AN ABSTRACT OF THE THESIS OF

Haluk Kent Tanik for the degree of Master of Science in Computer Science
presented on June 19, 2001.

TITLE:

ECDSA Optimizations on an ARM Processor for a NIST Curve Over GF(p)

Abstract approved: Redacted for Privacy

Çetin K. KOÇ

The Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve analog of the Digital Signature Algorithm (DSA) and a federal government approved digital signature method. In this thesis work, software optimization techniques were applied to speed up the ECDSA for a particular NIST curve over GF(p). The Montgomery multiplication is used extensively in the ECDSA. By taking advantage of the algorithmic properties of the Montgomery multiplication method, special structure of the curve parameters and also applying certain fundamental and specific software optimization techniques, we have achieved an overall 26% speed improvement. Further enhancements were made by implementing the Montgomery multiplication in the ARM assembly language that resulted in 44% speed improvement. The optimizations discussed in this thesis could easily be adapted to other curves with or without changes.

© Copyright by Haluk Kent Tanik
June 19, 2001
All Rights Reserved

ECDSA Optimizations on an ARM Processor for a NIST Curve Over GF(p)

by

Haluk Kent Tanik

A THESIS

submitted to

Oregon State University

In partial fulfillment of the

requirements for the degree of

Master of Science

Completed June 19, 2001

Commencement June 2002

ACKNOWLEDGMENTS

First, I would like to thank my family for their endless support and for encouraging me along the way during my studies.

I give special thanks to my major professor Dr. Koç for recognizing my potential and Dr. Minoura, Dr. Budd and Dr. Stetz for dedicating their time to participate in my graduate committee.

I would also like to thank the following people who have provided valuable help during my research: Serdar S. Erdem, Erkay Savas, Eda Turan and Tugrul Yanik.

Finally, I acknowledge financial support for this work from Secured Information Technology, Inc.

Haluk Kent Tanik
Corvallis, Oregon

Master of Science thesis of Haluk Kent Tanik presented on June 19, 2001

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Head of Computer Science Department

Redacted for Privacy

Dean of Graduate School

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

Redacted for Privacy

Haluk Kent Tanik, Author

TABLE OF CONTENTS

Page

1. INTRODUCTION 1

2. LITERATURE REVIEW 3

2.1. Information Security 3

2.2. Services provided by Cryptographic Systems 3

2.3. Types of Cryptosystems 4

2.4. Public-Key Cryptosystems (Security Services using Public-Key Cryptosystems) 5

2.4.1. Public-Key Encryption and Decryption -*Confidentiality* 6
2.4.2. Digital Signatures - Data Origin Authentication, Data Integrity, Non-repudiation 6
2.4.3. Cryptographic Protocol: Signed Challenges -*User Authentication* 8

2.5. Mathematical Problems for Cryptosystems 8

2.5.1. Discrete Logarithm Problem (DLP) 9
2.5.2. The Elliptic Curve Discrete Logarithm Problem (ECDLP) 9

2.6. Security and Efficiency Comparison of Public-Key Cryptosystems 10

2.6.1. Security 10
2.6.2. Efficiency 10
2.6.3. Computational Overheads 10
2.6.4. Key Size 11
2.6.5. Bandwidth 11

2.7. What are Elliptic Curves 12

2.7.1. Elliptic Curves Cryptography and Elliptic Curve Groups 14
2.7.2. Arithmetic in an Elliptic Curve Group over GF(p) 17
2.7.2.1. Adding distinct points on Elliptic Curve over GF(p) 17
2.7.2.2. Doubling the point on Elliptic Curve over GF(p) 17

2.8. ECDSA Protocol 17

2.8.1. ECDSA Key Generation 18
2.8.2. ECDSA Signature Generation 18
2.8.3. ECDSA Signature Verification 18

3. OPTIMIZATION OF ECDSA ALGORITHM 20

3.1 Montgomery Multiplication 20

3.2 Montgomery Multiplication Optimization 23

TABLE OF CONTENTS (CONTINUED)

Page

3.3 Other Optimizations 34

3.4. Assembly Language Implementation 37

4. RESULTS and DISCUSSION 40

5. CONCLUSION 46

BIBLIOGRAPHY 47

APPENDIX: BACKGROUND IN RECOMMENDED ELLIPTIC CURVES FOR FEDERAL GOVERNMENT USE 49

LIST OF FIGURES

Figure		Page
1.	The graph of $y^2=x^3+ax+b$, where a=1, b=1	13
2.	Geometric description of the addition of two distinct elliptic curve points: P + Q = R	14
3.	Geometric description of the doubling of an elliptic curve point: P + P = 2p = R	14
4.	Elliptic Curve points: $y^2 = x^3 + x$ over F_{23}	16

LIST OF TABLES

Table		Page
1.	Size of system parameters and key pairs (approximate value)	11
2.	Signature sizes on 2000-bit messages	11
3.	Size of encrypted 100-bit message	12
4.	Classical Montgomery Multiplication Algorithm	24
5.	Montgomery Multiplication Algorithm after removal of temporary array *tmp*	26
6.	The Montgomery multiplication algorithm after unrolling for first iteration	26
7.	Multiplication in first iteration (a), other loop iterations (b)	28
8.	Reduction with array condition check	29
9.	Reduction with single condition check	30
10.	Optimized carry correction section	32
11.	The optimized Montgomery Multiplication Algorithm	34
12.	Loop jamming and Loop invariant removal	35
13.	Loop initialization decision	36
14.	Double word operations in pseudocode (a), and in ARM assembly language (b)	39
15.	The Montgomery Multiplication operation breakdown before optimization	41
16.	The Montgomery Multiplication operation breakdown after optimization	43
17.	ECDSA timings and percent improvement. (a) classical method and C version comparison, (b) comparison with classical method and assembly version.	45

To my family...

for their endless patience, love and support

ECDSA Optimizations on an ARM Processor for a NIST Curve Over GF(p)

1. INTRODUCTION

Public-key cryptography was first introduced in 1976 by Whitfield Diffie and Martin Hellman. Since then public-key cryptographic systems have proven to be effective and more controllable than symmetric key systems. There are three types of public-key systems: integer factorization systems, discrete logarithm systems, and Elliptic Curve Cryptographic (ECC) systems. Among these three, however, the ECC offers significant efficiency savings that is advantageous in many applications, particularly when computational power, bandwidth, or storage space is limited. Neal Koblitz and Victor Miller independently in 1985 proposed using the group of points on an elliptic curve defined over a finite field to implement various discrete log cryptosystems. One such cryptographic protocol is the elliptic curve analog of the DSA, called ECDSA.

There are three US. Government Federal Information Processing Standard (FIPS)-approved algorithms for generating and verifying digital signatures. These are the Digital Signature Algorithm (DSA), RSA, and Elliptic Curve DSA (ECDSA; as specified in ANSI X9.62). DSA is based on the computational intractability of the discrete logarithm problem. ECDSA is the elliptic curve analogue of the Digital Signature Algorithm. ECDSA was accepted in 1998 as an ISO standard, in 1999 as an ANSI standard, and in 2000 as IEEE and NIST standards. Unlike the discrete logarithm problem and the integer factorization problem, no sub exponential-time algorithm is known for the elliptic curve discrete logarithm problem. Therefore, an algorithm that uses elliptic curves carries greater the strength-per-key-bit. Public key cryptography needs high-speed and space-efficient algorithms for modular multiplication. The Montgomery multiplication algorithm is considered one of the best and useful algorithms in modular arithmetic.

This thesis paper gives a brief introduction of digital signature scheme and discusses ECDSA from both mathematical point of view and also as schema and

gives a brief discussion about the security, implementation, and interoperability issues of ECDSA (Chapter 2). Since the Montgomery multiplication algorithm is extensively used in ECDSA and it uses modulus of *Curve P-224* extensively, the special structure of the curve will be well exploited in this algorithm. After optimizing this algorithm, other software optimization techniques will be applied to whole project to achieve better performance (Chapter 3). In Chapter 4, the result of optimizations will be presented and discussed. Chapter 5 will conclude this thesis work. Appendix will give brief background in recommended elliptic curves for federal government use and the curve parameters of the *Curve P-224* will be presented.

2. LITERATURE REVIEW

2.1. INFORMATION SECURITY

Information is a valuable asset that governments and commercial businesses depend on. However, information in "electronic form" faces more serious security threats than one printed on paper. Information in electronic form can potentially be stolen from a remote location or can be intercepted and manipulated. Thus, **information security** describes all measures taken to prevent unauthorized use of electronic data. The unauthorized use of data can be disclosure, alteration, substitution, or destruction of the data of interest. Information Security provides following three services:

- **Confidentiality** – prevention of unauthorized disclosure of information
- **Integrity** – prevention of unauthorized modification of information
- **Availability** – prevention of unauthorized withholding of information or resources

Among the various measures proposed, the use of cryptographic systems offers the highest level of security together with maximum level of flexibility. In fact, a cryptographic system that is managed and implemented correctly offers the highest level of security for electronic information available today.

2.2. SERVICES PROVIDED BY CRYPTOGRAPHIC SYSTEMS

Cryptographic systems (or **cryptosystems**) aim to provide all three services of information security. In order to understand how cryptosystems are implemented, confidentiality and integrity are also classified into five services that constitute "building blocks" for system security. These are:

- **Data confidentiality** – protection of data from unauthorized disclosure, which can be satisfied by use of encryption algorithms to hide the content of

messages.

- **Data Integrity** – protection of data from unauthorized alterations or destruction, which can be provided by integrity check functions to detect that whether a document has been modified or not.

- **Data origin authentication** – corroboration of the source of data, which can be satisfied by digital signatures such as ECDSA.

- **User authentication** – assurance that the parties involved in real time transaction are who they are supposed to be.

- **Non-repudiation** – the existence of evidence that data has been sent or received, therefore the sender or receiver cannot later falsely deny this fact.

Cryptosystems take into account the fact that not only it is important to secure a transaction but it is also important to do so efficiently.

2.3. TYPES OF CRYPTOSYSTEMS

Modern Cryptography does not depend on the secrecy of its algorithms, however cryptographic algorithms use keys to protect information. Thus, key management is an important issue and needs to be protected by access control mechanisms on computer systems. Encryption is the term used for the process that transforms intelligible data, called the plaintext (or cleartext), to an illegible version, called the ciphertext. This conversion is controlled by an electronic key, say k. Decryption, on the other hand, is used for transforming the ciphertext back into plaintext, and it is controlled by another related key, say l.

There are two classes of cryptosystems based on the relationship between k and l:

1. **Symmetric-key cryptosystems:** DES, FEAL, IDEA, RC5, etc

2. **Public-key cryptosystems:** ECC, ElGamal, Rabin, RSA, etc

In a symmetric-key cryptosystem, the same key is used for both encryption and decryption purposes. It must be kept secret. This becomes inefficient when there is a large network with users distributed over a wide area. It is inefficient because each individual in the network should have a 'distinct' key to communicate with each other. Thus, maintaining a large number of shared secret keys can become a difficult management task.

Public-key cryptosystems perform separated encryption and decryption. There are two keys; the public key, used in encryption, and the mathematically related private key used in decryption. Knowledge of the public key should not make it feasible to derive the private key. Simply, a person selects and publishes his/her public key, then everyone uses that public key to encrypt messages for that person. Since this person keeps his/her private key secret only he/she can decrypt the ciphertext. Furthermore, in a network of multi-users the number of keys that needs to be kept secret is much more smaller than the secret-key cryptosystems.

Because the symmetric-key systems can process data much faster than current public-key schemes, to find the middle way a "hybrid" system can also be used. This system makes use of the extra speed that a symmetric-key system provides, while employing a public-key system to avoid the key distribution problem.

2.4. PUBLIC-KEY CRYPTOSYSTEMS (SECURITY SERVICES USING PUBLIC-KEY CRYPTOSYSTEMS)

Public-key cryptosystems aim to provide all services of information security. It will be shown shortly that all the services of information security can be provided by the correct implementation of a public-key cryptosystem. To illustrate: Let Alice and Bob be two users wishing to communicate through hypothetical communication line, and let D_{bob} and E_{bob} be Bob's private key and public key, respectively. Finally, let Adv be the adversary who wants to challenge secure communication.

2.4.1. Public-key encryption and decryption - *confidentiality*

Suppose Alice wants to send a secret message to Bob. Alice first searches for Bob's public key, E_{bob}, from a public directory where all users' public key are available to each other. Thus, to send the message M to Bob, Alice first looks up E_{bob} in the public directory, then encrypts M by performing the public-key transformation using E_{bob} to transform M into ciphertext C (1), and sends C to Bob.

$$C = E_{bob}(M) \qquad (1)$$

Bob retrieves M by transforming C using his secret private key D_{bob} (2). Since only Bob knows his private key only he can decipher C.

$$M = D_{bob}(C) \qquad (2)$$

Thus, performing public-key encryption and decryption the service of confidentiality can be provided. However further steps should be taken to assure that the message Bob received has really come from Alice since anyone could have encrypted a message by using Bob's public key.

In order to fulfill the confidentiality service, all users need to have confidence that the public keys they retrieve are authentic and belong to the specified user. For this reason certificates are used. Certificates are issued by a Certification Authority (CA), which is a third party trusted by all users. CA guarantees the link between cryptographic key and the user through signing a document that contains information such as user name, key, name of the CA, expiration date of certificate, user privileges and other related information.

2.4.2. Digital signatures - *data origin authentication, data integrity, non-repudiation*

Digital signatures are the electronic equivalent of traditional handwritten signatures. Because electronic information is easy to duplicate electronic

signatures are formed in a different way than hand-written signatures.

Public-key cryptosystem for signing is different than its use in encryption. The difference is the order in which the keys are used is reversed. Recall that in data encryption, first Alice applied E_{bob} to M, then Bob decrypted using D_{bob}. In digital signatures, first Bob applies D_{bob} to compute his signature, and then Alice verifies the signature using E_{bob}.

Consider the case where Bob wants to sign a message M. Bob first hashes M using a hash function. The output, which is denoted $h(M)$, is called a message digest. M is signed by transforming $h(M)$ using D_{bob} (3). Bob then sends M and S to Alice as his signature on M.

$$S = D_{bob}(h(M)) \qquad (3)$$

To verify Bob's signature on M, Alice first retrieves E_{bob} from public directory, recomputes the message digest, $h(M)$, from M using the publicly known hash function and finally transforms S using E_{bob} and compares the result with $h(M)$. If Alice finds out that $E_{bob}(S)$ *is equal to* $h(M)$ then she accepts Bob's signature as valid. On the contrary, if it is not equal then Alice concludes that S is not Bob's signature for message and M has been altered.

It is easy to see that this scheme provides the services of data origin authentication, data integrity, and non-repudiation. Data origin authentication is provided because Alice is assured that only Bob could have computed the signature S since only Bob knows his decryption key, D_{bob}, and therefore only Bob could have transformed $h(M)$ to S. Data integrity is provided because once Bob signs M, the message cannot be altered, since changing M would change $h(M)$, so that S would no longer be a valid signature on the new message. Non-repudiation is provided because once Bob signs a message, he cannot later reject the fact that he signed it. To show this, Alice should save Bob's signature on M so that if Bob later denies signing M, the pair of M and S can prove that he is lying.

2.4.3. Cryptographic protocol: signed challenges -user authentication

Next, we need to show that how the use of a public-key cryptosystem provides user authentication. A Digital signature alone does not provide user authentication. This can be shown by the fact that once Bob has signed M by S, an adversary can save this a signature and use it to authenticate her as Bob at any later time. To overcome this, a cryptographic protocol is needed. A protocol is a sequence of messages and responses passed between Alice and Bob that provide one or both parties with some service.

The interaction between Alice and Bob is modified in following way: When Alice enters a real-time communication with Bob and wants to authenticate Bob, she first generates an unpredictable random number, which is called a *challenge.* Alice then sends the challenge to Bob. Now, Bob includes this random number before he signs. Alice now can be assured that she is communicating with Bob in "real-time". Thus, user authentication can be provided by the combination of a digital signature together with the unpredictability of Alice's random challenge.

2.5. MATHEMATICAL PROBLEMS FOR CRYPTOSYSTEMS

Public-key cryptographic systems base their security on the difficulty of solving mathematical problems. Many of the systems have been broken or shown to be impractical. Currently, only three types of systems are considered both secure and efficient. Examples of these systems and the underlying mathematical problems that their security is based are:

1. Integer factorization problem: Some well known examples are RSA and Rabin-Williams
2. Discrete logarithm problem (DLP): DSA, the Diffie-Hellman key agreement scheme, the ElGamal encryption and signature schemes, the Schnorr signature scheme, and the Nyberg-Rueppel signature scheme. Mathematicians have extensively studied the DLP for the past 20 years
3. Elliptic curve discrete logarithm problem (ECDLP): The elliptic curve

analog of the DSA , and the elliptic curve analogs of the Diffie-Hellman key agreement scheme, the ElGamal encryption and signature schemes, the Schnorr signature scheme, and the Nyberg-Rueppel signature scheme

None of the above problems have been proven to be intractable (i.e., difficult to solve in an polynomial time). However, they are believed to be intractable since years of intensive study by leading mathematicians and computer scientists have failed to yield efficient algorithms for solving them.

The security of ECC comes from the computational intractability of ECDLP. Since ECDLP is a different form of DLP both problems will be discussed below.

2.5.1. Discrete logarithm problem (DLP)

Let p be a prime number, then Zp denotes the set of integers $\{0, 1, 2, \ldots p - 1\}$, where addition and multiplication are performed modulo p. Then given an integer $a \in Zp$, and b which is the result of exponentiation of a, there exist following relation between a and b such that $b = a^e \pmod{p}$ for some e. The discrete logarithm problem modulo p is to determine the integer e for a given pair of b and a. The integer e is called the discrete logarithm of b to the base a. There is no efficient algorithm is known to date to solve the discrete logarithm problem modulo p.

2.5.2. The elliptic curve discrete logarithm problem (ECDLP)

Though it will later be discussed in detail, an elliptic curve, defined modulo a prime p, is the set of solutions (x, y) to an equation $y^2 = x^3 + ax + b \pmod{p}$, for two numbers a and b. Any (x, y) solutions to the above equation are a point on the elliptic curve. The elliptic curve discrete logarithm problem is as follows. Let p be a prime number and P be a point on the elliptic curve. Multiplying P by x times is simply addition of P to itself by x times. Suppose Q is a multiple of P, so that $Q = xP$ for some x. Then the "elliptic curve discrete logarithm problem" would be given P and Q find x.

2.6. SECURITY AND EFFICIENCY COMPARISON OF PUBLIC-KEY CRYPTOSYSTEMS

2.6.1. Security

It is clear that breaking the system really requires solving the underlying mathematical problem. It is unfortunate that there is no polynomial time algorithm to tell us which problem, the integer factorization problem, the discrete logarithm problem modulo p, or the elliptic curve discrete logarithm problem, takes fully exponential time. However, the elliptic curve discrete logarithm problem is currently considered harder than either the integer factorization problem or the discrete logarithm problem modulo p. To show this, we can compare the time required to break the ECC with the time required breaking RSA or DSA for various modulus sizes using the best general algorithm known. To achieve reasonable security, RSA and DSA should employ a 1024-bit modulus, while ECC needs a 160-bit modulus. Furthermore, 300-bit ECC is a great deal more secure than 2000 bit RSA or DSA.

2.6.2. Efficiency

Three distinct factors determine the efficiency of a public-key cryptographic system:

1. Computational overheads – amount of computation required performing the public key and private key transformations.
2. Key size – bit size to store the key pairs and any system parameters.
3. Bandwidth – amount of bits that must be communicated to transfer a signature or an encrypted message.

In order to be consistent, the comparisons will be made between systems offering similar levels of security i.e., 160-bit ECC will be compared with 1024-bit RSA and DSA.

2.6.3. Computational overheads

In RSA, with the expense of some security risks, a short public exponent can be

used to speed up signature verification and encryption. In both ECC and DSA, a large proportion of the signature generation and encrypting transformations can be pre-computed. With a good implementation of all three systems it can be shown that ECC is almost 10 times faster than either RSA or DSA.

2.6.4. Key size

Comparing the size of the system parameters and key pairs for the different systems shows that the system parameters and key pairs are shorter for the ECC than RSA and DSA (Table 1).

	System parameters (bits)	Public key (bits)	Private key (bits)
RSA	n/a	1088	2048
DSA	2208	1024	160
ECC	481	161	160

Table 1: Size of system parameters and key pairs (approximate value)

2.6.5. Bandwidth

When long messages are used for encryption or signature, all three types of systems show similar bandwidth requirements. However, since public-key cryptographic systems are often employed to transmit short messages it is best to compare all three systems with long and short messages. Suppose that each of above systems is being used to sign a 2000-bit message, or to encrypt a 100-bit message. The lengths of the signatures and encrypted messages are shown in tables 2 and 3, respectively.

	Signature size (bits)
RSA	1024
DSA	320
ECC	320

Table 2: Signature sizes on 2000-bit messages

	Encrypted message (bits)
RSA	1024
ElGamal	2048
ECC	321

Table 3: Size of encrypted 100-bit message

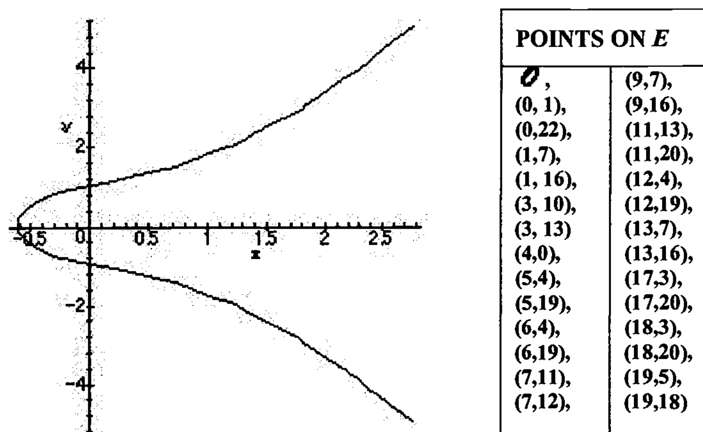
Analyzing tables clearly shows that ECC offers considerable bandwidth savings over the other types of public-key cryptographic systems when used to transform short messages. Based on the computational overheads, key size and bandwidth requirements it can be shown that the ECC provides greater efficiency than either integer factorization systems or discrete logarithm systems. Of course, once implemented, all these savings mean higher speeds, lower power consumption, and reduced code size.

2.7. WHAT ARE ELLIPTIC CURVES

An *elliptic curve E* over Z_p is defined by the following equation: $y^2 = x^3 + ax + b$, where $a, b \in Z_p$, and $4a^3 + 27b^2 \not\equiv 0 \pmod p$, along with a special point $\mathcal{O}$, called the *point at infinity* or *the zero point.* The set $E(Z_p)$ consists of all points (x, y) such that $x \in Z_p$, $y \in Z_p$ that satisfy the equation $y^2 = x^3 + ax + b$, together with $\mathcal{O}$. The point at infinity is a special point that is defined especially to take care of cases when an addition would otherwise remain undefined.

All of the above explanations relate to elliptic curves over the field of real numbers. As we will see, when elliptic curves are solved over other fields, the formulas for them will change a little.

Consider the following example for *elliptic curve over* $\mathbf{Z}_{23}$. Let p=23, a=1, and b=1 with elliptic curve E: $y^2 = x^3 + x + 1$ defined over $\mathbf{Z}_{23}$. One can easily verify that above equation indeed describes and elliptic curve such as: $4a^3+27b^2 = 4 + 4 = 8 \neq 0$, so E is indeed an elliptic curve. The E along with its points is given in figure 1.



POINTS ON *E*	
$\mathcal{O}$,	(9,7),
(0, 1),	(9,16),
(0,22),	(11,13),
(1,7),	(11,20),
(1, 16),	(12,4),
(3, 10),	(12,19),
(3, 13)	(13,7),
(4,0),	(13,16),
(5,4),	(17,3),
(5,19),	(17,20),
(6,4),	(18,3),
(6,19),	(18,20),
(7,11),	(19,5),
(7,12),	(19,18)

Figure 1: The graph of $y^2=x^3+ax+b$, where $a=1$, $b=1$

Once we have determined an elliptic curve, certain mathematical operations can be defined: namely addition and multiplication that can be determined geometrically or by algebraic equations. From the geometrical point of view, rules for addition over elliptic curves can be defined from a simple rule: If three points on an elliptic curve lie on a straight line, their sum is $\mathcal{O}$. And the rules for addition can be defined as follows:

1. $\mathcal{O}$ is the additive identity, i.e., $\mathcal{O} = -\mathcal{O}$. For any point P on the elliptic curve, $P + \mathcal{O} = P$
2. The negative of a point say $P_1=(x, y)$ is a point with the same x coordinate and but negative y coordinate. Note that the new point is also in the curve
3. To add two points on an elliptic curve E, say $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ with the different x coordinates, draw a straight line between them and find the third point of intersection on the curve and then take the reflection of this point in the *x-axis*. Let $R = (x_1, y_1)$ be point coming from the result of addition of P and Q. The figure 2 depicts how R is calculated.

4. To double point P, draw a tangent line and find the other point of intersection and take its reflection in the x-axis R. Then P + P = 2P = R (figure 3)

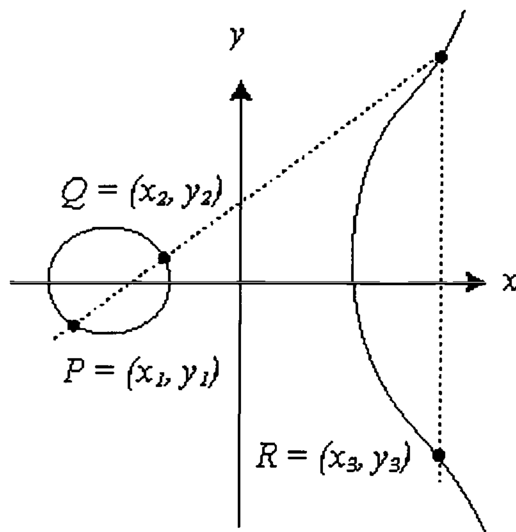


Figure 2: Geometric description of the addition of two distinct elliptic curve points

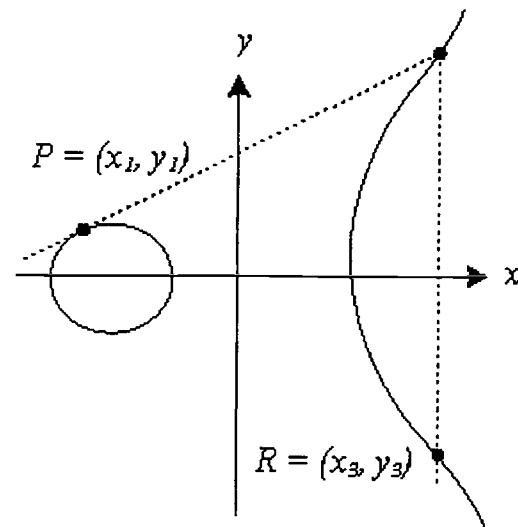


Figure 3: Geometric description of the doubling of an elliptic curve point

Multiplication of a point Q on an elliptic curve E by a positive integer is m is defined as a sum of m copies of Q. Thus, 2Q =Q+Q, 3P = P+ P+P and so on.

2.7.1. Elliptic curves cryptography and elliptic curve groups

Elliptic curves are generally used to form elliptic curve groups. A group is a set of elements with custom-defined arithmetic operations on those elements. In the case of elliptic curve groups, operations are defined geometrically. By introducing fixed properties to the elements of an elliptic curve group, such as making the number of points on such a curve finite, one can create an underlying field formed for an elliptic curve group. A field represents a set of numbers for which there are certain arithmetic operations defined, usually addition and subtraction. Above examples

were considering the infinite field of real numbers in order to illustrate the geometrical properties of elliptic curve groups. There exist numerous other infinite fields such as, whole numbers, imaginary numbers, and fractional numbers. For cryptographic purposes infinite fields are not useful. Thus, in the case of ECC cryptography, the curves are defined over specific more easily constrained finite fields. This will prevent problems such as round-off errors and infinitely repeating fractions. Most common fields are F(p) and F(2^m).

F(p) is a finite field that is commonly used for ECC. It is a field of whole numbers from 0 to p-1, where p is a prime number. Any calculations in F(p) are made modulo p so that the result will still fall into the F(p) space. Based on this the equation for elliptic curve (4) and determination of legality of an a and b pair (5) can be written as:

$$(y^2) \bmod p = (x^3 + ax + b) \bmod p \tag{4}$$

$$(4a^3 + 27b^2) \bmod p \tag{5}$$

Figure 4 shows the elliptic curve over field F_{23}, where a=1 and b=0 and the elliptic curve equation is $y^2 = x^3 + x$.

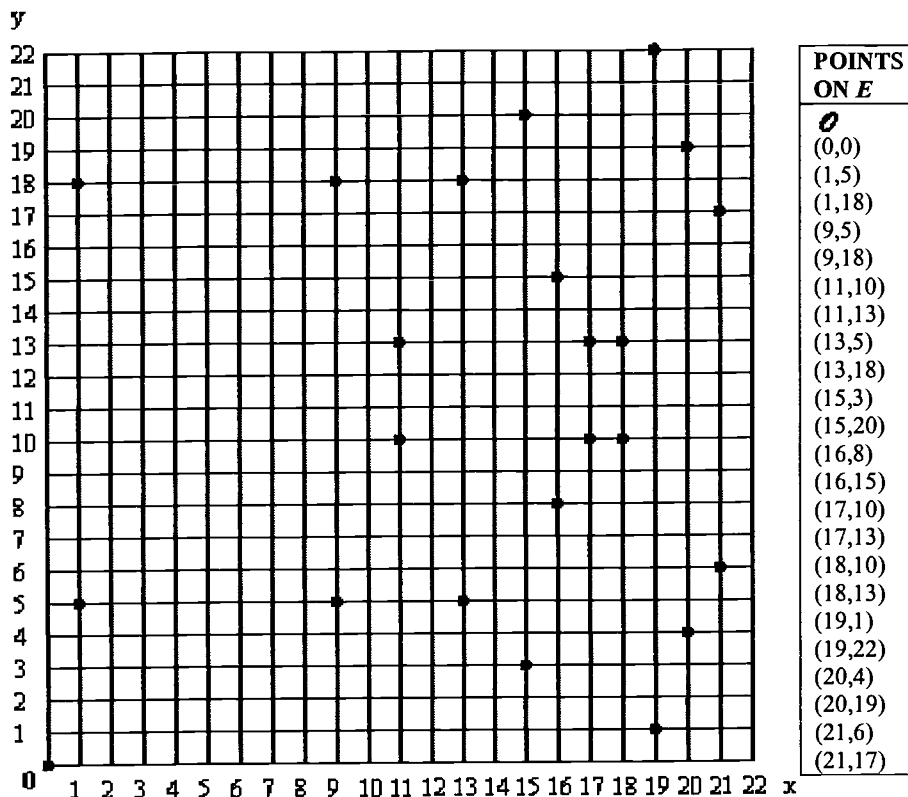


Figure 4: Elliptic Curve points: $y^2 = x^3 + x$ over F_{23}

It is easy to show that the point (9,5) satisfies this equation since:

$y^2 \bmod p = x^3 + x \bmod p$

52 mod 23 = 93 + 9 mod 23

25 mod 23 = 738 mod 23

2 = 2

The 23 points that satisfy this equation is shown in figure 4.

It should be noted that elliptic curves drawn in F(p) and in the field of real numbers do not look same. Because only a small set of whole numbers are allowed rather than having neat and smooth curves, the elliptic curves drawn in F(p) will have disjoint points Despite this the mathematical formulas for addition and multiplication still work. These rules are exactly the same as those for elliptic curve groups over real numbers, with the exception that computations are performed modulo p.

2.7.2. Arithmetic in an elliptic curve group over GF(p)

The arithmetic of the Galois field GF(p) is the usual mod p arithmetic. One of the main benefits of having elliptic curve groups GF(p) over real numbers is that the former has a finite number of points, which is desirable for cryptographic purposes. Unlike elliptic curves over real numbers, computations over the field of GF(p) involve no round off error. Suppose $P = (x_P, y_P)$, $Q = (x_Q, y_Q)$ and $R = (x_R, y_R)$, the addition and doubling can be described as follows.

2.7.2.1. <u>Adding distinct points on elliptic curve over GF(p)</u>

The negative of the point P is the point -P = (x_P, $-y_P$ mod p). If P and Q are distinct points,

i.e., x coordinates are not the same, such that P is not -Q, then P + Q = R where

$\lambda = (y_P - y_Q) / (x_P - x_Q) \bmod p$

$x_R = \lambda^2 - x_P - x_Q \bmod p$ and

$y_R = -y_P + \lambda(x_P - x_R) \bmod p$

2.7.2.2. <u>Doubling the point on elliptic curve over GF(p)</u>

Let P = (x_P, y_P) be a point on the curve, provided that $y_P \neq 0$,

2P = R where

$\lambda = (3x_P^2 + a) / (2y_P) \bmod p$

$x_R = \lambda^2 - 2x_P \bmod p$ and

$y_R = -y_P + \lambda(x_P - x_R) \bmod p$

2.8. ECDSA PROTOCOL

The Elliptic Curve Digital Signature Algorithm is a widely standardized signature scheme. Below is given the key generation, signature generation, and signature verification procedures for ECDSA.

2.8.1. ECDSA key generation

Each entity A does the following:

- Select an elliptic curve E defined over Z_p. The number of points in $E(Z_p)$ should be divisible by a large prime n
- Select a point $P \in E(Z_p)$ of order n
- Select a statistically unique and unpredictable integer d in the interval [1, n-1]
- Compute $Q = dP$
- A's public key is (E, P, n, Q); A's private key is d

2.8.2. ECDSA signature generation

Let m denote the message to be signed by A. A does the following:

- Select a unique and unpredictable integer k in the interval [1, n-1]
- Compute kP = (x_1, y_1) and $r = x_1 \bmod n$. (If r is equal to 0, go to step 1, this is for security purposes as if r is equal to zero then the signing equation $s = k^{-1}\{h(m)+dr\}$ mod n does not involve the private key d)
- Compute $k^{-1} \bmod n$
- Compute $s = k^{-1}\{h(m)+dr\} \bmod n$, where h is the Secure Hash Algorithm (SHA-1)
- If s = 0, then go to step 1. (If s = 0, then $s^{-1} \bmod n$ does not exist; s^{-1} is required in step 3 of signature verification)
- The signature for the message m is the pair of integers (r, s)

2.8.3. ECDSA signature verification

The Entity B does the following to verify A's signature (r, s) on m:

- Obtain an authentic copy of A's public key (E, P, n, Q)
- Verify that r and s are integers in the interval [1, n-1].
- Compute $w = s^{-1} \bmod n$ and $h(m)$.
- Compute $u_1 = h(m)w \bmod n$ and $u_2 = rw \bmod n$.

- Compute $u_1P + u_2Q = (x_0, y_0)$ and $v = x_0 \bmod n$.
- Accept the signature if and only if $v = r$

ANSI X9.62, and NIST mandates that $n > 2^{160}$. Instead of each user generating its own elliptic curve, the same curve E over Zp, and point P of order n can be used for both users; these quantities are called *system parameters* or *domain parameters*. Thus, an entity's public key consists only of the point Q, which in turn will lead to smaller-sized public keys. Because it is difficult to generate system parameters, i.e., E, P and n, and they are public, their generation can be audited and checked for validity.

3. OPTIMIZATION OF ECDSA ALGORITHM

To achieve a faster optimization technique, it is important to identify performance bottlenecks. To do this a profiler was used and observed that almost 75 percent of the time was spent on the Montgomery multiplication algorithm. This is because the Public key cryptography needs high-speed and space-efficient algorithms for modular multiplication. Elliptic curve cryptosystems over the field GF(p) and the Diffie-Hellman key exchange algorithm can be such examples. The Montgomery multiplication algorithm, due to Peter L. Montgomery, is considered one of the best and most useful algorithms in modular arithmetic. This algorithm is used in ECDSA. This section first introduces the algorithm used in the project then shows how it was optimized. After determining the bottlenecks the aim was to redesign the program rather than do trivial optimization techniques, which are generally done by the compiler.

3.1 MONTGOMERY MULTIPLICATION

The Montgomery multiplication algorithm is an efficient way to compute the modular multiplications and squaring required during the modular multiplication process. It computes fast multiplication by replacing division by a modulus n with division by a power of 2. It is used to compute the modular product of two integers in the following way (6):

$$c = a.b \pmod{n}, \tag{6}$$

where a, b and n are k-bit binary numbers, i.e., $2^{k-1} \leq a, b, n < 2^k$ and the resulting k-bit number c in (6) will be produced without performing a division by the modulus n.

The Montgomery algorithm computes (7)

$$MontMult(a, b) = a.b.r^{-1} \pmod{n} \tag{7}$$

where a, $b < n$ and r such that $gcd(n, r) = 1$. Even though the algorithm works for any r that is relatively prime to n, it gives better efficiency when r is taken to be a power of 2. Choosing r as value of 2^k will yield division by power of 2, which is a fast operation on general-purpose computers and this will yield a faster and simpler implementation of modular multiplication.

In order to describe Montgomery multiplication we need to first define the n-residue of an integer a < n as (8)

$$\bar{a} = a \, .r \,(\text{mod}\ n) \tag{8}$$

The set $\{a.r \ (\text{mod}\ n) \mid 0 \le a \le n\text{-}1\}$ is a complete residue system, i.e., it contains all numbers between 0 and n-1. Therefore, a one-to-one correspondence between the numbers in the range 0 and n-1 and the numbers in the above set is possible.

The Montgomery multiplication algorithm takes advantage of this and computes the n-residue of the product of two integers whose n-residue is given. Therefore, given two n-residue $\bar{a}$ and b, the n-residue Montgomery product is defined as (9)

$$\bar{c} = \bar{a} \,.\, \bar{b}.\, r^{-1} (mod\ n) \tag{9}$$

where r^{-1} is $1/r$ mod n, i.e., it is the number with the property $r^{-1}.r = 1$ (mod n). It can be shown that the resulting product c is indeed the n-residue of the product $c = a.b$ (mod n), because

$$\begin{aligned} \bar{c} &\equiv \bar{a} \,.\, \bar{b}.\, r^{-1} \ (\text{mod}\ n), \\ &= a.r.b.r.r^{-1} \ (\text{mod}\ n) \\ &= a.r.b \ (\text{mod}\ n) \\ &= c.r \ (\text{mod}\ n). \end{aligned}$$

An additional quantity n', an integer with the property $r.r^{-1}\text{-}n.n' = 1$ is needed for the Montgomery multiplication algorithm. By using the extended Euclidean algorithm n' and r^{-1} can easily be computed.

The Montgomery product computation as follows:

function MontMult($\bar{a}$, $\bar{b}$)
Step 1. $t := \bar{a} \,.\, \bar{b}$
Step 2. $u := (t + (t \,.\, n' \bmod r) \,.\, n\,) \,/\, r$
Step 3. if $u \geq n$ then return u-n else return u

Note that the Montgomery multiplication algorithm involves multiplication modulo r and division by r rather than division by n. Since r is a power of 2, operations involving r are intrinsically faster. However, since converting from residue to n-residue, computing n', and converting the result back to ordinary residue are time-consuming operations, the Montgomery product computation algorithm is inefficient when a single modular multiplication is needed. It is rather more suitable when several modular multiplications with the same modulus n are needed, such as modular exponentiation. In this case, the exponentiation operation is performed by a series of squaring and multiplication operations modulo n. Suppose we want to compute $b = a^e$ (mod n). Let j be the number of bits in the exponent e. The exponentiation algorithm given below computes $b = a^e$ (mod n) with O(j) calls to the Montgomery multiplication algorithm. Note that step 4 of the modular exponentiation algorithm computes x using x via the property of the Montgomery algorithm: $MontMult(\bar{x}, 1) = x.\, r^{-1} = x.\, r.\, r^{-1} = x \bmod n$.

function ModExp(a, e, n)
Step 1. $\bar{a} = a.r \bmod n$
Step 2. $\bar{x} = 1 \,.\, r \bmod n$
Step 3. for $i := j$-1 downto 0
    Step 4. $\bar{a}$ = MontMult($\bar{x}$, $\bar{x}$)
    Step 5. if $e_i = 1$ then $\bar{x}$ = MontMult($\bar{x}$, $\bar{a}$)
Step 6. return x := MontMult($\bar{x}$, 1)

Step 3 uses the Montgomery multiplication operation and it only performs multiplication modulo 2^k and division by 2^k. The result produced is in n residue form and conversion to the ordinary residue can be obtained by calling Montgomery multiplication with arguments x and 1. To verify this note that since $\bar{x}$=$x.r$ (mod n) it implies that

$$
\begin{aligned}
x &= \bar{x}.\, r^{-1} \ (\text{mod } n) \\
&= \bar{x}.\, r^{-1}.1 \ (\text{mod } n) \\
&= \text{MontMult}(\bar{x}, 1)
\end{aligned}
$$

In next section, the optimization techniques will be applied to ECDSA. The time analysis is performed by counting the total number of multiplications, additions/subtractions, and memory read and write operations in term of the size of the input parameter s. As an illustration consider following two operations, where C stands for Carry and S stands for Sum.

$$
\begin{aligned}
&(C, S) := t[i] + a[i] * p[j] + C \\
&t[j] := t[i] + m
\end{aligned}
$$

The first one requires three memory reads, two additions, and one multiplication while the second one requires two memory reads, one addition, and one memory write.

3.2 MONTGOMERY MULTIPLICATION OPTIMIZATION

Current implementation of the Montgomery multiplication, given in table 4, integrates multiplication and reduction steps. In other words, rather than computing the entire product of a and b, then reducing, it alternates between iterations of the outer loops for multiplication and reduction. This is possible because the value of m in the ith iteration of the outer loop for reduction depends only on the value $tmp[i]$, which is completely computed by the ith iteration of the outer loop for the multiplication. The Montgomery multiplication algorithm given in figure 5 is used in ECDSA. It computes $t = a*b*m \ (mod\ p)$, where $m=2^{-h}$, p is a prime number and m is the number of bits in p (also known as the order of the underlying field).

The algorithm has three main sections. The first section is for initialization of array tmp and t, the second section, which we will call as main loop, is the place where all the other necessary computations are performed and the last section is where carry correction is done. The main loop itself can be divided into two sections, which are the multiplication section, and the reduction section.

```
for i=0 to s-1                                   /*initialization */
        tmp[i] :=0, t [i] :=0
tmp[s] :=0

for i=0 to s-1
        C:=0                                     /*multiplication */
        for j=0 to s-1
                (C,S) := tmp[j] + a[j]*b[i] + C
                tmp[j] := S
        (C,S) := tmp[s] + C
        tmp[s] := S
        tmp[s+1] := C

        C := 0                                   /* reduction */
        m := tmp[0]*P0Prime mod 2^WORDSIZE
        (C,S) := tmp[0] + m*p[0]
        for j=1 to s-1
                (C,S) := tmp[j] + m*p[j] + C
                tmp[j-1] := S
        (C,S) := tmp[s] + C
        tmp[s-1] := S
        tmp[s] := tmp[s+1] + C

if tmp[s] == 0                                   /* carry correction */
        for i=0 to s-1
                t[i] = tmp[i]
else
        C = 1
        for i=0 to s-1
                (C,S) := tmp[i] + ~p[i] + C
                t[i] = S
```

Table 4. Classical Montgomery Multiplication Algorithm

Since the Montgomery multiplication algorithm is used extensively (around 75 % of the time) in current ECDSA implementation much effort was spent on the improvement of this algorithm. Another reason to choose the Montgomery multiplication as a beginning point was that the prime modulus p of *Curve P-224 (*see Appendix*)* that has a special structure is extensively used in this algorithm. Thus, by exploiting the special structure of the prime modulus an efficient signature implementation was aimed in this thesis work. The prime modulus p of the curve in hexadecimal notation is in the form

p = 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0x00000000, 0x00000000, 0x00000001. Considering each word has 32 bits, this prime modulus is seven

words and starting from the least significant word (from the right) the first word is equal to one, the second and third word is equal to zero and the remaining four words are 0xffffffff.

Despite the fact that the optimization done in this thesis work was for this special curve and for the signature algorithm, some optimizations performed can easily be adapted to other curves regardless of which curve is being used.

The first optimization performed was removal of temporary array *tmp* used in the algorithm. Note that the carry correction section of the code copies *directly* the temporary array *tmp* to output array *t*. If *t* is used in place of *tmp,* the *tmp* array along with its initialization and the *if* section will no longer be needed. To accommodate this change, the else section will be converted with the opposite of *if* condition and a new variable *t7* and *t8* will be introduced to hold the value of $t[s]$ and $t[s+1]$ as the output array can only hold s elements inside it (table 5).

To continue with other optimizations the main loop needed to be unrolled. Loop unrolling, if used properly, can be advantageous. However, an unrolled loop is larger than the "rolled" version. Carelessly unrolled loops will increase the number of instructions. In limited environments the instruction cache is important for efficiency, thus filling the instruction cache for instructions that will be used less time can reduce performance. So, it is desirable to unroll the loop if significant changes are going to be made. Consider we are unrolling the main loop in its first iteration. The new algorithm is given in table 6.

```
for i=0 to s-1
/*initialization */
        t[i] := 0
t7 := 0, t8 := 0

  for i=0 to s-1
        C:=0                    /*multiplication */
        for j=0 to s-1
                (C,S) := t[j]+a[j]*b[i]+C
                t[j] := S
        (C,S) := t7 + C
        t7 := S
        t8 := C

        C := 0                  /* reduction */
        m:= t[0]*P0Prime mod
  2^WORDSIZE
        (C,S) := t[0] + m*p[0]
        for j=1 to s-1
                (C,S) := t[j]+m*p[j]+C
                t[j-1] := S
        (C,S) := t7 + C
        t[s-1] := S
        t7 := t8 + C

    if t7 != 0              /* carry correction */
            C = 1;
            for i=0 to s-1
                    (C,S) := t[i]+~p[i]+C
                    t[i] = S
```

Table 5. Montgomery Multiplication Algorithm after removal of temporary array *tmp*

```
for i=0 to s-1                          /*initialization */
        t[i] :=0
t7:=0, t8:=0

 i=0
        C:=0
  /*multiplication */
        for j=0 to s-1
                (C,S) := t[j]+a[j]*b[i]+C
                t[j] := S
        (C,S) := t7 + C
        t7 := S
        t8 := C

        C := 0                  /* reduction */
        m:= t[0]*P0Prime mod 2^WORDSIZE
        (C,S) := t[0] + m*p[0]
        for j=1 to s-1
                (C,S) := t[j] + m*p[j] +C
                t[j-1] := S
        (C,S) := t7 + C
        t[s-1] := S
        t7 := t8 + C

 for i=1 to s-1
  SIMILAR TO ABOVE MULTIPLICATION
  SECTION

  SIMILAR TO ABOVE REDUCTION
  SECTION

    if t7 != 0              /* carry correction */
            C = 1;
            for i=0 to s-1
                    (C,S) := t[i]+~p[i]+C
                    t[i] = S
```

Table 6. The Montgomery multiplication algorithm after unrolling for first iteration

The multiplication section in the first iteration of main loop, given below, has special information that can be exploited. Inside the *for* loop *t*[*j*] for each *j* is used in addition operation and then it is assigned to the value held by S. However, in the first iteration of main loop, *t*[*j*] initially holds zero for each loop iteration which makes it obsolete to use in the addition.

```
C:=0                    /*multiplication */
for j=0 to s-1
        (C,S) := t[j] + a[j]*b[i] + C
        t[j] := S
(C,S) := t7 + C
t7 := S
t8 := C
```

The addition of *t[j]* can be removed, giving a single addition rather than 2 additions. Similarly, *t7* initially holds zero thus the operation (*C*, *S*) := *t7* + *C* will only yield *C* in *S* and zero in *C*. Since *t8* is now zero the assignment of *t8* can be removed. To accommodate this in the reduction section of first iteration of main loop the addition of *t8* will also be removed. Thus, the new multiplication section for the first main loop iteration becomes:

```
C:=0     /*multiplication */
for j=0 to s-1
        (C,S) :=  a[j]*b[i] + C
        t[j] := S
t7 := C
```

Furthermore, since in each loop iteration assignments to *t[j]* and *t7* are done before the values they hold are used, it is no longer necessary to do initialization in the initialization section, thus the initialization loop can be removed.

The last optimization that can be done in the multiplication loop is the unrolling of the first iteration. Since *C* holds zero in the *first* iteration there is no need to add *C*. Furthermore, since C will be assigned to a value before it is used its initialization can also be removed.

All these optimization can be applied to all curves provided that the main loop is unrolled for the first iteration. However the last optimization involving *C* can be applied to all other loops as well. The new multiplication section for the first iteration and other iteration of main loop is given in table 7.a and 7.b, respectively.

(a)	(b)

(C,S) := a[0]*b[0] /*multiplication first iteration of main*/
t[0] := S
for j=1 to s-1
    (C,S) := a[j]*b[i] + C
    t[j] := S
t7 := C
	(C,S) := a[0]*b[0] /*multiplication other iteration of main*/
	t[0] := S
	for j=1 to s-1
	    (C,S) := t[j] + a[j]*b[i] + C
	    t[j] := S
	(C,S) := t7 + C
	t7 := S
	t8 := C

Table 7. Multiplication in first iteration (a) , other loop iterations (b)

Next optimization involves the reduction section. Note that the reduction section shown below is different from than the first reduction section in the last operation. In the first iteration of main loop it is *t7=C.*

```
C := 0                 /* reduction */
m := t[0]*P0Prime mod 2^WORDSIZE
(C,S) := t[0] + m*p[0]
for j=1 to s-1
        (C,S) := t[j] + m*p[j] + C
        t[j-1] := S
(C,S) := t7 + C
t[s-1] := S
t7 := t8 + C
```

Since $p[0]$ is equal to one the multiplication involving $p[0]$ can be removed, i.e., $(C, S)= t[0]+m*p[0]$ becomes $(C, S) = t[0]+m$. Next, we will be unrolling the *for* loop for the first three iterations, this is because p[1] and p[2] are equal to zero, and $p[3\text{-}6]$ are equal to 0xffffffff and this information can be used to remove certain operations such as multiplication of $m*p[j]$ for j=1 and j=2. The above section of the algorithm after unrolling becomes:

```
C := 0  /* reduction */
m := t[0]*P0Prime mod 2^WORDSIZE
(C,S) := t[0] + m

(C,S) := t[1] + C
t[0] := S

(C,S) := t[2] + C
t[1] := S
```

```
(C,S) := t[3] + m*p[3] + C
t[2] := S

for j=4 to s-1
        (C,S) := t[j] + m*p[j] + C
        t[j-1] := S
(C,S) := t7 + C
t[s-1] := S
t7 := t8 + C
```

The pseudo code $(C, S)= t[0]+m$ is the addition of two single words namely, $t[0]$ and m. This addition will yield C either 0 or 1. Since we know that C can be only 1 and 0 a test case can be included. There are two ways to exploit this special situation. First case is given in table 8.

```
C := 0  /* reduction */
m := t[0]*P0Prime mod 2^WORDSIZE
        (C,S) := t[0] + m

if (t[1] != 0xffffffff)
        t[0] := t[1] + C
        t[1] := t[2]

        n= m*p[3]
        (C,S) := t[3] + n
        t[2] := S
else
        (C,S) := t[1] + C
        t[0] := S

        (C,S) := t[2] + C
        t[1] := S

        n= m*p[3]
        (C,S) := t[3] + n + C
        t[2] := S
for j=4 to s-1
        (C,S) := t[j] + n + C
        t[j-1] := S

(C,S) := t7 + C
t[s-1] := S
t7 := t8 + C
```

Table 8. Reduction with array condition check

In the above implementation, $t[1]$ and $t[2]$ are first checked if they are equal to 0xffffffff. This is because $(C,S)= t[0]+m$ will yield C as 0 or 1 and if $t[1]$ is not equal to 0xffffffff regardless of carry the addition will not yield a carry. Since p[1] and p[2] are equal to zero, operations involving them can be eliminated. In the third case we introduce a new variable n to hold the value of $m*p[3]$ because $p[3\text{-}6]$ are the same so by introducing a new variable the computation involving the multiplication of $m*p[j]$ for $4< j <7$ will be saved. One point that needs to be considered is the case $t[1]$ and $t[2]$ are equal to 0xffffffff (if condition is not satisfied) in which case the control will enter into the else section and normal computations will be performed. However, it will be too rare for this case to happen, thus no efficiency will be gained. In fact, the possibility that the else section will be entered is 16^{-8}, or 2^{-32}.

The second case is given in table 9.

```
C := 0  /* reduction */
m := t[0]*P0Prime mod 2^WORDSIZE
        (C,S) := t[0] + m

if (C == 0)
        t[0] := t[1]
        t[1] := t[2]

        n:= m*p[3]
        (C,S) := t[3] + n
        t[2] := S

else
        (C,S) := t[1] + C
        t[0] := S

        (C,S) := t[2] + C
        t[1] := S

        n:= m*p[3]
        (C,S) := t[3] + n
        t[2] := S
for j:=4 to s-1
        (C,S) := t[j] + n + C
        t[j-1] := S
(C,S) := t7 + C
t[s-1] := S
t7 := t8 + C
```

Table 9. Reduction with single condition check

C is checked against 0 or 1. If it is zero, meaning no carry, $t[0]$ will be equal to $t[1]$. Since no carry is generated $t[1]$ will be equal to $t[2]$. Similar to first case discussed above, a new variable n will be introduced to save some computations of $m*p[4\text{-}5]$.

The first case was used when the implementation was done in the C programming language as it gave better efficiency. However, the second case was used when implementation was done in ARM assembly language as it gave better efficiency.

Final optimization for the Montgomery multiplication algorithm involved the section of code that involves carry check and carry correction, which is:

```
if t7 != 0  /* carry correction */
        C = 1
        for i=0 to s-1
                (C,S) := t[i] + ~p[i] + C
                t[i] = S
```

Now suppose that the last iteration of main loop just ended and control entered to the carry correction section. In first iteration of the *for* loop $p[0]$=0x00000001, reverse of this is 0xFFFFFFFE and the addition of C, which is one, will result in 0xFFFFFFFF. Any single word summed with 0xFFFFFFFF will yield one less than that word in the low 32-bit (low word) and only 1 in high 32 bit word (which is C in this respective).

For the case i=1, we know that C is 1 and $p[1]$ and $p[2]$ are both zero ($p[1,2]$=0x00000000), reverse of this as a single word is 0xFFFFFFFF and the addition of C (which is 1) along with $t[1]$ is equal to $t[1]$ in the low word and one in the high word. A similar rule applies for the loop in third iteration (i=2). When i=3, a special case is tested to see if $t[3]$ is NOT equal to 0xFFFFFFFF. If it is equal normal calculations will be performed. If it is not equal then the addition of the carry, which is 1, will not yield a carry thus there is no need to propagate the carry. Recall that reverse of 0xffffffff is equal to 0. The new algorithm becomes:

```
if t[s] != 0  /* carry correction */
        t[0] = t[0]-1
        t[1] = t[1]
        t[2] = t[2]
        if t[3] != 0xffffffff
                t[3]= t[3] +1
                t[4]= t[4]
                t[5]= t[5]
                t[6]= t[6]
        else
                for i=3 to s-1
                        (C,S) := t[i] + C
                        t[i] = S
```

Since some assignments are just themselves they can be removed. The resulting code no longer uses the prime modulus. Can there be any further improvements? The answer is yes. In the else section we know that *t*[3] is 0xffffffff and *C* is 1. Thus *t*[3] will become 0; *t*[4] will become *t*[4]+1, and now we need to check if *t*[5] is equal to 0xffffffff. But this will increase the code size. Furthermore, the chance we will end up in else section is 1 in 16^{-8}, which is equal 2^{-32}. Thus there should be a middle point to choose where to stop or continue. In this algorithm, the else section has not been optimized to achieve less code size. The carry correction section after optimization is given in table 10.

```
if t[s] != 0  /* there is no carry; no correction */
        t[0] = t[0]-1;
            if t[3] != 0xffffffff
                    t[3]= t[3] +1
else
            for i=3 to s-1
                    (C,S) := t[i] + C
                    t[i] = S
```

Table 10. Optimized carry correction section

A final optimization is a restriction of the use of the array *t*. This is because an operation involving an array element will take more clock cycles than an operation involving a simple variable. Thus, seven new variables *t0, t1, t2, t3, t4, t5, t6* were

introduced in place of *t*[0], *t*[1], *t*[2], *t*[3], *t*[4], *t*[5], *t*[6], respectively. The only complication with this is where the assignments to the output array should be. One possibility is to perform all operations with simple variables and just before the function returns assign each variable with its corresponding array element. The second possibility is to unroll the main loop for the last iteration and do final assignments to array elements rather than non-array elements. The latter will be more efficient as there will be no need to assign each variable to its corresponding array element. Since we will be using simple variables the *for* loop in the multiplication and reduction section need to be unrolled.

After all these optimization the new Montgomery multiplication algorithm is given in table 11.

Statements	Statements (continued)

i=0
(C,S) := a[i]*b[0] /*multiplication */
t0 := S
(C,S) := a[i]*b[1] + C
t1 := S
(C,S) := a[i]*b[2] + C
t2 := S
(C,S) := a[i]*b[3] + C
t3 := S
(C,S) := a[i]*b[4] + C
t4 := S
(C,S) := a[i]*b[5] + C
t5 := S
(C,S) := a[i]*b[6] + C
t6 := S
t7 := C
 	
C := 0 /* reduction */
m := t0*P0Prime mod 2^WORDSIZE
(C,S) := t[0] + m
if (t1 != 0xffffffff)
    t0 := t1 + C
    t1 := t2
    n= m*p[3]
    (C,S) := t3 + n
    t2 := S
else
    (C,S) := t1 + C
    t0 := S
    (C,S) := t2 + C
    t1 := S
    n= m*p[3]
    (C,S) := t3 + n
	else
	    (C,S) := t1 + C
	    t0 := S
	    (C,S) := t2 + C
	    t1 := S
	    n= m*p[3]
	    (C,S) := t3 + n
	    t2 := S
	
	(C,S) := t4 + n + C
	t3 := S
	(C,S) := t5 + n + C
	t4 := S
	(C,S) := t6 + n + C
	t5 := S
	(C,S) := t7 + C
	t6 := S
	t7 := C + t8
	
	i = s-1
	(C,S) := t0 + a[i] * b[0]
	t[0] := S
	(C,S) := t1 + a[i]*b[1] + C
	t1 := S
	(C,S) := t2 + a[i]*b[2] + C
	t2 := S
	(C,S) := t3 + a[i]*b[3] + C
	t3 := S
	(C,S) := t4 + a[i]*b[4] + C
	t4 := S
	(C,S) := t5 + a[i]*b[5] + C
	t5 := S
	(C,S) := t6 + a[i]*b[6] + C

```
        t2 := S
(C,S) := t4 + n + C
t3 := S
(C,S) := t5 + n + C
t4 := S
(C,S) := t6 + n + C
t5 := S
(C,S) := t7 + C
t6 := S
t7 :=  C

for i=1 to s-1   /*main loop */
        (C,S) := t0 +  a[i] * b[0]
         t[0] := S
        (C,S) := t1 +  a[i]*b[1] + C
        t1 := S
        (C,S) := t2 +   a[i]*b[2] + C
        t2 := S
        (C,S) := t3 +  a[i]*b[3] + C
        t3 := S
        (C,S) := t4 +   a[i]*b[4] + C
        t4 := S
        (C,S) := t5 +  a[i]*b[5] + C
        t5 := S
        (C,S) := t6 +  a[i]*b[6] + C
        t6 := S
        (C,S) := t7 + C
        t7 := S
        t8 :=C
        C := 0   /* reduction */
        m := t0*P0Prime mod 2^WORDSIZE
        (C,S) := t0+ m
        if (t1 != 0xffffffff)
                t0 := t1 + C
                t1 := t2
                n= m*p[3]
                (C,S) := t3 + n
                t2 := S
```

```
t6 := S
(C,S) := t7 + C
t7 := S
t8 :=C

C := 0   /* reduction */
m := t0*P0Prime mod 2^WORDSIZE
(C,S) := t0+ m
if (t1 != 0xffffffff)
        t[0] := t1 + C
        t[1] := t2
        n= m*p[3]
        (C,S) := t3 + n
        t[2] := S
else
        (C,S) := t1 + C
        t[0] := S
        (C,S) := t2 + C
        t1 := S
        n= m*p[3]
        (C,S) := t3 + n
        t[2] := S
(C,S) := t4 + n + C
t[3] := S
(C,S) := t5 + n + C
t[4] := S
(C,S) := t6 + n + C
t[5] := S
(C,S) := t7 + C
t[6] := S
t7 :=  C + t8

if t7 != 0  /* carry correction */
        t[0] = t[0]-1;
        if t[3] != 0xffffffff
                t[3]= t[3] +1
        else
                for i=3 to s-1
                        (C,S) := t[i] + C
                        t[i] = S
```

Table 11. The optimized Montgomery Multiplication Algorithm

3.3 OTHER OPTIMIZATIONS

All other files were scanned for the possibility to remove initialization loops and the initialization loops were removed in the files that permitted removal of them.

Next optimization involved the combination of adjacent loops that iterate over the same range of the same variable. This technique is called loop jamming.

Consider following two pseudocodes given on table 12. The one on the left is one before the optimization and the one on the right after it is optimized. It can be seen that in each *if* and *else* statements there are two *for* loops that have an index variable namely *pos* ranging from 0 to *s*. Thus, two *for* loops in each conditional statements can be united to give two *for* loops rather than four, as shown in optimized version.

The next optimization is about loop invariant computations, i.e., any part of a computation that does not depend on the loop variable and which is not subject to side effects. They can be moved out of the loop entirely. One might think that most compilers are pretty good at doing this. But there can be cases where compiler is playing it safe in case of side effects. "Computation" can be anything, such as arithmetic computations, array indexing, pointer dereferencing, and even calls to pure functions. Analyzing unoptimized pseudocode shows that there are loop invariant computations, such as the variable *Q->pxdata->wlen* is assigned to same value over the each loop iteration. Thus, it and others can be taken out of the loop (Table 12).

```
if (condition)
        for pos:=0 to s
                Q->pxdata->value[pos] = 0
                Q->pxdata->wlen =1
                Q->pxdata->value[0] =1
        for pos:=0 to s
                Q->pydata->value[pos] = 0
                Q->pydata->wlen =1
else
        for pos:=0 to s
                Q->pxdata->value[pos] = 0
                Q->pxdata->wlen =1
        for pos:=0 to s
                Q->pydata->value[pos] = 0
                Q->pydata->wlen =1
```

```
if(condition)
        Q->pxdata->value[0] =0
        Q->pydata->value[0] =1
        for pos:=1 to s-1
                Q->pxdata->value[pos] = 0
                Q->pydata->value[pos] = 0
        Q->pxdata->wlen =1
        Q->pydata->wlen =1
else
        for pos:=0 to s-1
                Q->pxdata->value[pos] = 0
                Q->pydata->value[pos] = 0
        Q->pxdata->wlen =1
        Q->pydata->wlen =1
```

Table 12. Loop jamming and Loop invariant removal

Another form of optimization involved policies and services. Consider following two pseudocodes taken out of the project (table 13). The caller, function

X, calls function ADD, and they both initialize the same array, which wastes precious execution time.

funct X(some stuff)	**funct** ADD(t1, s, other stuff)

...
for i=0 to s-1
    t1[i] :=0
    t2 [i]:=0
    t3 [i]:=0
ADD(t1, s, other stuff)
...
ADD(t2, s, other stuff)
...
ADD(t3, s, other stuff)
...
	...
	for i=0 to s-1
	    t1[i] :=0
	...

Table 13. Loop initialization decision

Despite that this seems a trivial case, there should be a policy about who should have the responsibility to initialize the array. In this thesis work this responsibility was given to callee, as caller should only care about the service that the callee will give. Thus removed initialization of arrays in functions like *X*.

Since ARM architecture was supporting function inlining, some functions were inlined as part of optimization. Function inlining is used to eliminate the overhead associated with calling and returning from a function by expanding the body of the function inline. Recall that whenever a function calls another function the caller performs some actions such as evaluating parameters and pushing them on the stack and pushing return address on the stack then control is passed to the callee who performs its own operations and control given back to caller. However, if the callee is an inlined function, during compilation the compiler will replace the section of call to inlined function with the body of it. It should be remembered that making functions inline increases the code size. Thus, considering that this thesis work considers the limited space environments functions that have very small code size were made inline.

3.4. ASSEMBLY LANGUAGE IMPLEMENTATION

Despite the fact that codes written in assembly language is difficult to maintain most of the times it gives faster execution time. In order to fully speed up the ECDSA, the Montgomery multiplication algorithm discussed above written in ARM assembly language.

The ARM is a RISC (Reduced Instruction Set Computer). It has all the features of RISC architecture. ARM is used in environments where high performance, low power consumption is needed. It has following RISC features:

- a large register set
- a load-store architecture
- simple addressing modes
- uniform and fixed length instruction fields which simplify instruction decode

Among other additional features, the ARM architecture provides conditional execution of all instructions to maximize data throughput. ARM has thirty-one, 32-bit registers. The sixteen of these are available at any time; the others are used to speed up the exception processing. The ARM instruction set can be divided in to four classes:

- Branch instructions
- Data processing instructions
- Load and store instructions
- Coprocessor instructions

It is worth discussing the Data-processing instructions. These instructions perform operations on the general-purpose registers. There are three types of them:

1. Arithmetic/Logic Instructions: The format of this instructions takes up to two source operands, performs an arithmetic or logic operations on those operands, after that the result is stored into a register and the condition code flags are optionally updated based on the result.
2. Multiply Instructions: There are two types of multiplication instructions

based on the result of the multiplication. The first one has 32-bit result that store the 32-bit result into a register; the other has 64-bit result that is stored in 64-bit results into two registers. Both type of instruction can optionally perform accumulate operation which was used in this thesis work.

3. Status Register transfer instructions: There are two status registers in ARM architecture. The content of those registers can be transferred to or from a general-purpose register.

One of the most important added features of ARM architecture is that all ARM instructions can be conditionally executed. In other words, using those instructions not necessarily mean that they will be executed for sure. Their execution will depend on the values of condition flags that are in the one of the status registers. To achieve conditional execution feature, every instruction contains a 4-bit condition code field. These four condition flags are N, Z, C, and V (Negative, Zero, Carry, oVerflow) and they are collectively known as the condition code flags. Programmer can specify conditional instruction execution and if the condition is satisfied that instruction will be executed. As an illustration consider add instruction given below. The Add instruction has form of:

```
ADD{cond}{S} Rd, Rn, <shifter_operand>
```

It adds the value of "shifter_operand' to the value of register Rn and stores the result in destination register Rd. The "shifter_operand" can be a simple register or an immediate value. If *S* is concatenated to the instruction it will update the condition code flags based on the result. However, it should be noted updating the code flags varies with different instructions. Consider we are executing following instructions:

```
ADD      Rd, R1, R2  ; Rd= R1 + R2
ADDCC    Rd, R1, R2  ; If carry flag is clear perform Rd = R1 + R2
ADDS     Rd, R1, R2  ; Rd= R1 + R2 and update the flags
ADDCCS   Rd, R1, R2  ; If carry flag is clear first perform Rd = R1 +
R2 then and update the flags
```

A big challenge when implementing the Montgomery multiplication algorithm in assembly language is the way to implement double word multiplications. Fortunately, ARM assembly language has very rich instruction set for this purpose. The section of the assembly language that performs the double word multiplication and addition is shown in table 14.b that corresponds to the pseudocode shown in table 14.a.

(a)	(b)
(C,S) := t[j] + m*p[j] + C t[j-1] := S	1. ldr r2, [r7, #+4]! ; load t[j] 2.. adds r2, r2, r3 ; S = t[j] + C_previous 3. mov r3, #0 4. adc r3, r3, #0 ;C = carry (from t[j] + C_previous) 5. umlal r2, r3, r0, r1 ; (C,S) = (C,S) + m * p[j] 6. str r2, [r7, #-4] ; store t[j-1]

Table 14. Double word operations in pseudocode (a), and in ARM assembly language (b)

In above assembly language, the instruction in first line loads *t*[*j*] in register *r2* and then adds *t*[*j*] with *r3* that holds the previous carry (*C*) value. After that *r3* is reset to zero (line 3), then *r3* is added to itself along with zero and one if the carry flag is set. Note that *adc* is a standalone instruction that performs addition with carry flag all the time. In line 5, accumulated multiplication is performed. The register *r0* holds *m* and *r1* holds *p*[*j*]. What this instruction does is it first multiplies *m* and *p*[*j*] and the result, which is a double word, is accumulated with the values stored in *r2* (low order word) and *r3* (high order word). Finally, it stores the *S* into *t*[*j*].

4. RESULTS AND DISCUSSION

In previous chapter optimization techniques that were used discussed. This section will present the result of this work and then will discuss the results. Compilers intrinsically optimize the programs they compile. The compiler probably performs some of the optimizations performed in this thesis work such as removal of loop invariant codes from the loops. However compilers cannot know the design of the program thus their optimization is restricted by rules.

Since the Montgomery multiplication algorithm was the bottleneck in the ECDSA, optimizing this algorithm would promise speed up of the ECDSA. One way to compare the efficiency of several algorithm is to count the number of operations such as the addition, multiplication, memory read or write that the algorithms perform. Thus, the number of multiplication, addition and memory read and memory write for each classical and improved method was counted. Table 15 shows the classical Montgomery multiplication before optimizations, whereas table 16 shows the Montgomery multiplication operation breakdown after the optimization.

STATEMENTS	Operations				Iterations
	Mult	Add	Read	Write	
for i=0 to s /*initialization */	-	-	-	-	-
tmp[i] :=0, t [i] :=0	0	0	0	2	s
tmp[s] :=0	0	0	0	1	1
for i=0 to s-1	-	-	-	-	-
C:=0 /*multiplication */	0	0	0	0	s
for j=0 to s-1	-	-	-	-	-
(C,S) := tmp[j] + a[j]*b[i] + C	1	2	3	0	s^2
tmp[j] := S	0	0	0	1	s^2
(C,S) := tmp[s] + C	0	1	1	0	s
tmp[s] := S	0	0	0	1	s
tmp[s+1] := C	0	1	0	1	s
C := 0 /* reduction */	0	0	0	0	s
m := tmp[0]*n' mod W	1	0	2	1	s
(C,S) := tmp[0] + m*p[0]	1	1	3	0	s
for j=1 to s-1	-	-	-	-	-
(C,S) := tmp[j] + m*p[j] + C	1	2	3	0	s(s-1)
tmp[j-1] := S	0	0	0	1	s(s-1)
(C,S) := tmp[s] + C	0	1	1	0	s
tmp[s-1] := S	0	0	0	1	s

tmp[s] := tmp[s+1] + C	0	1	1	1	s
if tmp[s] == 0 /* carry correction */	-	-	-	-	-
for i=0 to s-1	-	-	-	-	-
t[i] = tmp[i];	0	0	1	1	s
else	-	-	-	-	-
C = 1;	0	0	0	0	1
for i=0 to s-1	-	-	-	-	-
(C,S) := tmp[i] + ~p[i] + C	0	2	2	0	s
t[i] = S	0	0	0	1	s
	s^2+2s	$4s^2+3s$ or $4s^2+5s$	$6s^2+6s$ or $6s^2+7s$	$2s^2+8s+1$	

Table 15: The Montgomery Multiplication operation breakdown before optimization

STATEMENTS	Operations				Iterations
	Mult	Add	Read	Write	
i=0	-	-	-	-	-
(C,S) := a[i]*b[0]	1	0	2	0	1
/*multiplication */	0	0	0	1	1
t0 := S	1	1	2	0	1
(C,S) := a[i]*b[1] + C	0	0	0	1	1
t1 := S	1	1	2	0	1
(C,S) := a[i]*b[2] + C	0	0	0	1	1
t2 := S	1	1	2	0	1
(C,S) := a[i]*b[3] + C	0	0	0	1	1
t3 := S	1	1	2	0	1
(C,S) := a[i]*b[4] + C	0	0	0	1	1
t4 := S	1	1	2	0	1
(C,S) := a[i]*b[5] + C	0	0	0	1	1
t5 := S	1	1	2	0	1
(C,S) := a[i]*b[6] + C	0	0	0	1	1
t6 := S	0	0	0	1	1
t7 := C	0	0	0	0	1
C := 0 /* reduction */	1	0	2	1	1
m := t0* n' mod W	0	1	2	0	1
(C,S) := t0 + m	-	-	-	-	1
if (t1 != 0xffffffff)	0	1	1	1	1
t0 := t1 + C	0	0	1	1	1
t1 := t2	1	0	2	1	1
n= m*p[3]	0	1	2	0	1
(C,S) := t3 + n	0	0	0	1	1
t2 := S	-	-	-	-	1
else	0	1	1	0	1
(C,S) := t1 + C	0	0	0	1	1
t0 := S	0	1	1	0	1
(C,S) := t2 + C	0	0	0	1	1
t1 := S	1	0	2	1	1
n= m*p[3]	0	1	2	0	1
(C,S) := t3 + n	0	0	0	1	1
t2 := S	0	2	2	0	1
(C,S) := t4 + n + C	0	0	0	1	1
t3 := S	0	2	2	0	1
(C,S) := t5 + n + C	0	0	0	1	1
t4 := S	0	2	2	0	1
(C,S) := t6 + n + C	0	0	0	1	1

t5 := S	0	1	1	0	1
(C,S) := t7 + C	0	0	0	1	1
t6 := S	0	0	0	1	1
t7 := C	-	-	-	-	-
for i=1 to s-1 /*main loop */	1	1	3	0	s-1
(C,S) := t0 + a[i] * b[0]	0	0	0	1	s-1
t[0] := S	1	2	3	0	s-1
(C,S) := t1 + a[i]*b[1] + C	0	0	0	1	s -1
t1 := S	1	2	3	0	s-1
(C,S) := t2 + a[i]*b[2] + C	0	0	0	1	s-1
t2 := S	1	2	3	0	s-1
(C,S) := t3 + a[i]*b[3] + C	0	0	0	1	s-1
t3 := S	1	2	3	0	s-1
(C,S) := t4 + a[i]*b[4] + C	0	0	0	1	s-1
t4 := S	1	2	3	0	s-1
(C,S) := t5 + a[i]*b[5] + C	0	0	0	1	s-1
t5 := S	1	2	3	0	s-1
(C,S) := t6 + a[i]*b[6] + C	0	0	0	1	s-1
t6 := S	0	1	1	0	s-1
(C,S) := t7 + C	0	0	0	1	s-1
t7 := S	0	0	0	1	s-1
t8 :=C	0	0	0	0	s-1
C := 0 /* reduction	1	0	2	1	s-1
*/	0	1	2	0	s-1
m := t0* n' mod W	-	-	-	-	s-1
(C,S) := t0+ m	0	1	1	1	s-1
if (t1 != 0xffffffff)	0	0	1	1	s-1
t0 := t1 + C	1	0	2	1	s-1
t1 := t2	0	1	2	0	s-1
n= m*p[3]	0	0	0	1	s-1
(C,S) := t3 + n	-	-	-	-	s-1
t2 := S	0	1	1	0	s-1
else	0	0	0	1	s-1
(C,S) := t1 + C	0	1	1	0	s-1
t0 := S	0	0	0	1	s-1
(C,S) := t2 + C	1	0	2	1	s-1
t1 := S	0	1	2	0	s-1
n= m*p[3]	0	0	0	1	s-1
(C,S) := t3 + n	0	2	2	0	s-1
t2 := S	0	0	0	1	s-1
(C,S) := t4 + n + C	0	2	2	0	s-1
t3 := S	0	0	0	1	s-1
(C,S) := t5 + n + C	0	2	2	0	s-1
t4 := S	0	0	0	1	s-1
(C,S) := t6 + n + C	0	1	1	0	s-1
t5 := S	0	0	0	1	s-1
(C,S) := t7 + C	0	1	1	1	s-1
t6 := S	-	-	-	-	-
t7 := C + t8	1	1	3	0	1
i = s-1	0	0	0	1	1
(C,S) := t0 + a[i] * b[0]	1	2	3	0	1
t[0] := S	0	0	0	1	1
(C,S) := t1 + a[i]*b[1] + C	1	2	3	0	1
t1 := S	0	0	0	1	1
(C,S) := t2 + a[i]*b[2] + C	1	2	3	0	1
t2 := S	0	0	0	1	1
(C,S) := t3 + a[i]*b[3] + C	1	2	3	0	1
t3 := S	0	0	0	1	1
(C,S) := t4 + a[i]*b[4] + C	1	2	3	0	1

t4 := S	0	0	0	1	1
(C,S) := t5 + a[i]*b[5] + C	1	2	3	0	1
t5 := S	0	0	0	1	1
(C,S) := t6 + a[i]*b[6] + C	0	1	1	0	1
t6 := S	0	0	0	1	1
(C,S) := t7 + C	0	0	0	1	1
t7 := S	0	0	0	0	1
t8 :=C	1	0	2	1	1
C := 0 /* reduction */	0	1	2	0	1
m := t0*n' mod W	-	-	-	-	1
(C,S) := t0+ m	0	1	1	1	1
if (t1 != 0xffffffff)	0	0	1	1	1
t[0] := t1 + C	1	0	2	1	1
t[1] := t2	0	1	2	0	1
n= m*p[3]	0	0	0	1	1
(C,S) := t3 + n	-	-	-	-	1
t[2] := S	0	1	1	0	1
else	0	0	0	1	1
(C,S) := t1 + C	0	1	1	0	1
t[0] := S	0	0	0	1	1
(C,S) := t2 + C	1	0	2	1	1
t1 := S	0	1	2	0	1
n= m*p[3]	0	0	0	1	1
(C,S) := t3 + n	0	2	2	0	1
t[2] := S	0	0	0	1	1
(C,S) := t4 + n + C	0	2	2	0	1
t[3] := S	0	0	0	1	1
(C,S) := t5 + n + C	0	2	2	0	1
t[4] := S	0	0	0	1	1
(C,S) := t6 + n + C	0	1	1	0	1
t[5] := S	0	0	0	1	1
(C,S) := t7 + C	0	1	1	1	1
t[6] := S	-	-	-	-	-
t7 := C + t8	0	1	1	1	1
if t7 != 0 /* carry correction */	-	-	-	-	-
t[0] = t[0]-1;	0	1	1	1	1
if t[3] != 0xffffffff	-	-	-	-	-
t[3]= t[3] +1	-	-	-	-	-
else	0	1	1	0	s-3
for i=3 to s-1	0	0	0	1	s-3
(C,S) := t[i] + C					
t[i] = S					
	9(s+1)	24(s+1)	26s+47 or 23s+43	19s+20 or 20s+17	

Table 16: The Montgomery Multiplication operation breakdown after optimization

Since our optimization was for a special curve, the length of word is fixed to be 7 (s=7). By knowing this, we can compare actual number of operations each algorithm performs. The classical method has 63 (s2+2s) multiplication operations whereas the optimized version has 56 multiplication operations. The number of additions for classical method has lower bound of 364 ($4s^2+3s$) and upper bound of

378 (4 s^2+5s); however, in the optimized version there is only 192 addition operations. The memory read in classical method requires 336 ($6s^2$+6s) or 343 ($6s^2$+7s) operations whereas in improved version it only requires 204 (23s+43) or 229 (26s+47) operations. And finally the memory read for classical method takes 155 ($2s^2$+8s+1) operations, in improved version, however, it takes only 153 or 157 operations. The reason the memory write is almost the same as the classical method is that a new variable *n* was introduced. However in the optimized version it is difficult to reach the upper bound of operations. For example, consider the reduction case where there is an extra addition operation in *else* section. The chances are the control will enter this section is 2^{-32}, which is very remote possibility.

However it is clear that the improved version of the Montgomery multiplication has 11% less multiplication operations, 48% less addition operations and 36% less memory read operations. This savings will contribute to efficiency as we will see later.

To achieve better performance of the ECDSA the Montgomery multiplication was also written in ARM assembly language. In assembly language it is important to have less instruction size because execution time has inverse relationship to the number of instructions in the code. To reduce the number of instructions, we used the version of the Montgomery multiplication just before replacing all output array with simple variables. Then, one might think if less instruction size is important why in C implementation we replaced the output array with simple variables that resulted in larger instruction size. In assembly language once a variable is read from memory to a register, the register can be reused over and over. However, in C programming every time you need to use an array a memory reference has to be done. In fact, restricting the use of output array or any other array in C code we are making it convenient for compiler to execute in faster fashion which negates the effect of the having larger instruction size.

The actual performance of the ECDSA signature algorithm before and after optimization is given in table 17. The performance analysis was done in

Armulator, which is a collection of programs that emulate the instruction sets and architecture of ARM processor. The frequency of emulator was set up with 80MHz and over several hundred measurements was taken. Table 17.a compares the performance of classical method and improved the ECDSA implemented in C language. There is almost 26% overall speed up achieved. When the Montgomery multiplication is implemented in ARM assembly language overall 44% speed up achieved (table 17.b).

	Classical Method	Improved Method (in C)	Percent Improvement
a	189.20 ms	129.28ms	26.39

	Classical Method	Improved Method (in ARM asm)	Percent Improvement
b	189.20 ms	106.96ms	43.47

Table 17. ECDSA timings and percent improvement. (a) classical method and C version comparison, (b) comparison with classical method and assembly version.

The timing results are as expected. Since codes in assembly language, if programmed carefully, are tend to be faster than equivalent code written in high-level language and that was the case for this work.

If the use of output array is not restricted the code size will decrease dramatically however there will be some efficiency loss around 8%.

Many techniques discussed in this thesis work can easily be adapted to other curve implementations. One such example can be the restricting the use of output array. If the length of the word is known the number of times a loop will iterate will be known too. And once this is known, the loops can be unrolled and many optimizations discussed here can easily be adapted.

It should be noted that most of the references consider operations $n=m*p[3]$ and $p[0]=p[1]*p[3]$ having equivalent number of memory read and write operations. However, in this work it was shown that in terms of efficiency they are different.

5. CONCLUSION

In this thesis work, not only general software optimization techniques were applied to the ECDSA but also we took advantage of special structure of *Curve P-224* parameters to achieve speed-up. Since the Montgomery multiplication was executing 75% of the time, it was redesigned to take advantage of special structure of the curve parameters. Further speed up was achieved by implementing it in the ARM assembly language. The work done in this thesis work should be considered as a maintenance phase of a software development process. However, future works may concentrate on some design changes as well. One possible design change could be introduction of new structures that will be passed to *all* functions by reference. Since many of the function calls has more than four parameters inside them and compilers like ARM keeps up to four parameters in the registers and stores the others in the memory stack. Thus, having more than four parameters in a function call would necessitate the memory access and reduce the performance. However, considering that codes written in high-level languages are easy to maintain but difficult to speed up, 26% improvement for a software product is quite significant.

BIBLIOGRAPHY

[1] National Institute for Standards and Technology. Digital Signature Standard. FIPS publication 186-2, January 2000

[2] Musa, M. *Improved Montgomery algorithms using special primes and impact on elliptic curve digital signature.* Master's thesis, Department of Electrical and Computer Engineering Oregon State University, June 2000.

[3] Menezes, A.J., van Oorschot, P.C., Vanstone, S.A. *Handbook of applied cryptography*. Submitted. CRC Press.

[4] Koc, C. K., Acar T. and Kaliski Jr., B. S. *Analyzing and comparing Montgomery multiplication algorithms*. IEEE Micro, 16(3):26 33, June 1996.

[5] Kelley, A., Pohl, I. *C by Dissection The Essentials of C programming.* Addison-Wesley, California, 1996.

[6] Aho, A.V., Sethi, R., Ullman, J.D. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, California, 1988.

[7] Jaggar, D. *Advanced RISC Machines Architectural Reference Manual.* Prentice-Hall, Inc, New Jersey, 1996.

[8] Ghezzi, C., Jazayeri, M., Mandrioli, D. *Fundamentals of Software Engineering.* Prentice-Hall, Inc, New Jersey, 1991.

[9] Qualline, Steve. *Practical C programming.* O'Reilly & Associates, California, 1997.

[10] Gollmann, Dieter. *Computer Security.* John Wiley & Sons, New York, 1999.

APPENDIX

APPENDIX: BACKGROUND IN RECOMMENDED ELLIPTIC CURVES FOR FEDERAL GOVERNMENT USE

Federal government has recommended choices of private key length and underlying fields for elliptic curve use.

1 PARAMETER CHOICES

1.1 Choice of key lengths

The main parameters for elliptic curve cryptography are the elliptic curve E and a *base point* G on E. The base point has a large prime order r. The results of the multiplication of f *and* r for some integer f, gives the number of points on the curve. The f is called the *cofactor* and it is not divisible by r. The cofactor is generally 1, 2 or 4. Having small cofactor results in the same length of the private and public keys.

1.2 Choice of underlying fields

There are given two kinds of fields.

A prime field, the field GF(p), which contains a prime number p of elements. Its elements are the integers modulo p, and the field arithmetic is implemented in the arithmetic of integers modulo p.

A binary field, the field $GF(2^m)$, which contains 2^m elements for some m, which is called the degree of the field. The elements of $GF(2^m)$ are the bit strings of length m, and the field arithmetic is implemented in terms of operations on the bits.

1.3 Choice of curves

Two kinds of curves are given based on how their coefficients are generated:

1. *Pseudo-random* curves: The coefficients are generated from a seeded cryptographic hash. The *Curve P-224* used in this thesis is a pseudo-random curve.

2. *Special curves*: The coefficients and underlying field have been pre-selected to optimize the efficiency of the curve operations.

1.4 Choice of base points

Any point of order r can serve as the base point. The government specification provides a sample base point $G = (G_x, G_y)$ for each recommended curve . It is also possible if users may want to generate their own base points.

2. CURVE P-224

This curve is over prime fields. The prime modulus of *Curve P-224* is of a special type called generalized *Mersenne numbers* for which modular multiplication can be carried out more efficiently than in general.

For each prime p, a pseudo-random curve

$$E:\ y^2 \equiv x^3 - 3x + b\ (mod\ p)$$

of prime order r is listed. Note that the coefficient of x i.e., a is chosen to be -3 efficiency reasons. The cofactor f is 1. The following parameters are given:

- p: The prime modulus (in decimal form)
- r: The order r (in decimal form)
- s; The 160-bit input seed to SHA-1 based algorithm
- c: The output of the SHA-1 based algorithm
- b: The coefficient (satisfying $b^2 c \equiv -27\ (mod\ p)$)
- G_x: The base point x coordinate
- G_y: The base point y coordinate

The bit strings and field elements are given in hexadecimal notation.

p = 26959946667150639794667015087019630673557916260026308143510066298881
r = 26959946667150639794667015087019625940457807714424391721682722368061
s = bd713447 99d5c7fc dc45b59f a3b9ab8f 6a948bc5
c = 5b056c7e 11dd68f4 0469ee7f 3c7a7d74 f7d12111 6506d031 218291fb
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