96262c6f 5d9e98bf 9292dc29 f8f41dbd 289a147c e9da3113 b5f0b8c0 0a60b1ce 1d7e819d 7a431d7c 90ea0e5f

Curve P-521

p = 68647976601306097149819007990813932172694353001433054093944634591855431833976560521225596406614545549772963113914808580371219879997166438125740282911150571 51

r = 686479766013060971498190079908139321726943530014330540939\
446345918554318339765539424505774633321719753296399637 1\
3633211138647686124403803403728088927070054 49

b = 051 953eb961 8e1c9a1f 929a21a0 b68540ee a2da725b 99b315f3 b8b48991 8ef109e1 56193951 ec7e937b 1652c0bd 3bb1bf07 3573df88 3d2c34f1 ef451fd4 6b503f00
G x = c6 858e06b7 0404e9cd 9e3ecb66 2395b442 9c648139 053fb521 f828af60 6b4d3dba a14b5e77 efe75928 fe1dc127 a2ffa8de 3348b3c1 856a429b f97e7e31 c2e5bd66
G y = 118 39296a78 9a3bc004 5c8a5fb4 2c7d1bd9 98f54449 579b4468 17afbd17 273e662c 97ee7299 5ef42640 c550b901 3fad0761 353c7086 a272c240 88be9476 9fd16650

Appendix C: ECDSA applet source code

```
/** ***************
 * ECDSA.java
 *
 * Author: Rhea Stadick
 * Date: May 30, 2005
 ******************/


import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.math.BigInteger;
import java.security.*;
import java.io.UnsupportedEncodingException;
import javax.swing.*;

import java.io.*;
import java.util.StringTokenizer;

import java.io.DataInputStream;

import java.io.IOException;
import java.net.*;


public class ECDSA extends Applet{
/*****************************************************************************************
 *
 * These are the hard-coded parameters of all of the NIST recommended curves for GF(P).
*****************************************************************************************/
   /**** P-192 ****/
          String p192X = "188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012";
   String p192Y = "07192b95ffc8da78631011ed6b24cdd573f977a11e794811";
   String p192B = "64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1";
   String p192P = "6277101735386680763835789423207666416083908700390324961279";
   String p192Order = "6277101735386680763835789423176059013767194773182842284081";

   /**** P-224 ****/
          String p224X = "b70e0cbd6bb4bf7f321390b94a03c1d356c21122343280d6115c1d21";
   String p224Y = "bd376388b5f723fb4c22dfe6cd4375a05a07476444d5819985007e34";
   String p224B = "b4050a850c04b3abf54132565044b0b7d7bfd8ba270b39432355ffb4";
   String p224P = "26959946667150639794667015087019630673557916260026308143510066298881";
   String p224Order =
"26959946667150639794667015087019625940457807714424391721682722368061";

   /**** P-256 ****/
          String p256X = "6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296";
   String p256Y = "4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5";
   String p256B = "5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b";
   String p256P =
"115792089210356248762697446949407573530086143415290314195533631308867097853951";
```

```
   String p256Order =
"115792089210356248762697446949407573529996955224135760342422259061068512044369";

   /**** P-384 ****/
         String p384X = "aa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e0" +
   "82542a385502f25dbf55296c3a545e3872760ab7";
   String p384Y = "3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113" +
   "b5f0b8c00a60b1ce1d7e819d7a431d7c90ea0e5f";
   String p384B = "b3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f" +
   "5013875ac656398d8a2ed19d2a85c8edd3ec2aef";
   String p384P = "394020061963944792122790401001436138050797392704654466679482934042457217714968703290472660882589380018616069731123190";
   String p384Order = "39402006196394479212279040100143613805079739270465446667948293405279627659399113263569398956308152294913554433653942643";

         /**** P-521 ****/
   String p521X = "c6858e06b70404e9cd9e3ecb662395b4429c648139053fb521"+
         "f828af606b4d3dbaa14b5e77efe75928fe1dc127a2ffa8de3348b3c1856a429bf97e7e31c2e5bd66";
         String p521Y ="11839296a789a3bc0045c8a5fb42c7d1bd998f54449579b4468"+
"17afbd17273e662c97ee72995ef42640c550b9013fad0761353c7086a272c24088be94769fd16650";
   String p521P =
"6864797660130609714981900799081393217269435300143305409394463459185543183" +
"3976560521225596406614545549772963113914808580371219879997166438125740282911150571510";
   String p521B = "051953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3"+
         "b8b489918ef109e156193951ec7e937b1652c0bd3bb1bf073573df883d2c34f1ef451fd46b503f00";
   String p521Order =
"6864797660130609714981900799081393217269435300143305409394463459185543183" +
"3976553942450577463332171975329639963713633211138647686124403803403728088927070054490";

   /* GLOBAL VARIABLES */
   public BigInteger[] xArray = {new BigInteger(p521X,16),new BigInteger(p384X,16),new
BigInteger(p256X,16), new BigInteger(p224X,16),new BigInteger(p192X,16)};
   public BigInteger[] yArray = {new BigInteger(p521Y,16),new BigInteger(p384Y,16),new
BigInteger(p256Y,16), new BigInteger(p224Y,16),new BigInteger(p192Y,16)};
   public BigInteger[] bArray = {new BigInteger(p521B,16),new BigInteger(p384B,16),new
BigInteger(p256B,16), new BigInteger(p224B,16),new BigInteger(p192B,16)};
   public BigInteger[] pArray = {new BigInteger(p521P),new BigInteger(p384P),new
BigInteger(p256P),new BigInteger(p224P),new BigInteger(p192P)};
   public BigInteger[] orderArray = {new BigInteger(p521Order),new BigInteger(p384Order),new
BigInteger(p256Order),new BigInteger(p224Order),new BigInteger(p192Order)};
   public ECPoint[] baseArray = {new ECPoint(xArray[0],yArray[0]),new
ECPoint(xArray[1],yArray[1]),new ECPoint(xArray[2],yArray[2]),
                              new ECPoint(xArray[3],yArray[3]),new ECPoint(xArray[4],yArray[4])};

   public EllipticCurve curve = new EllipticCurve(new BigInteger("-3").mod(pArray[0]),bArray[0]);
   public int arrayIndex;
   public BigInteger zero = BigInteger.ZERO;
   public File signMsgFile = null;
   public File verMsgFile = null;
   public File verPubKeyFile = null;
   public File verSigFile = null;
   public ECPoint pubKey;
   public BigInteger[] rs = new BigInteger[2]; // holds signature
   public BigInteger privateKey;
   public File privKeyFile = null;
```

```
/* SIGNATURE UI COMPONENTS */
Label curveLb = new Label("Select Curve");
Choice curveList = new Choice();
Label keysLb = new Label("Generate Keys");
Label emailLb = new Label("E-mail Address");
TextField emailTxt = new TextField();
Label signLb = new Label("Generate Signature");
	Button privKeyBtn = new Button("Generate & Save Private Key");
	Button pubKeyBtn = new Button("Generate & Save Public Key");
	Label pkFileLb = new Label("Upload Private Key File");
	TextField pkFileTxt = new TextField();
	Button pkBrowseBtn = new Button("Browse...");
	Label msgFileLb = new Label("Upload Message File");
	TextField msgFileTxt = new TextField();
	Button msgBrowseBtn = new Button("Browse...");
	Button signBtn = new Button("Generate & Save Signature");

	/* VERIFY UI COMPONENTS */
	Label verLb = new Label("Verify Signature");
	Choice verCurveList = new Choice();
	Label verCurveLb = new Label("Select Curve");
	Label msgLb = new Label("Message File");
	TextField verFileTxt = new TextField();
	Button verBrowseBtn = new Button("Browse");

	List pubKeyLst = new List();

	Label verPubKeyLb = new Label("Public Key File");
	TextField verPubKeyTxt = new TextField();
	Button verPubFileBtn = new Button("Browse");
	Label sigLb = new Label("Signature File");
	TextField verSigFileTxt = new TextField();
	Button verSigBtn = new Button("Browse");
	Button verBtn = new Button("Verify Signature");

	boolean isStandalone = false;

	/*
	 * Initialize Applet
	 */
	public void init() {
		setBackground(Color.WHITE);
		try {
			UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
		}
		catch(Exception e) { };
		try {
			initComponents();
		}
		catch (Exception e) {
			e.printStackTrace();
		}
	}

	/*
```

```
 * Initialize Components
 */
public void initComponents() throws Exception {
		setLayout(null);
		setLocation(new Point(0, 0));
		setSize(new Dimension(950, 500));

		curveLb.setLocation(new Point(300, 10));
		curveLb.setFont(new Font("" + Font.ROMAN_BASELINE, Font.BOLD, 12));
		curveLb.setSize(new Dimension(100,30));
		curveLb.setVisible(true);

		curveList.setLocation(new Point(400, 15));
		curveList.setSize(new Dimension(250, 30));
		curveList.addItem("Curve: NIST P-521");
		curveList.addItem("Curve: NIST P-384");
		curveList.addItem("Curve: NIST P-256");
		curveList.addItem("Curve: NIST P-224");
		curveList.addItem("Curve: NIST P-192");
		curveList.setVisible(true);

		/*** Set Size and Location of Signature Components ***/
		keysLb.setLocation(new Point(10, 60));
		keysLb.setFont(new Font("" + Font.ROMAN_BASELINE, Font.BOLD, 12));
		keysLb.setSize(new Dimension(300,30));
		keysLb.setVisible(true);

		emailLb.setLocation(new Point(10, 100));
		emailLb.setSize(new Dimension(100,30));
		emailLb.setVisible(true);

		emailTxt.setLocation(new Point(110, 100));
		emailTxt.setSize(new Dimension(340, 30));
		emailTxt.setVisible(true);

		privKeyBtn.setLocation(new Point(10, 150));
		privKeyBtn.setSize(new Dimension(200, 30));
		privKeyBtn.setVisible(true);

		pubKeyBtn.setLocation(new Point(250, 150));
		pubKeyBtn.setSize(new Dimension(200, 30));
		pubKeyBtn.setVisible(true);

		signLb.setLocation(new Point(10, 215));
		signLb.setFont(new Font("" + Font.ROMAN_BASELINE, Font.BOLD, 12));
		signLb.setSize(new Dimension(300,30));
		signLb.setVisible(true);

		pkFileLb.setLocation(new Point(10, 250));
		pkFileLb.setSize(new Dimension(300,30));
		pkFileLb.setVisible(true);

		pkFileTxt.setLocation(new Point(10, 290));
		pkFileTxt.setSize(new Dimension(300, 30));
		pkFileTxt.setVisible(true);
```

```
pkBrowseBtn.setLocation(new Point(350, 290));
pkBrowseBtn.setSize(new Dimension(100, 30));
pkBrowseBtn.setVisible(true);

msgFileLb.setLocation(new Point(10, 340));
msgFileLb.setSize(new Dimension(300,30));
msgFileLb.setVisible(true);

msgFileTxt.setLocation(new Point(10, 380));
msgFileTxt.setSize(new Dimension(300, 30));
msgFileTxt.setVisible(true);

msgBrowseBtn.setLocation(new Point(350, 380));
msgBrowseBtn.setSize(new Dimension(100, 30));
msgBrowseBtn.setVisible(true);

signBtn.setLocation(new Point(10, 430));
signBtn.setSize(new Dimension(440, 50));
signBtn.setVisible(true);

/*** Set Size and Location of Verify Components ***/
verLb.setLocation(new Point(500, 60));
verLb.setFont(new Font("" + Font.ROMAN_BASELINE, Font.BOLD, 12));
verLb.setSize(new Dimension(300,30));
verLb.setVisible(true);

msgLb.setLocation(new Point(500, 100));
msgLb.setSize(new Dimension(100,30));
msgLb.setVisible(true);

verFileTxt.setLocation(new Point(500, 140));
verFileTxt.setSize(new Dimension(300, 30));
verFileTxt.setVisible(true);

verBrowseBtn.setLocation(new Point(840, 140));
verBrowseBtn.setSize(new Dimension(100, 30));
verBrowseBtn.setVisible(true);


verPubKeyLb.setLocation(new Point(500, 190));
verPubKeyLb.setSize(new Dimension(300,30));
verPubKeyLb.setVisible(true);

pubKeyLst.setLocation(new Point(500, 230));
pubKeyLst.setSize(new Dimension(440, 90));
pubKeyLst.setVisible(true);

displayPublicKeys(); // display all saved Public Keys

sigLb.setLocation(new Point(500, 340));
sigLb.setSize(new Dimension(300,30));
sigLb.setVisible(true);

verSigFileTxt.setLocation(new Point(500, 380));
verSigFileTxt.setSize(new Dimension(300, 30));
verSigFileTxt.setVisible(true);
```

```
        verSigBtn.setLocation(new Point(840, 380));
        verSigBtn.setSize(new Dimension(100, 30));
        verSigBtn.setVisible(true);

        verBtn.setLocation(new Point(500, 430));
        verBtn.setSize(new Dimension(440, 50));
        verBtn.setVisible(true);


        add(curveLb);
        add(curveList);

        /*** Add Signature Components ***/
        add(keysLb);
        add(emailLb);
        add(emailTxt);
        add(privKeyBtn);
        add(pubKeyBtn);
        add(signLb);
        add(pkFileLb);
        add(pkFileTxt);
        add(pkBrowseBtn);
        add(msgFileLb);
        add(msgFileTxt);
        add(msgBrowseBtn);
        add(signBtn);

        /*** Add Verification Components ***/
        add(verLb);
        add(msgLb);
        add(verFileTxt);
        add(verBrowseBtn);
        add(pubKeyLst);
        add(verPubKeyLb);
        add(sigLb);
        add(verSigFileTxt);
        add(verSigBtn);
        add(verBtn);

        /*** Add an action listner to all Applet components ***/
        ActionListener listener = new ActionListen();
        ItemListener ilistener = new ItemListen();
        curveList.addItemListener(ilistener);
        privKeyBtn.addActionListener(listener);
        pubKeyBtn.addActionListener(listener);
        pkBrowseBtn.addActionListener(listener);
        msgBrowseBtn.addActionListener(listener);
        signBtn.addActionListener(listener);
        verBrowseBtn.addActionListener(listener);
        pubKeyLst.addActionListener(listener);
        verPubFileBtn.addActionListener(listener);
        verSigBtn.addActionListener(listener);
        verBtn.addActionListener(listener);

}
```

```
class ItemListen implements ItemListener{
	public void itemStateChanged(ItemEvent e){
		arrayIndex = curveList.getSelectedIndex();
curve = new EllipticCurve(new BigInteger("-3").mod(pArray[arrayIndex]),bArray[arrayIndex]);
	}
}

class ActionListen implements ActionListener{
	JFrame popupFrame = new JFrame();
	JOptionPane popup = new JOptionPane();
	public void actionPerformed(ActionEvent event){
		try{
			if(event.getSource() == privKeyBtn){
				SecureRandom sr = new SecureRandom();
				privateKey = new BigInteger(512,sr);
				privateKey =
			privateKey.mod((orderArray[arrayIndex].subtract(new BigInteger("1"))));
				File pkFile = null;

				JFileChooser chooser = new JFileChooser();
				if(chooser.showOpenDialog(null) ==
					JFileChooser.APPROVE_OPTION)
				{
					String fPath = ""+chooser.getCurrentDirectory();
					fPath = fPath+ "\\" + emailTxt.getText()+".privKey";
					pkFile = new File(fPath);
					//pkFile = chooser.getSelectedFile();
				}

				PrintWriter pw = new PrintWriter(new
					FileOutputStream(pkFile));
				pw.println("ID:");
				pw.println(emailTxt.getText());
				pw.println("" + curveList.getItem(arrayIndex));
				pw.println("Private Key:");
				pw.println("" + privateKey.toString(16));
				pw.close();
			}
			if(event.getSource() == pubKeyBtn){
				pubKey = multPoint(privateKey, baseArray[arrayIndex]);
			  try{
			    String x = pubKey.getAffineX().toString();
			    String id = emailTxt.getText();
			    String y = pubKey.getAffineY().toString();
			    String data = id + pubKey.getAffineX() + pubKey.getAffineY();
			    URL url = new URL("http","web.engr.orst.edu",
			    "/~stadick/pubKeyAccess.php?id="+id+"&x="+x+"&y="+y);
			    URLConnection con = url.openConnection();

			    con.setDoOutput(true);
			    con.setDoInput(true);
			    con.setUseCaches(false);
			    con.setRequestProperty("Content-type", "text/plain");
			    con.setRequestProperty("Content-length", data.length()+"");
```

```
                PrintStream out = new PrintStream(con.getOutputStream());
                out.print(data);
                System.out.println(con.getContent());
                System.out.println(con.getURL());
                out.flush();
                out.close();

                pubKeyLst.add(id);
               }
            catch(MalformedURLException e){
                    System.out.println("Malformed URL Exception");
            }
            catch(IOException e){
                    System.out.println("IO Exception: " + e.toString());
            }
                    JOptionPane.showMessageDialog(popupFrame,
"Your Public Key has been stored and can be refrenced by your email address and curve selection.");

            }
            if(event.getSource() == pkBrowseBtn)
            {
                    JFileChooser chooser = new JFileChooser();
                    if(chooser.showOpenDialog(null) ==
                            JFileChooser.APPROVE_OPTION)
                    {
                            privKeyFile = chooser.getSelectedFile();
                    }
                    pkFileTxt.setText(privKeyFile.getAbsolutePath());

                    BufferedReader buffer =
                            new BufferedReader(new FileReader(privKeyFile));
                    String temp = buffer.readLine();
                    temp = buffer.readLine();
                    temp = buffer.readLine();
                    temp = buffer.readLine();
                    privateKey = new BigInteger(buffer.readLine(), 16);
                    buffer.close();
            }

            if(event.getSource() == msgBrowseBtn)
            {
                    JFileChooser chooser = new JFileChooser();
                    if(chooser.showOpenDialog(null) ==
                            JFileChooser.APPROVE_OPTION)
                    {
                            signMsgFile = chooser.getSelectedFile();
                    }
                    msgFileTxt.setText(signMsgFile.getAbsolutePath());
            }
            if(event.getSource() == signBtn)
            {
                    String signPath =
signMsgFile.getAbsolutePath().replaceAll(signMsgFile.getName(), "");
                    String msgFileName = signMsgFile.getName();
                    StringTokenizer st = new StringTokenizer(msgFileName, ".");
                    signPath = signPath + st.nextToken()+".sign";
```

```
                    File signFile = new File(signPath);
                    rs = sign();

                    BufferedReader buffer =
                    new BufferedReader(new FileReader(privKeyFile));
                    String temp = buffer.readLine();
                    String id = buffer.readLine();
                    buffer.close();

                    PrintWriter pw = new PrintWriter(new
                                FileOutputStream(signFile));
                    pw.println("ID:");
                    pw.println(id);
                    pw.println(""+curveList.getItem(arrayIndex));
                    pw.println("Signature for file " + msgFileName + "  (r,s):");
                    pw.println(rs[0].toString(16));
                    pw.println(rs[1].toString(16));
                    pw.close();
                    JOptionPane.showMessageDialog(popupFrame,
                                "Signature saved to: " + signFile);

            }
            if((event.getSource() == verBrowseBtn)||
                                        (event.getSource() == verSigBtn)
                                ){
                    JFileChooser chooser = new JFileChooser();
                    if(chooser.showOpenDialog(null) ==
                                JFileChooser.APPROVE_OPTION)
                    {
                            if(event.getSource() == verBrowseBtn){
                                verMsgFile = chooser.getSelectedFile();
                            verFileTxt.setText(verMsgFile.getAbsolutePath());
                            }
                            else if (event.getSource() == verSigBtn){
                                verSigFile = chooser.getSelectedFile();
                            verSigFileTxt.setText(verSigFile.getAbsolutePath());
                            }
                    }
            }
            if(event.getSource()== verBtn){
                    verPubKeyFile = new File("publicKeys\\" +
                            pubKeyLst.getSelectedItem() + ".pubKey");
                    System.out.println("verPubKeyFile selected: " +
                            verPubKeyFile.getAbsolutePath());
                    String ver = verify();
                    JOptionPane.showMessageDialog(popupFrame, ver);
            }
    }
    catch(IOException e2){
            System.out.println(e2);
            JOptionPane.showMessageDialog(popupFrame, e2);
    }
    catch(NoSuchAlgorithmException e2){
            System.out.println(e2);
            JOptionPane.showMessageDialog(popupFrame, e2);
```

```
                    }
                }
        }

        public void displayPublicKeys() {
                try{
                 URL url = new URL("http","web.engr.orst.edu", "/~stadick/pubKeyIDs.txt");
                 URLConnection con = url.openConnection();

                 con.setDoInput(true);
                 con.setUseCaches(false);
                 con.setRequestProperty("Content-type", "text/plain");
                 DataInputStream in = new DataInputStream(con.getInputStream());
                 String id;
                 while ((id = in.readLine()) != null) {
                                pubKeyLst.add(id);
                 }
                 in.close();
                }
                catch(MalformedURLException e){
                System.out.println("Malformed URL Exception");
                }
                catch(IOException e){
                System.out.println("IO Exception: " + e.toString());
                }
        }

        public ECPoint getPubKey() throws MalformedURLException,IOException {
                BigInteger x = BigInteger.ZERO;
                BigInteger y = BigInteger.ZERO;
                URL url = new URL("http","web.engr.orst.edu", "/~stadick/pubKeyList.txt");
           URLConnection con = url.openConnection();

           con.setDoInput(true);
           con.setUseCaches(false);
           con.setRequestProperty("Content-type", "text/plain");
           DataInputStream in = new DataInputStream(con.getInputStream());
                String id;
                int count = 0;
                while ((id = in.readLine()) != null) {
                        if((count%3 == 0) && id.equals(pubKeyLst.getSelectedItem())){
                                x = new BigInteger(in.readLine());
                                y = new BigInteger(in.readLine());
                                break;
                        }
                        }
                in.close();

                ECPoint q = new ECPoint(x,y);
                return q;
        }

        BigInteger[] sign() throws
UnsupportedEncodingException,NoSuchAlgorithmException,IOException{
                BigInteger kinv, r, s, k;
                BufferedReader buffer = new BufferedReader(new FileReader(privKeyFile));
```

```
                String temp = buffer.readLine();
                temp = buffer.readLine();
                temp = buffer.readLine();
                temp = buffer.readLine();
                privateKey = (new BigInteger(buffer.readLine(),16));
                buffer.close();

                //pubKey = multPoint(privateKey, baseArray[arrayIndex]);
                SecureRandom sr = new SecureRandom();
                k = new BigInteger(512, sr);
                k = k.mod(orderArray[arrayIndex].subtract(new BigInteger("1")));
                while((k.compareTo(java.math.BigInteger.ZERO) == 0) ||
!(k.gcd(orderArray[arrayIndex]).compareTo(java.math.BigInteger.ONE) == 0)){
                        k = new BigInteger(512,sr);
                        k = k.mod(orderArray[arrayIndex].subtract(new BigInteger("1")));
                }
                ECPoint p = multPoint(k, baseArray[arrayIndex]);
                if(p.getAffineX().equals(zero)){
                        return sign();
                }

                BigInteger h = sha512(signMsgFile);

                r = p.getAffineX();
                kinv = k.modInverse(orderArray[arrayIndex]);
                s = (kinv.multiply((h.add((privateKey.multiply(r)))))).mod(orderArray[arrayIndex]);
                if(s.compareTo(java.math.BigInteger.ZERO) == 0)
                        return sign();

                rs[0] = r;
                rs[1] = s;

                return rs;
        }


        String verify()  throws
UnsupportedEncodingException,NoSuchAlgorithmException,IOException{
                BigInteger w, u1, u2,v;
                BigInteger rs[] = new BigInteger[2];
                ECPoint q = new ECPoint(zero, zero);
                ECPoint pt;

                BufferedReader buffer = new BufferedReader(new FileReader(verSigFile));

                String temp = buffer.readLine();
                temp = buffer.readLine();
                temp = buffer.readLine();
                temp = buffer.readLine();
                rs[0] = new BigInteger(buffer.readLine(), 16);
                rs[1] = new BigInteger(buffer.readLine(),16);
    buffer.close();

    q = getPubKey();
```

```
                if(rs[0].compareTo(orderArray[arrayIndex])==1 || rs[0].compareTo(BigInteger.ONE)==-
1 || rs[1].compareTo(orderArray[arrayIndex])==1 || rs[1].compareTo(BigInteger.ONE)==-1){
                        return "***SIGNATURE WAS NOT IN VALID RANGE***";
                }

                w = rs[1].modInverse(orderArray[arrayIndex]);
                BigInteger h = sha512(verMsgFile);

                u1 = h.multiply(w);
                u2 = rs[0].multiply(w);

                ECPoint u1P = multPoint(u1, baseArray[arrayIndex]);
                ECPoint u2Q = multPoint(u2, q);
                pt = addPoints(multPoint(u1,baseArray[arrayIndex]), multPoint(u2, q));

                v = pt.getAffineX();
                v = v.mod(orderArray[arrayIndex]);

                if(v.compareTo(rs[0])==0){
                        return "Valid Signature";
                }
                else{
                        return "Invalid Signature";
                }
        }


        /*** Point Addition ***/
        ECPoint addPoints(ECPoint p1, ECPoint p2){
                BigInteger bigX;
                ECPoint p3 = new ECPoint(zero, zero);
          BigInteger slope,x,y, temp;
          if( p1.getAffineY().equals(zero) || p2.getAffineY().equals(zero) || (p1.infinity && p2.infinity) ||
(p1.getAffineX().equals(p2.getAffineX()) && p1.getAffineY().equals(p2.getAffineY().negate()))){
                p3.infinity = true;
          }
          else{  // IF POINTS ARE EQUAL
                        if((p1.getAffineX().equals(p2.getAffineX())) &&
                                        (p1.getAffineY().equals(p2.getAffineY()))){
                        temp = new BigInteger(""+3);
                        slope =
((temp.multiply(p1.getAffineX().pow(2))).add(curve.getA())).multiply((p1.getAffineY().add(p1.getAffinY(
))).modInverse(pArray[arrayIndex]));
                        x =
(slope.multiply(slope)).subtract(p1.getAffineX().add(p1.getAffineX()));  //(slope*slope) - (2*p1.x);
                        x = x.mod(pArray[arrayIndex]);
                        y =
(slope.multiply(p1.getAffineX().subtract(x))).subtract(p1.getAffineY());
                }else{    // POINTS ARE NOT EQUAL
                        if(p1.infinity){
                                x = p2.getAffineX();
                                y = p2.getAffineY();
                        }
                        else if(p2.infinity){
                                x = p1.getAffineX();
                                y = p1.getAffineY();
```

```
                                        }
                                        else{
                                                temp =
                                (p2.getAffineX().subtract(p1.getAffineX())).modInverse(pArray[arrayIndex]);
                                                slope =
(p2.getAffineY().subtract(p1.getAffineY())).multiply(temp);             //(p2.y - p1.y)/(p2.x - p1.x);
                                                x =
((slope.multiply(slope)).subtract(p1.getAffineX())).subtract(p2.getAffineX());
//(slope*slope) - p1.x - p2.x;
                                                y =
(slope.multiply(p1.getAffineX().subtract(x))).subtract(p1.getAffineY());
        //slope*(p1.x - x) - p1.y;
                                        }
                                        x = x.mod(pArray[arrayIndex]);
                                }
                                y = y.mod(pArray[arrayIndex]);

                                p3 = new ECPoint(x,y);
                        }
                        return p3;
        }

        /*** Scalar Multiplication of Points ***/
        ECPoint multPoint(BigInteger s, ECPoint p){
                String binS = s.toString(2);
                ECPoint q = new ECPoint(zero,zero);
                q.infinity = true;

                if(binS.substring(0,1).equals("1")){
                        q = p;
                }
                for(int i = 1; i< binS.length(); i++){
                        q = addPoints(q,q);
                        if(binS.substring(i, i+1).equals("1")){
                                q = addPoints(q,p);
                        }
                }
                return(q);

        }

        public BigInteger sha512(File file) throws
UnsupportedEncodingException,NoSuchAlgorithmException,IOException{
    FileInputStream fileStrm = new FileInputStream(file);
          byte[] byteMsg =new byte[(int)file.length()];

          fileStrm.read(byteMsg);
    MessageDigest sha = MessageDigest.getInstance("SHA-512");
    sha.update(byteMsg);
    byte[] hash = sha.digest();
    BigInteger h = new BigInteger(hash);
    return h;
        }

        private class ECPoint{
                BigInteger x;
```

```
		BigInteger y;
		boolean infinity;

		public ECPoint(BigInteger xPos, BigInteger yPos){
			x = xPos;
			y = yPos;
			infinity = false;
		}

		public boolean equals(ECPoint pt){
			if(this.x.equals(pt.getAffineX()) && this.y.equals(pt.getAffineY())){
				return true;
			}else
				return false;
		}

		public BigInteger getAffineX(){
			return this.x;
		}

		public BigInteger getAffineY(){
			return this.y;
		}
	}

	private class EllipticCurve{
			BigInteger a;
			BigInteger b;
		public EllipticCurve(BigInteger aVal, BigInteger bVal){
			a = aVal;
			b = bVal;
		}

		public BigInteger getA(){
			return this.a;
		}

		public BigInteger getB(){
			return this.b;
		}
	}
}
```