The Elliptic Curve Digital Signature Algorithm (ECDSA) is one of the most popular algorithms to digitally sign streams or blocks of data. In this thesis we concentrate on porting and optimizing the ECDSA on the ARM7 processor for a particular NIST curve over GF(2^m). The selected curve is a binary curve of order 233. We show that for this particular curve, the ECDSA can be implemented significantly faster than the general case. The optimized algorithms have been implemented in C and the ARM assembly. The analysis and performance results indicate that by using certain machine and curve specific techniques, the ECDSA signature can be made up to 41% faster.
ECDSA Optimizations on ARM Processor for a NIST Curve Over \( \text{GF}(2^m) \)

by

Eda Turan

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 15, 2001
Commencement June 2002

APPROVED:

Redacted for Privacy

Major Professor, representing Electrical and Computer Engineering

Redacted for Privacy

Chair of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of the 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 other reader upon request.

Redacted for Privacy

Eda Turan, Author
First, I'd like to thank my family for their endless support and for encouraging me along the way during my studies.

I give special thanks to Dr. Koç, my major professor, for recognizing my potential, for his sponsorship, and for his reviews and help in producing this thesis.

I also thank Dr. Tenca, Dr Fein and Dr. Eldin 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: Murat Aydos, Serdar Erdem, Francisco Rodríguez-Henríquez, Albert Levi, Erkay Savaş, Haluk Kent Tanık and Tuğrul Yanık.

Finally, I acknowledge financial support for this work from rTrust.

Eda Turan
Corvallis, Oregon
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2. BACKGROUND</td>
<td>3</td>
</tr>
<tr>
<td>2.1. INFORMATION SECURITY</td>
<td>3</td>
</tr>
<tr>
<td>2.2. CRYPTO_SYSTEMS</td>
<td>4</td>
</tr>
<tr>
<td>2.2.1. Secret Key Cryptosystems</td>
<td>4</td>
</tr>
<tr>
<td>2.2.2. Public Key Cryptosystems</td>
<td>5</td>
</tr>
<tr>
<td>2.3. HARD MATHEMATICAL PROBLEMS</td>
<td>7</td>
</tr>
<tr>
<td>2.3.1. Integer Factorization</td>
<td>7</td>
</tr>
<tr>
<td>2.3.2. Discrete Logarithm</td>
<td>8</td>
</tr>
<tr>
<td>2.3.3. Elliptic Curve Discrete Logarithm</td>
<td>8</td>
</tr>
<tr>
<td>2.4. ELLIPTIC CURVE CRYPTOGRAPHY</td>
<td>9</td>
</tr>
<tr>
<td>2.4.1. Mathematical Background</td>
<td>9</td>
</tr>
<tr>
<td>2.4.2. Elliptic Curves Over Finite Fields</td>
<td>11</td>
</tr>
<tr>
<td>2.4.3. Elliptic Curve Digital Signature Algorithm (ECDSA)</td>
<td>13</td>
</tr>
<tr>
<td>2.5. ARM OVERVIEW</td>
<td>15</td>
</tr>
<tr>
<td>2.5.1. Architecture</td>
<td>15</td>
</tr>
<tr>
<td>2.5.2. ARM Instruction Set</td>
<td>16</td>
</tr>
<tr>
<td>2.5.3. Using Inline assemblers</td>
<td>18</td>
</tr>
<tr>
<td>3. ANALYSIS AND IMPROVEMENTS</td>
<td>21</td>
</tr>
<tr>
<td>3.1. IMPLEMENTATION</td>
<td>21</td>
</tr>
<tr>
<td>3.1.1. Use of Elliptic Curves</td>
<td>21</td>
</tr>
<tr>
<td>3.1.2. Point Arithmetic</td>
<td>22</td>
</tr>
<tr>
<td>3.2. IMPROVEMENTS</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1. General optimization techniques</td>
<td>28</td>
</tr>
<tr>
<td>3.2.2. Field Multiplication</td>
<td>29</td>
</tr>
<tr>
<td>3.3. ECDSA TIMINGS AFTER OPTIMIZATIONS</td>
<td>44</td>
</tr>
<tr>
<td>4. CONCLUSION</td>
<td>46</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>48</td>
</tr>
<tr>
<td>APPENDIX: RECOMMENDED ELLIPTIC CURVES FOR FEDERAL GOVERNMENT USE</td>
<td>51</td>
</tr>
<tr>
<td>(July 1999)</td>
<td></td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure | Page
--- | ---
3.1. Karatsuba multiplication algorithm | 31
3.2. 8-word recursive karatsuba multiplication | 32
3.3. 8 and 4-word karatsuba multiplication | 33
3.4. 8-word iterative karatsuba multiplication | 33
LIST OF TABLES

Table                                      Page
2.1. Condition Fields in ARM assembly language      18
3.1. Karatsuba multiplication timings              32
3.2. Optimized Karatsuba multiplication            34
3.3. 4-word Karatsuba in assembly                  36
3.4. Lookup table in 1-word multiplication          38
3.5. 1-word multiplication in assembly             42
3.6. 1-word multiplication optimization results    42
3.7. Reduction in assembly                         44
3.8. Reduction optimization results                44
3.9. ECDSA optimization results on ARM7             45
3.10. ECDSA optimization results on ARM9            45
To my family…

for their endless love and support
1. INTRODUCTION

Information has been one of the most important and valuable resources of governments and businesses. Before the electronic era, information was printed on paper. It was stored and transmitted physically. To protect the information from possible attacks, physical precautions were sufficient. With revolution in electronics, it becomes easier and popular to keep information in electronic form. Despite of the easiness in storing and transmitting electronically, it is more difficult to protect electronic information against security threats than printed information. It is much easier to reach, stole or intercept and modify the information stored or transmitted electronically. As threats on information security become more damaging, researches on security gain much more importance.

Cryptosystems serve as a tool to protect users from these various security attacks. There are two kinds of systems in use, which are symmetric and public key cryptosystems. Based on these, cryptographic routines such as encryption, decryption, authentication, signature, are implemented. The security of such routines depends on some mathematical problems, which are known to be hard to solve such as integer factorization.

In 1985, Neal Koblitz and Victor Miller independently proposed a new idea of using elliptic curves on public key systems, which is known as elliptic curve cryptography (ECC). The security of ECC is based on a hard mathematical problem, as the other cryptosystems. The problem is known as elliptic curve discrete logarithm problem and it is much harder than the analogous ones defined.
Because of the difficulty of the hard problem, among any known public key systems ECC delivers the highest strength per bit. This means equivalent levels of security as the other cryptosystems can be achieved with smaller fields and shorter parameter lengths.

The performance of elliptic curve cryptosystems depends on efficient implementation of cryptographic algorithms, such as encryption, signature etc. As these routines are built on elliptic curves, the efficiency of them depends on the arithmetic operations on elliptic curves over finite fields such as point addition. And the building blocks of these arithmetic operations are field operations over finite fields such as addition, multiplication etc. That’s why, performance of field operations affects performance of elliptic curves. This fact leads us to work on the implementation of field operations of such cryptosystems.

In this thesis we concentrate on improving the performance of elliptic curve digital signature algorithm (ECDSA) on ARM. We choose to work on a particular binary curve (B-233) recommended by NIST for U.S. Federal Government. Our work is mainly focused on techniques to optimize the signature algorithm for this particular curve. For time efficiency, the most time consuming parts of the algorithm are improved with general and machine specific optimization techniques.

Chapter 2 section 2.1 and section 2.2 provide general information about security services, and cryptosystems such as symmetric and public key systems. In section 2.3, hard mathematical problems are explored. Elliptic curve cryptosystems are presented in detail in section 2.4. Also, ECDSA is analyzed in detail. Section 2.5 mainly explores ARM processor architecture and instruction set.

Chapter 3 section 3.1 explores the implementation details of the ECDSA code. In section 3.2, applied optimization techniques and ARM assembly language implementation details are presented. And section 3.3 summarizes the overall results.

Chapter 4 is the conclusion for the thesis.

Appendix provides information about recommended curves by NIST.
2. BACKGROUND

In this chapter we have discussed the necessary background information for our thesis. These include security services, cryptosystems, hard mathematical problems, elliptic curve cryptography and ARM processor architecture.

2.1. INFORMATION SECURITY

Information is known to be an asset for governments and businesses. Information security includes the measures taken to prevent unauthorized access to electronic data. There are three fundamental services of information security. These are:

- **Confidentiality**: concealment of data from unauthorized parties.
- **Integrity**: assurance that data is genuine.
- **Availability**: the system still functions efficiently after security provisions are in place.

Cryptographic systems offer these services to keep information secure. To clarify the application of cryptographic systems, confidentiality and integrity are sub-classified into the following services. These services can be considered as building blocks of security.

**Confidentiality**: The information in a computer system should be accessible only for reading by authorized parties. Confidentiality ensures this access control even if the information is transmitted.

**Authentication**: Authentication ensures the receiver of a message to be able to identify the sender of the message.

**Integrity**: The information in a computer system and the transmitted information can only be modified by the authorized parties. Integrity ensures to keep the information unmodified if the access is not authorized. In data transmission, messages should be received as sent.
Nonrepudiation: Nonrepudiation assures that neither the sender nor the receiver of a message can deny the transmission they involved.

Access Control: This requires the control of access to information resources by the target system if possible.

Availability: Availability ensures that the information is available for access by authorized parties when needed.

2.2. CRYPTO SYSTEMS

To store or transmit data securely, it is transformed into an illegible form using encryption. An electronic key, which is simply a binary string, controls the conversion of original data into its encrypted form. A cryptosystem is a method of encrypting messages that can be decrypted only if the decryption key and algorithm is known. Cryptography can be considered as the art of creating and applying cryptosystems in real life. Cryptanalysis is the art of breaking cryptosystems. Cryptology is the study of both cryptography and cryptanalysis.

There are two types of cryptosystems that are commonly used. These are:

- Secret Key Cryptosystems
- Public Key Cryptosystems

2.2.1. Secret Key Cryptosystems

Secret key algorithms rely on one shared key for encryption and decryption. The original message is called plaintext. It is encrypted into an unreadable text, which is called ciphertext. Plaintext is encrypted to produce ciphertext by using an encryption algorithm with a secret key, which is only known by the sender and receiver. The key is created independent of the plaintext. Once cipher text is produced, it is transmitted to the receiver. The receiver can transform cipher text to
its original plaintext using the corresponding decryption algorithm and the same key.

These systems encrypt large amounts of data efficiently, but they encounter problems in key management issues. As the protocol states, each individual should have a distinct key to communicate with each other person in the network. That’s why, in large networks, key distribution and management in symmetric cryptosystems becomes a problem.

2.2.2. **Public Key Cryptosystems**

Public key cryptosystems are introduced in 1976 by Whitfield Diffie and Martin Hellman. Public key cryptography techniques proposed as a method for the most difficult problems of secret key cryptography. These are the key distribution and digital signatures.

Public key algorithms use two different but related keys for encryption and decryption. The encryption key (public key) and the cryptographic algorithm are known. The decryption key (private key) is private, only known by its owner. These two keys are related by a hard one-way function. By knowing the public key, private key cannot be computed. The key management problems of secret key encryption are solved with public key systems. Also, with public key systems digital signatures can be efficiently implemented.

Public key cryptosystems are capable of fulfilling the main objectives of information security. They can be used efficiently for confidentiality purposes. In terms of an arbitrary group, encryption process can be described as follows:

1. Each end system in a network generates its public and private key for encryption and decryption purposes.
2. Each system publishes its public key by keeping it in a public server or file, such that, all systems have access to the public keys.
3. When A wants to send an encrypted message to B, A encrypts the message using B’s public key and the encryption algorithm.
4. When B receives the encrypted message, B decrypts the message using its own private key and corresponding decryption algorithm. Even if the message is hacked during transmission, no other system can decrypt the message since only B knows its private key.

Public key cryptosystems can also be used for authentication purposes. Suppose B wants to be sure the encrypted message is from A. The steps of authentication process are as follows:

1. A encrypts a message by using its own private key, which generates the digital signature for that message, and sends it to B.
2. When B receives the message, it can decrypt it using A’s public key.
3. Since the message is encrypted by using A’s private key, it can only be read by using A’s public key. This ensures that the message is originally from A.

Digital signatures can be implemented using public key systems. Digital signatures can be considered as the electronic equivalent of normal signatures. The process of creating digital signatures is more complex than written signatures. It should be secure because it is easier to duplicate electronic information than paper-based information. The use of public key cryptosystems differs in encrypting and signing process. The essential difference between them is that the order in which the keys are used is reversed. The signature process is summarized as follows:

1. When A wants to sign a message, it first transforms the message using a hash function.
2. The output value of the hash function is specific to the content of the message. This output value, h(M) is called a message digest and can be considered as the fingerprint of the message.
3. Then, A applies its private key to the message digest using a digital signature algorithm to produce a digital signature.
4. A sends the message to B along with its signature.
5. Once B receives the message and the signature, it transforms the signature using A’s public key, to get h(m). Then, B computes the message digest by using the hash function and the message itself. Finally, B compares these
two values of h(m) to see if they are equal or not. If this comparison succeeds, then B accepts, A’s signature is valid.

2.3. HARD MATHEMATICAL PROBLEMS

The security of public key cryptosystems depends on some difficult mathematical problems. If the fastest algorithm to solve a mathematical problem with the current resources takes long time (exponential time) relative to the input size, then the problem is considered to be difficult. For all mathematical problems, with very small input sizes, it is straightforward to solve them even if the problems are meant to be difficult. As the input size grows, it should get harder to solve the problem. When designing public key cryptosystems based on a mathematical problem, it is a critical in terms of security to find a problem, which takes exponential time to solve with the fastest algorithm.

In years, many of the proposed public key cryptosystems have been broken. Today there are three known hard mathematical problems, which form basis to cryptosystems. These problems are integer factorization, discrete logarithm, and elliptic curve discrete logarithm problem. RSA, US government’s DSA and ECC are examples of the cryptosystems based on the computational complexity of these mathematical problems, respectively. We will briefly describe these problems.

2.3.1. Integer Factorization

Let n be an integer such that pq = n, where p and q are prime numbers. Integer Factorization problem is defined as finding p and q where n is known. RSA is one of the examples to the public key cryptosystems whose security relies on integer factorization problem. In RSA, with today’s computing technology, n should be at least 1024 bits to provide its security.
2.3.2. **Discrete Logarithm**

Let \( p \) be a prime number. We have the following relation between \( g \) and \( e \) where \( g \) and \( e \) are integers in the interval \([0, p-1]\).

\[ e = g^d \pmod{p}, \text{ for some } d. \]

The discrete logarithm problem states that in modulo \( p \), there is no efficient algorithm to find \( d \), given \( g \) and \( e \) for large input size. Taher ElGamal was the first one who proposed to design public key cryptosystems based on discrete logarithm problem. One of the examples of these systems is known as DSA. In DSA, the prime \( p \) should also be chosen large enough for a secure scheme.

2.3.3. **Elliptic Curve Discrete Logarithm**

An elliptic curve over a finite field is the set of solutions \((x, y)\) to an equation of the form \( y^2 = x^3 + ax + b \) over \( \text{GF}(p) \) or \( y^2 + xy = x^3 + ax^2 + b \) over \( \text{GF}(2^n) \) for some \( a \) and \( b \). The elliptic curves will be analyzed in detail.

Let \( P \) be a point on elliptic curve, and \( p \) be a prime number. For some integer \( k \);

\[ Q = kP, \text{ where } Q \text{ is a point on the curve}. \]

Elliptic curve discrete logarithm problem is defined to determine \( k \), where \( Q \) and \( P \) are given. The security of elliptic curve cryptography relies on this problem.

The elliptic curve discrete logarithm is known to be harder than both integer factorization and discrete logarithm problem modulo \( p \). In ECC, for moderate security, the prime \( p \) is chosen to be at least 160 bits.

Compared to RSA and DSA, the modulus size is significantly small, which leads to a significant improvement in efficiency. This can also be considered as an advantage for some applications where memory or computational power resources are limited.
2.4. ELLIPTIC CURVE CRYPTOGRAPHY

Elliptic curve systems have been studied for the past 150 years. In 1985, Neal Koblitz and Victor Miller independently proposed first cryptographic applications of elliptic curve cryptography (ECC). The security of ECC is based on the computational complexity of the elliptic curve discrete logarithm problem.

2.4.1. Mathematical Background

In order to understand elliptic curve concepts, we need to know the mathematical structure it is based on.

A group, \( G \) is a set together with an operation \( \varphi \), which has the following properties:

- \( G \) is closed. That is, if \( x, y \in G \), then \( x \varphi y \in G \)
- \( G \) is associative. That is, if \( x, y \) and \( z \in G \), then \( (x \varphi y) \varphi z = x \varphi (y \varphi z) \)
- There exists an identity element \( e \in G \), for each \( x \in G \), such that \( x \varphi e = e \varphi x \).
- For each element \( x \in G \), there is a \( y \in G \), such that \( x \varphi y = y \varphi x = e \)

The group \( G \) is called an abelian or commutative group, if the operation \( \varphi \) is commutative. That is, \( x, y \in G \), then \( x \varphi y = y \varphi x \).

A finite field contains finite set of elements, \( F \), with two binary operations on \( F \), + and *, addition and multiplication, respectively. \( F \) satisfies the following:

- \( F \) is closed under both + and *
- \( F \) forms a commutative group under the operator +.
- \( F\) - {0} forms a commutative group under the operator *.
- \( F \) is distributive for * operation. The operator * distributes over the operator +, such that \( a * (b + c) = a * b + a * c \).

The order of the field is defined as the number of elements in the finite field. There exist a finite field of order \( q \), if and only if \( q \) is a prime power. If \( q \) is a prime number, then
prime power, then there is only one finite field with the order \( q \), and that is denoted as \( F_q \).

The two common choices for underlying finite field are:

- \( F_p \), the order of the finite field is an odd prime (\( q = p \)).
- \( F_{2^m} \), the order of the finite field is a power of 2 (\( q = 2^m \)).

There are also studies on finite fields with the order as a power of a prime (\( q = p^n \)). In this thesis, we will focus on elliptic curves over \( F_{2^m} \).

Over finite field \( F_{2^m} \), arithmetic rules can be described by either polynomial or normal basis representation. Detailed explanation is in Appendix. We will be using polynomial representation.

Let \( f(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_1x + f_0 \) where \( f_i \in \{0, 1\} \) for \( i = 0, 1, \ldots, m-1 \). \( f(x) \) is an irreducible polynomial of degree \( m \) over \( F_2 \), that is \( f(x) \) cannot be factored as a product of two polynomials of less than degree \( m \) over \( F_2 \). Each such irreducible polynomial \( f(x) \) defines a polynomial representation of \( F_{2^m} \).

Polynomials of degree less than \( m \) over \( F_2 \) comprise the finite field \( F_{2^m} \).

\( F_{2^m} = \{(a_{m-1} \ldots a_1a_0) : a_i \in \{0, 1\}\} \), where \( (a_{m-1} \ldots a_1a_0) \) is the representation of the polynomial \( a_{m-1}x^{m-1} + \ldots + a_1x + a_0 \). The following arithmetic operations are defined with polynomial basis representation on \( F_{2^m} \).

Let \( a, b \in F_{2^m} \), \( a = (a_{m-1} \ldots a_1a_0) \) and \( b = (b_{m-1} \ldots b_1b_0) \) and \( f(x) \) is the irreducible polynomial. The operations are:

- **Addition**
  \[ a + b = c \text{ where } c = (c_{m-1} \ldots c_1c_0) \text{ and } c_i = (a_i + b_i) \mod 2, c \in F_{2^m} \]

- **Multiplication**
  \[ a \cdot b = c, c = (r_{m-1} \ldots r_1r_0) \]
  The polynomial \( (r_{m-1}x^{m-1} + \ldots + r_1x + r_0) \) is defined to be the remainder polynomial, when \( (a_{m-1}x^{m-1} + \ldots + a_1x + a_0) \cdot (b_{m-1}x^{m-1} + \ldots + b_1x + b_0) \) is divided by \( f(x) \) over \( F_2 \).

- **Inversion**
  If \( a \neq 0 \), the inverse of \( a \), \( t = a^{-1} \), where \( t \in F_{2^m} \), is the unique element for which \( a \cdot t = 1 \).
2.4.2. **Elliptic Curves Over Finite Fields**

There are two types of elliptic curve groups defined over two types of finite field. These are elliptic curves over $\text{GF}(p)$ and $\text{GF}(2^m)$.

### 2.4.2.1. Elliptic Curves over $\text{GF}(p)$

Let $\text{GF}(p)$ be a finite field of characteristic two. An elliptic curve $E$ over $\text{GF}(p)$ is defined by the following equation, where $a, b \in \text{GF}(p)$, and $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$. $p$ is a prime, where $p > 3$.

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

The elliptic curve group over $\text{GF}(p)$, $E$, consists of all the points on the curve and the point at infinity, which is denoted as $O$.

### 2.4.2.2. Elliptic Curves over $\text{GF}(2^m)$

An elliptic curve $E$ over $\text{GF}(2^m)$ is defined by the equation of the following form, where $a, b \in \text{GF}(2^m)$, $b \neq 0$

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

The points $(x, y)$ on the curve satisfy this equation over $\text{GF}(2^m)$, where $x, y \in \text{GF}(2^m)$. The elements of an elliptic curve group over $\text{GF}(2^m)$, $E$, are all the points of that elliptic curve and the point at infinity, $O$.

### 2.4.2.3. Arithmetic Operations on Elliptic Curves over $\text{GF}(2^m)$

In affine coordinates, the elliptic group operations are given as follows.

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be elements of an elliptic curve group $E$ over $\text{GF}(2^m)$.

- For all $P \in E$, $P + O = O + P = P$. 
• $P = (x_1, y_1)$, then $(x_1, y_1) + (x_1, x_1+y_1) = O$. Here, the point $(x_1, x_1+y_1)$ is the negative of $P$ and denoted as $-P$. $-P$ is also a point on the curve.

• Point Addition
Let $P$ and $Q$ be distinct points in $E$, and $Q 
eq -P$. Then $P + Q = R = (x_3, y_3)$, where

\[
\lambda = \frac{y_1 - y_2}{x_1 + x_2}
\]

\[
x_3 = \lambda^2 + \lambda x_1 + x_2 + a
\]

\[
y_3 = \lambda (x_1 + x_3) + x_3 + y_1
\]

• Point Doubling
$P = (x_1, y_1) \in E$
If $x_1 = 0$, then $2P = 0$.
Provided that $x_1 \neq 0$, $2P = R = (x_3, y_3)$, where

\[
x_3 = x_1^2 + \frac{b}{x_1}
\]

\[
y_3 = x_1^2 + (x_1 + \frac{y_1}{x_1}) x_3 + x_3
\]

Here is an example for elliptic curve point addition:
Consider the irreducible polynomial $f(x) = x^4 + x + 1$ represents the finite field $F_{2^4}$.
Consider the elliptic curve $E : y^2 + xy = x^3 + \alpha^4 x^2 + 1$ over $F_{2^4}$. So, $a = \alpha^4$ and $b = 1$.

1. Let $P = (\alpha^6, \alpha^8)$ and $Q = (\alpha^3, \alpha^{13})$. $P+Q = (x_3, y_3)$ is computed as:

\[
x_3 = \left(\frac{\alpha^8 + \alpha^{13}}{\alpha^6 + \alpha^3}\right)^2 + \frac{\alpha^8 + \alpha^{13}}{\alpha^6 + \alpha^3} + \alpha^6 + \alpha^3 + \alpha^4 = \left(\frac{\alpha^3}{\alpha^2}\right)^2 + \frac{\alpha^3}{\alpha^2} + \alpha^6 + \alpha^3 + \alpha^4 = 1
\]

\[
y_3 = \left(\frac{\alpha^8 + \alpha^{13}}{\alpha^6 + \alpha^3}\right)(\alpha^6 + 1) + 1 + \alpha^8 = \left(\frac{\alpha^3}{\alpha^2}\right)(\alpha^{13}) + \alpha^2 = \alpha^{13}
\]

$P+Q = (1, \alpha^{13})$

2. Let $P = (\alpha^6, \alpha^8)$. $P+P = 2P = (x_3, y_3)$ is computed as:
2.4.3. Elliptic Curve Digital Signature Algorithm (ECDSA)

ECDSA is an algorithm to create digital signatures using elliptic curves. It is known to be the elliptic curve analogue of DSA (Digital Signature Algorithm). It was first proposed in 1992 by Scott Vanstone. It was accepted in 1998 as an ISO (International Standards Institute) standard (ISO 14888-3), in 1999 as an ANSI (American National Standard Institute) standard (ANSI X9.62), and in 2000 as an IEEE (Institute of Electrical and Electronics Engineers) standard (IEEE 1363-2000) and a FIPS (Federal Information Processing Standard) standard (FIPS 186-2). We will briefly discuss the algorithm in this subsection.

Domain parameters for ECDSA consist of an elliptic curve $E$ defined over a finite field $F_q$ of characteristic $q$ and a base point $P \in E(F_q)$. They are summarized as follows:

1. A field size $q$, where either an odd prime $q = p$, or $q = 2^m$.
2. Representation used for the elements of $F_q$.
3. Two field elements $a$ and $b$ in $F_q$ which define the equation of the elliptic curve $E$ over $F_q$ (that is, $y^2 = x^3 + ax + b$ for $q = p$ or $y^2 + xy = x^3 + ax^2 + b$ for $q = 2^m$).
4. Two field elements $x_g$ and $y_g$ in $F_q$ which define a base point $P(x_g, y_g)$ of prime order in $E(F_q)$.
5. The order $n$ of the point $P$, with $n > 2^{160}$ and $n > 4\sqrt{q}$.
6. The cofactor $h = \#E(F_q)/n$. 

\[
\begin{align*}
x_3 &= (\alpha^6)^2 + \frac{1}{(\alpha^6)^2} = \alpha^{12} + \alpha^3 = \alpha^{10} \\
y_3 &= (\alpha^6)^2 + \left(\frac{\alpha^6}{\alpha^8}\right)\alpha^{10} + \alpha^{10} = \alpha^{12} + \alpha^{13} + \alpha^{10} = \alpha^8 \\
2P &= (\alpha^{10}, \alpha^8)
\end{align*}
\]
ECDSA can be examined in three main parts. These are key generation, signature generation, and signature verification.

2.4.3.1. ECDSA key generation

The steps for the key generation are as follows:
1. An elliptic curve $E$ over $\mathbb{Z}_p$ is selected. Number of points in $E(\mathbb{Z}_p)$ should be a power of a large prime $n$.
2. A point $P \in E(\mathbb{Z}_p)$ of order $n$ is selected as base point.
3. A unique integer $1 \leq d \leq n-1$ is selected as private key.
4. $Q = dP$ is computed.
5. This user’s public key is $(E, P, n, Q)$ and private key is $d$.

2.4.3.2. ECDSA signature generation

To sign a message:
1. An integer $k$ in the interval $[1, n-1]$ is selected.
2. $kP = (x_1, y_1)$ is computed and $r = x_1 \mod n$
   If $r = 0$, then go to step 1.
3. $k^{-1} \mod n$ is computed.
4. $s = k^{-1} \{h(m) + dr\}$ is computed. Here $h$ is the Secure Hash Algorithm (SHA-1).
5. If $s=0$, then go to step 1.
6. The signature for the message $m$ is $(r, s)$.

2.4.3.3. ECDSA signature verification

To verify a message:
1. A copy of A’s public key $(E, P, n, Q)$ is needed. $r$ and $s$ are checked to be integers in the interval $[1, n-1]$.
2. $w = s^{-1} \mod n$ and $h(m)$ are computed.
3. $u_1 = h(m)w \mod n$ and $u_2 = rw \mod n$ are computed.
4. \( u_1P + u_2Q = (x_0, y_0) \) and \( v = x_0 \mod n \) are computed.

5. Check to see if \( v = r \) or not. If they are equal, then the signature is accepted.

### 2.5. ARM OVERVIEW

One of the most important motivations of this thesis was to optimize signature generation for chosen particular curve on an ARM9 platform. We have followed two ways for this purpose. The language of the software was originally C. One way is to write efficient C code for ARM platform. The other is to optimize some functions using ARM assembly. Both techniques are used for efficiency. In order to work on an ARM platform, ARM architecture and instruction set are examined in detail.

#### 2.5.1. Architecture

ARM is a RISC (Reduced Instruction Set Computer) and incorporates all the typical features of RISC architecture. It has a large uniform register file and load-store architecture, which means data operations are only applied on register contents. ARM has simple addressing modes and uniform and fixed length instruction fields. Besides these, ARM has some additional features. In every data-processing instruction, it has control over both ALU and shifter to maximize the use of a shifter and an ALU. In addressing modes, it has auto-increment and auto-decrement features to optimize program loops. In memory access instructions, it has multiple load and store instructions, which maximize data throughput. And finally, it has conditional execution of all instructions to maximize execution throughput. With these additional features to a typical RISC architecture, it provides efficient implementations with low power consumption, and minimal die size.

ARM has thirty-seven 32-bit registers. These can be classified as follows:
• 30 of them are general-purpose registers. 15 of them are visible at a time, r0, r1, ... r14. Register 14 is assigned as Linked Register (LR) and used to hold the address of the next instruction (return address) after a subroutine call is made. It can be used as a general-purpose register at other times. Register 13 is used as a stack pointer (SP), which points to the next location in stack. Register 12 is used during function calls. All other registers are for general usage.

• Register 15 (r15) is used as Program Counter (PC). It is used as a pointer to the next instruction. Since all ARM instructions are 4-byte long, PC is incremented by one word from each instruction in ARM state. PC, itself contains 30 bits, because the instructions are word-aligned.

• The Current Program Status Register (CPSR) holds flags such as copies of ALU status flags, the current processor mode and interrupt disable flags.

• Five Saved Program Status Registers (SPSRs) are used to store CPSR when an exception occurs.

2.5.2. **ARM Instruction Set**

All ARM instructions are 4-byte (32-bit) long and are stored word-aligned in memory. By checking ALU flags in CPSR, all ARM instructions can be executed conditionally. All instructions access the fifteen general-purpose registers (r0–r14) and some also allow access to program counter (r15). The ARM ALU has a 32-bit barrel shifter, which is powerful for very general shift and rotate operations. The second operand to all data processing and single register load and store instructions can be shifted before the operation, as a part of the same instruction.

ARM instructions can be classified into the following groups:

• **Data Processing Instructions**

  Addition, subtraction or bit-wise logic operations are some of the operations performed with these instructions. They operate on two general-purpose registers
and place the result in a third register. Instructions for long multiplication give a 64-bit result placed in two registers.

- **Branch Instructions**

These instructions are used to branch forward, backward or to subroutines.

- **Memory Access Instructions (Single register and multiple register)**

These are the load and store instructions to get or place any subset of the general-purpose registers from or to memory.

- **Status register access Instructions**

These instructions are used to move the contents in between CPSR or an SPSR and a general-purpose register.

- **Semaphore Instructions**

These instructions are used to load and alter a memory semaphore.

- **Coprocessor Instructions**

These instructions are used to support a general way to extend ARM architecture.

All ARM instructions can be conditionally executed. Data processing instructions can update the four condition code flags in the CPSR (Negative (N), Zero (Z), Carry (C) and oVerflow (V)) according to their result. Subsequent instructions can be conditionally executed according to these flags. There are fifteen conditions implemented depending on the status of the condition code flags. These are shown in Table 2.1.
Table 2.1. Condition Fields in ARM assembly language

<table>
<thead>
<tr>
<th>Condition Field {cond}</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>Not Equal</td>
</tr>
<tr>
<td>CS</td>
<td>Unsigned higher or same</td>
</tr>
<tr>
<td>CC</td>
<td>Unsigned lower</td>
</tr>
<tr>
<td>MI</td>
<td>Negative</td>
</tr>
<tr>
<td>PL</td>
<td>Positive or zero</td>
</tr>
<tr>
<td>VS</td>
<td>Overflow</td>
</tr>
<tr>
<td>VC</td>
<td>No Overflow</td>
</tr>
<tr>
<td>HI</td>
<td>Unsigned higher</td>
</tr>
<tr>
<td>LS</td>
<td>Unsigned lower or same</td>
</tr>
<tr>
<td>GE</td>
<td>Greater or equal</td>
</tr>
<tr>
<td>LT</td>
<td>Less than</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than</td>
</tr>
<tr>
<td>LE</td>
<td>Less than or equal</td>
</tr>
<tr>
<td>AL</td>
<td>Always</td>
</tr>
</tbody>
</table>

2.5.3. Using Inline assemblers

The inline assemblers enable one to use most ARM assembly language instructions within a C or C++ program. With use of inline assembler, more efficient code can be achieved. Also, it allows to access parts of the processor that cannot be accessed from C. The inline assembler also expands the complex instructions and optimizes the assembly language code written by the user.

The inline assembly language is supported with _asm specifier by the ARM C compiler inside a C function. Here is an example of an inline assembly language block in a C code:

```c
function Example{
    [C-code]
}
The inline assembler is a high-level assembler. It does not support some of the low-level features that are available to armasm, such as branching by writing to pc. For example,

In armasm:

```
AREA subrout, CODE, READONLY
ENTRY
start  MOV  r0, #10
       MOV  r1, #3
       BL   label
label  ADD  r0, r0, r1
       MOV  pc, lr
END
```

In inline assembler, the last MOV instruction, which writes to pc as a return from branching, cannot be performed. Instead of this instruction, the user should branch back to the return location at the end of the subroutine. This restricts use of subroutine calls because the user should know where to return, which is mostly not the case.

The other issue with inline assembler is function calls. Function calls that use BL instruction from inline assembly must specify the input, output and corrupted registers, different from armasm. Here is an example of inline assembly language block in C code:

```c
void my_strcpy(char *src, char *dst) {
    int ch;
    __asm{
```
main(void) {
    char a[] = "Hello World";
    char b[20];
    __asm {
        MOV r0, a
        MOV r1, b
        BL my_strcopy, {r0, r1}, {}, {}  
    } 
    return 0; }
3. ANALYSIS AND IMPROVEMENTS

In this chapter, implementation details of ECDSA code is analyzed and optimizations and results are discussed.

3.1. IMPLEMENTATION

We aim to optimize signature generation software for the particular curve we have chosen. In order to optimize, the code, and especially the parts that consume most of the resources should be well understood. First of all, we will examine these parts in detail.

3.1.1. Use of Elliptic Curves

Among recommended curves by NIST for US federal government use, we have chosen to work on an elliptic curve over GF(2^m). Elliptic curves over GF(2^m) can have efficient hardware and software implementations, compared to the curves over GF(p). With the use of particular curves, efficiency can be improved in various levels.

The particular curve chosen over GF(2^m) has the following parameters:
Curve B-233 with polynomial basis representation:

\[ m = 233 \text{ bits} \]
\[ p(t) = t^{233} + t^{74} + 1, \text{ where } t = 2. \]

Base point order \( r \):
\[ r = 6901746346790563787434755862277025555839812737345013555379383634485463 \]

Parameters:
\[ a = 1 \]
b = 066 647ede6c 332c7f8c 0923bb58 213b333b 20e9ce42 81fe115f 7d8f90ad
Base point x and y coordinates:
G_x = 0fa c9dfcbbac 8313bb21 39f1bb75 5f6f65bc 391f8b36 f8f8eb73 71fd558b
G_y = 100 6a08a419 03350678 e58528be bf8a0bef f867a7ca 36716f7e 01f81052

The detailed information about NIST curves and their parameters is in Appendix.

We have worked on signature generation part of a word-based implementation of ECDSA. Both field arithmetic and normal arithmetic operations on long integers are performed one word at a time rather than bit by bit. Size of a word is determined as 32 bits. Since ARM instructions operate on 32-bit registers, it is efficient to handle 32 bits at a time. In our elliptic curve, the key and parameter size is 233-bit. 233 / 32 = 7.28, which is 7 words and 9 bits. We have 8 words, with only 9 bits used in the last word.

In the next subsections, we will talk about the algorithms previously implemented.

3.1.2. Point Arithmetic

In Elliptic Curve Digital Signature Algorithm, the most time-consuming arithmetic operation is scalar multiplication, where a point P on the curve is multiplied by a scalar integer k, Q = kP. Various approaches have been proposed to implement point multiplication. One of these algorithms is Algorithm 2P in projective coordinates, which has been introduced by J. López and R. Dahab.

Algorithm 2P:
Input: an integer k ≥ 0 and a point P = (x, y) ∈ E
Output: Q = kP
If k = 0 or x = 0 then output (0, 0) and stop.
1. Set k ← (k_{i-1} ... k_1 k_0)_2
2. Set X_1 ← -x, Z_1 ← 1, X_2 ← x^4 + b, Z_2 ← x^2
3. for I from 1 to 2 downto 0 do
   if k = 1 then
      Madd(X₁, Z₁, X₂, Z₂), Mdouble(X₂, Z₂)
   else
      Madd(X₂, Z₂, X₁, Z₁), Mdouble(X₁, Z₁)
   return (Q = Mxy(X₁, Z₁, X₂, Z₂))

In this algorithm, the number of field operations is as follows:

<table>
<thead>
<tr>
<th>Field Operation</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>3 \lfloor \log_2 k \rfloor + 7</td>
</tr>
<tr>
<td>Multiplication</td>
<td>6 \lfloor \log_2 k \rfloor + 10</td>
</tr>
<tr>
<td>Squaring</td>
<td>5 \lfloor \log_2 k \rfloor + 3</td>
</tr>
<tr>
<td>Inversion</td>
<td>1</td>
</tr>
</tbody>
</table>

Addition and squaring in GF(2^m) can be implemented relatively fast. Recall field addition over GF(2^m):

\[ a, b \in GF(2^m) \]

\[ a + b = c \text{ where } c = (c_m \ldots c_1 c_0) \text{ and } c_i = (a_i + b_i) \mod 2, c \in GF(2^m) \]

Addition operation over GF(2^m) can be implemented using XOR operand, which is efficiently executed by most of the platforms. Since doubling is based on addition, it can be also implemented fast.

Inversion is only performed once, which doesn’t affect the performance of signature generation so much. Since the most time-consuming and used field operation is multiplication, we will analyze the implementation and optimizations in multiplication. In order to multiply two n-word numbers over GF(2^m), Karatsuba multiplication algorithm and a reduction algorithm implementations have been used.
3.1.2.1. Karatsuba Multiplication Algorithm

In 1962, A.A. Karatsuba proposed an asymptotically faster algorithm for multiplying two n-bit non-negative numbers by using a divide-and-conquer approach. We will analyze Karatsuba multiplication algorithm in detail.

Let A and B be n-bit integers:

\[ A = a_{n-1}2^{n-1} + \ldots + a_12 + a_0 \]
\[ B = b_{n-1}2^{n-1} + \ldots + b_12 + b_0 \]

\[ A \times B = C = c_{2n-2}2^{2n-2} + c_{2n-3}2^{2n-3} + \ldots + c_0 \]

Operands can be divided into two halves as high and low half of the n-bit number. To multiply two n-bit numbers, in conventional multiplication, 4 n/2-bit multiplications are needed. Multiplying by a power of 2 can be done by shift operations; therefore it is efficient. We illustrate the conventional multiplication algorithm as follows:

\[ A = A_h A_l \]
\[ A_h = (a_{n-1} \ldots a_{(n/2)+1} a_{n/2}) \]
\[ A_l = (a_{(n/2)-1} \ldots a_0) \]
\[ B = B_h B_l \]
\[ B_h = (b_{n-1} \ldots b_{(n/2)+1} b_{n/2}) \]
\[ B_l = (b_{(n/2)-1} \ldots b_0) \]

\[ C = (A_h \times 2^{n/2} + A_l) \times (B_h \times 2^{n/2} + B_l) \]
\[ C = (A_h \times B_h) \times 2^n + (A_h \times B_l + A_l \times B_h) \times 2^{n/2} + A_l \times B_l \]

In Karatsuba multiplication, only 3 n/2-bit multiplications are needed to multiply two n-bit numbers.

\[ U = A_h \times B_h \]
\[ V = A_l \times B_l \]
\[ W = (A_h + A_l) \times (B_h + B_l) \]
\[ A_h \cdot B_l + A_l \cdot B_h = W - U - V \]

\[ C = A \cdot B \]
\[ C = U \cdot 2^n + (W-U-V) \cdot 2^{n/2} + V \]

There are three recursive calls in the function, corresponding to three \( n/2 \)-bit multiplications. Instead of applying the algorithm bit-wise, we will work on word base implementation of Karatsuba.

\textit{function Karatsuba}

\textit{input} : \( a, b \) : \( n \)-word integer
\textit{n} : integer

\textit{return} : \((2n)\)-word integer

\textit{local variables} : \( A_h, A_l, B_h, B_l \) : \((n/2)\)-word integer
\( U, V, W \) : \( n \)-word integer

\texttt{begin}

\textit{if} \( n = 1 \) \textit{then}

\textit{return} \( a_0 \cdot b_0 \);

\textit{else}

\( A_h \leftarrow (a_{n-1} \ldots a_{n/2}) \);
\( A_l \leftarrow (a_{n/2-1} \ldots a_0) \);
\( B_h \leftarrow (b_{n-1} \ldots b_{n/2}) \);
\( B_l \leftarrow (b_{n/2-1} \ldots b_0) \);
\( U \leftarrow \text{Karatsuba} (A_h, B_h, n/2) \);
\( V \leftarrow \text{Karatsuba} (A_l, B_l, n/2) \);
\( W \leftarrow \text{Karatsuba} (A_h + A_l, B_h + B_l, n/2) \);

\textit{return} \( U \cdot 2^n + (W-U-V) \cdot 2^{n/2} + V \);

\textit{end if};

\textit{end Karatsuba};
If \( n = 1 \), then these two one-word integers are multiplied. We haven't used Karatsuba algorithm to implement this case.

If \( n \) is a power of 2, then there is \( 3^{\log_2(n/k)} \) number of recursive calls to \( k \)-word multiplications, where \( k \) is also a power of 2. For example, if \( n = 8 \), there are 3 recursive calls to 4-word multiplication, and \( 3^2 \) calls to 2-word multiplication.

### 3.1.2.2. Reduction

To recall the multiplication operation defined over \( F_{2^m} \):

\[
a = (a_{m-1}x^{m-1} + \ldots + a_1x + a_0)
\]

\[
b = (b_{m-1}x^{m-1} + \ldots + b_1x + b_0)
\]

\[
a \ast b = r = (r_{m-1}x^{m-1} + \ldots + r_1x + r_0), \quad \text{where the polynomial}
\]

\[
(a_{m-1}x^{m-1} + \ldots + a_1x + a_0) \ast (b_{m-1}x^{m-1} + \ldots + b_1x + b_0)
\]

is defined to be the remainder, when divided by \( f(x) \), the irreducible polynomial, over \( F_2 \).

Therefore, after the multiplication operation, the output should be reduced by irreducible polynomial. There are various algorithms and ideas proposed for reduction step. The purpose of this routine is to reduce \( 2n \)-word product of two \( n \)-word numbers into an \( n \)-word number based on an irreducible polynomial, where \( n = \lceil m/w \rceil \), \( m \) is the degree and \( w \) is the word size, such that:

\[
p(x) = x^m + x^{k_1} + x^{k_2} + x^{k_3} + 1
\]

A trinomial over \( F_2 \) is a polynomial of the form \( x^m + x^{k_1} + 1 \), where \( 1 \leq k_1 \leq m-1 \). A pentanomial over \( F_2 \) is a polynomial of the form \( x^m + x^{k_1} + x^{k_2} + x^{k_3} + 1 \), where \( 1 \leq k_1 < k_2 < k_3 \leq m-1 \). This routine is implemented by using improved reduction algorithm on trinomial and pentanomial irreducible polynomials. The two following algorithms are combined in the code.

**Input:** The degree of irreducible polynomial, \( m \); the location of the middle coefficient of the polynomial, \( k_1 \); and an operand \( C \) with a length of \( 2wlen \) words, where \( wlen = \lceil m/w \rceil \).

**Output:** The reduced polynomial of length \( wlen \) such that \( C = C \mod P \).
Procedure TrinResidue2(C, m, k1)

0. begin
1. nbits = m mod w;
2. Shiftn = k1/w;
3. for i from pwlen-1 downto wlen
4. \[ C[i-wlen, i-wlen+nbits] = C[i-wlen, i-wlen+nbits] + C[i, i+nbits]; \]
5. \[ C[i-wlen+Shiftn] = C[i-wlen+Shiftn] + C[i]; \]
6. nbits = w;
7. end

Input: The degree of irreducible polynomial, m; the location of the middle coefficients of the polynomial, k1, k2, k3, and an operand C of length 2wlen words, where wlen = \lfloor m/w \rfloor.

Output: The reduced polynomial of length wlen words, such that, C = C mod P.

Procedure PentaResidue (C, m, k1, k2, k3)

0. begin
1. nbits = m mod w;
2. Shiftk1 = k1/w;
3. Shiftk2 = k2/w;
4. Shiftk3 = k3/w;
5. for i from pwlen-1 downto wlen
6. \[ C[i-wlen, i-wlen+nbits] = C[i-wlen, i-wlen+nbits] + C[i, i+nbits]; \]
7. \[ C[i-wlen+Shiftk1] = C[i-wlen+Shiftk1] + C[i]; \]
8. \[ C[i-wlen+Shiftk2] = C[i-wlen+Shiftk2] + C[i]; \]
9. \[ C[i-wlen+Shiftk3] = C[i-wlen+Shiftk3] + C[i]; \]
10. nbits = w;
11. end
3.2. IMPROVEMENTS

In this subsection, we present our optimization techniques and assembly implementation details. The timings and results are also provided for significant performance changes.

3.2.1. General optimization techniques

We have used software engineering techniques to optimize the signature generation for this particular curve. Before starting to optimization, it is important to identify the performance bottlenecks, by using a profiler. This helps to find the parts of the program that consume most of the resources. It is also important to know the priority of the performance parameters for this specific software.

There are various optimization techniques. Some of them are as follows:

- Choosing a better algorithm
- Writing clear, simple code
- Perspective
- Understanding the options of compiler
- Inlining
- Loop unrolling
- Loop jamming
- Loop inversion
- Strength reduction
- Loop invariant computations
- Coding for common case
- Tail recursion elimination
- Lookup table
- Sorting
- Variables
- Function calls
- Digestibility
- String operations
- FP parallelism
- Getting a better compiler
- Stack usage
- Coding in assembly
- Shared library overhead
- Machine-specific optimization

In practice, some of these techniques are applied to ECDSA implementation in general. We have worked on the signature generation process and applied these techniques to particular functions. These are point multiplication, point addition, etc.

As we have mentioned, the most time consuming fundamental operation is field multiplication. Field multiplication is modular multiplication over the finite field. Over GF(2^m), it consists Karatsuba multiplication and reduction. The optimizations are mainly focused on these two parts.

3.2.2. **Field Multiplication**

Field Multiplication performs modular multiplication over GF(2^m) which includes Karatsuba multiplication and reduction routines. As we have mentioned, field multiplication is one of the main operations of point multiplication. Therefore, it plays an important role in the program.

FieldMul ( ){
    KaratMul () //performs Karatsuba multiplication
    Residue () //performs reduction
}
The field multiplication function had 5 parameters. Two of them are for the operands and one for the product of these operands. The other parameter is for the memory reserved for local usage. And one last parameter is passed to retrieve the information about curve. All parameters are passed as pointers.

In ARM, in order to minimize the overhead of parameter passing to functions, it is important to ensure that small functions take four or fewer arguments. The reason is that up to four words of arguments can be passed to a function in registers. These arguments can be integer compatible, (such as, char, shorts, ints and floats all take one word) or structures of up to four words. If more arguments are passed, then the fifth and subsequent words are passed on to the stack, not registers. This causes an extra cost of storing and reloading these words for each call to the function.

In this function, the pointer to the curve information is used to retrieve the word and bit-length of the curve parameters. Since a particular curve is used where parameter sizes are fixed, there is no need to pass this pointer with in calls to field multiplication. Those fixed values are directly assigned to the variables in the function. The number of parameters is reduced to four. Each argument is one word length and they are passed to the function in registers. This machine specific optimization reduces the cost of storing an extra parameter in the calling function and reloadding it in the called function to the stack. Also, this incurs the cost to retrieve the curve information, which is already known; each time the function is executed.

In the next subsections, we will talk about Karatsuba multiplication and reduction functions.

3.2.2.1. Karatsuba Multiplication

- KaratMul ( )

Figure 3.1 illustrates general Karatsuba multiplication for n-word numbers.
Figure 3.1. Karatsuba multiplication algorithm

```
Figure 3.1. Karatsuba multiplication algorithm

This function is for general recursive Karatsuba multiplication. As it is mentioned there are three recursive calls in Karatsuba algorithm. Since recursive calls cause an overhead in performance, $n = 2, 3, \ldots 8$ word Karatsuba multiplications have been implemented in an iterative way, without any recursive calls. These are the most common word lengths for now. General recursive implementation of Karatsuba is used for $n > 8$ word multiplications. If needed, the iterative version of $n > 8$ word multiplications can be implemented. In this function, for these 7 cases and one-word conventional multiplication, the appropriate iterative function is called depending on the word length. If $n > 8$, then the recursive implementation is used.

As we are working on a particular curve, its parameter size and word length is fixed. There is no need to check the word length, since it is known in advance. The appropriate function can be directly called without checking the word lengths. This incurs the cost of comparing and checking.

There are two ways to multiply two 233-bit numbers. If we divide 233 by 32, we will see that it occupies 7 full words and extra 9 bits. One way is to operate on 8 words. Two numbers are multiplied by using 8-word Karatsuba multiplication and do some optimizations when the last words are used, since there are only 9 bits in last words. The other way is to operate on 7 words and 9 bits separately. Here are the timings to compare 8 and 7-word multiplication.
```
Table 3.1. Karatsuba multiplication timings

<table>
<thead>
<tr>
<th>Word length</th>
<th>Signature time (µs)</th>
<th>Field Mult. Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-word</td>
<td>131969 µs</td>
<td>77 µs</td>
</tr>
<tr>
<td>7-word</td>
<td>136453 µs</td>
<td>83 µs</td>
</tr>
</tbody>
</table>

As a result, 8-word multiplication is used to multiply two 233-bit numbers.

KaratMul is used to determine the multiplication function depending on the word length. As 8-word multiplication is chosen, there is no need to have an extra function for this decision. The routine for 8-word multiplication can be directly called from FieldMul function, instead of KaratMul. Removing KaratMul eliminates the overhead of calling an extra function and passing parameters.

- 8-WordMul()

Figure 3.2 shows the diagram of recursive 8-word Karatsuba multiplication. And figure shows the diagram of its iterative version.

Figure 3.2. 8-word recursive Karatsuba multiplication
Figure 3.3. 8 and 4-word Karatsuba multiplication

In the original program, Karatsuba multiplication for 8-word numbers is implemented without any recursive calls to avoid the overhead of recursive calls. So, the calls to \( n/2 = 4 \)-word multiplication routines are replaced with their iterative implementation.

In iterative 8-word multiplication, there are no condition loops to unroll or invert. Also, the operations are performed using only XOR and shift instructions. These instructions can be efficiently executed and there is no other expression that yields the same value but is cheaper to compute to replace them. There are also function calls to one-word multiplication routines. At this point, one optimization technique is inlining. One-word multiplication functions can be defined as macros. This way some speedup can be obtained, however there are some overheads of this
approach. Since the one-word multiplication function is not short, inlining increases code size and makes the code hard to maintain. Also, profilers don’t see macros, so it’s hard to optimize any further once we have done this.

The other option is to write inline ARM assembly. People vary on the approach one should take while writing assembly. Some suggest writing your own assembler version of a function from scratch while others recommend taking the compiler’s version as a starting point and just fine-tuning it. We have decided to write our own assembly for the functions. At this point, there were two scenarios to choose. One of them was to write the 8-word multiplication illustrated in Figure 3.4 all in assembly. The second option was to add one more step, by calling 4-word multiplication from 8-word multiplication (Figure 3.3) and write both in assembly. For both scenarios one-word multiplication will be in assembly. In the original code, we compare the execution time differences of two different schemes as follows:

Table 3.2. Optimized Karatsuba multiplication

<table>
<thead>
<tr>
<th>Karatsuba Mult. In C</th>
<th>Signature time (μs)</th>
<th>Field Mult. Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using iterative 8-word mul.</td>
<td>129886 μs</td>
<td>76 μs</td>
</tr>
<tr>
<td>Using 8-word mul, which calls iterative 4-word mul</td>
<td>131969 μs</td>
<td>77 μs</td>
</tr>
</tbody>
</table>

The difference between using these two schemes is computed as follows:

\[(131969 - 129886) / 131969 = 0.015 = 1.5 \%\]

Using iterative 8-word multiplication in C is 1.5% faster than the other version in signature time. Since the difference is not so significant, it will be more clear and maintainable to implement the 8-word multiplication that calls iterative 4-word multiplication, rather than implementing one long routine.
function 8-WordMul{
    \( A_h \leftarrow (a_7 \ a_6 \ a_5 \ a_4); \)
    \( A_l \leftarrow (a_3 \ a_2 \ a_1 \ a_0); \)
    \( B_h \leftarrow (b_7 \ b_6 \ b_5 \ b_4); \)
    \( B_l \leftarrow (b_3 \ b_2 \ b_1 \ b_0); \)
    \( U \leftarrow 4\text{-WordMul}(A_h, B_h) \)
    \( V \leftarrow 4\text{-WordMul}(A_l, B_l) \)
    \( W \leftarrow 4\text{-WordMul}(A_h \oplus A_l, B_h \oplus B_l) \)
    Result \( \leftarrow U \cdot 2^8 + (W - U - V) \cdot 2^4 + V; \)
}

- 4-WordMul ( )

This routine performs Karatsuba multiplication of two operands of 4-word length. There are only shift operations and function calls to one-word multiplication functions. This function is implemented in ARM assembly.

4-WordMul ( ){
    \( A_h \leftarrow (a_3 \ a_2); \)
    \( A_l \leftarrow (a_1 \ a_0); \)
    \( B_h \leftarrow (b_3 \ b_2); \)
    \( B_l \leftarrow (b_1 \ b_0); \)
    \( D \leftarrow 1\text{-WordMul} (A_3, B_3); \)
    \( E \leftarrow 1\text{-WordMul} (A_2, B_2); \)
    \( F \leftarrow 1\text{-WordMul} (A_3 \oplus A_2, B_3 \oplus B_2); \)
    \( U \leftarrow D \cdot 2^2 + (F - D - E) \cdot 2 + E; \)
    \( D \leftarrow 1\text{-WordMul} (A_1, B_1); \)
    \( E \leftarrow 1\text{-WordMul} (A_0, B_0); \)
    \( F \leftarrow 1\text{-WordMul} (A_1 \oplus A_0, B_1 \oplus B_0); \)
Here is the table of instruction type and number of assembly language implementation of 4-word Karatsuba multiplication.

**Table 3.3. 4-word Karatsuba in assembly**

<table>
<thead>
<tr>
<th>Operation Type</th>
<th># of instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>25 (43 memory reads)</td>
</tr>
<tr>
<td>Store</td>
<td>5 (18 memory writes)</td>
</tr>
<tr>
<td>Other</td>
<td>61</td>
</tr>
</tbody>
</table>

Number of load and store instructions and number of actual memory read and writes differ because of load and store multiple register instructions. The ARM instruction sets include instructions that load and store multiple registers to and from memory. Multiple register transfer instructions are efficient for moving the contents of several registers to and from memory. The use of multiple register transfer instructions provides the following advantages:

- Smaller code size
There is only one instruction fetch overhead for multiple load and store, rather than many instruction fetches.

Only one register writeback cycle is required compared to one for each register.

1-WordMul()

The bottom step of Karatsuba multiplication is to multiply two one-word integers. One-word multiplication can be implemented in various ways to improve efficiency depending on the platform and environment.

In our case, it has been implemented using lookup tables. Each time the function for 32-bit multiplication is called; a lookup table for the first operand is generated. The lookup table consists of products of the first operand, say a. There are two options available for lookup table usage. One is creating a table according to the result of multiplying a, by three bits of second operand, b, at a time. Since each time three bits of b are used, there are \(2^3 = 8\) combinations, which are 0, a, 2a, ..., 7a. The other is using 4 bits of b at a time. Then, there are \(2^4 = 16\) possible results from 0 to 15a. These lookup tables are defined as macros, so the calculations are performed in the macros. These two lookup tables have some advantages and disadvantages in terms of efficiency. Since the tables are created in a macro, so in the function, each time the function is called, a table is generated. That's why, it is efficient to use a smaller table, which is relatively fast to implement than the larger table. After the table is created, the multiplication operation is performed. Compared to the 16-entry table, the 8-entry table needs more steps to complete the multiplication of two one-word numbers. The following table compares the effect of using 3-bit and 4-bit lookup table to the execution time of the signature generation. B-233 curve and original code is used for both.
Table 3.4. Lookup table in 1-word multiplication

<table>
<thead>
<tr>
<th>Lookup Table</th>
<th>Signature Time (µs)</th>
<th>Field Mult. Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-bit</td>
<td>129886 µs</td>
<td>77 µs</td>
</tr>
<tr>
<td>4-bit</td>
<td>148053 µs</td>
<td>89 µs</td>
</tr>
</tbody>
</table>

One-word multiplication is also written in ARM assembly. First, the lookup table scheme for one-word multiplication is analyzed to determine if it is efficient to implement this in assembly. When the lookup table is generated, it is stored in memory. For each data to complete one word multiplication, memory access is inevitable. It is known that memory access costs relatively more than most of the operations. There is a way to implement one word multiplication over GF(2^m) without using lookup table. By using only shift and XOR operations, it is possible to implement one-word multiplication with less instructions and faster. The pseudocode of the C version of this scheme is as follow:

```c
function One-word ()
    w ← wordSize ← 32;
    i ← 1;
    while (i < 32)
        if (a[w-i] = 1) then
            tmp[1] ← tmp[1] ⊕ (b >> i);
            tmp[0] ← tmp[0] ⊕ (b << (w-i));
            i ← i+1;
        if (a[w-i] = 1) then
            tmp[1] ← tmp[1] ⊕ (b >> i);
            tmp[0] ← tmp[0] ⊕ (b << (w-i));
```
i ← i+1

\[ c[0] ← \text{tmp}[0]; \]

\[ c[1] ← \text{tmp}[1]; \]

In iteration, two bits of the first argument are checked starting from the most significant ones. So, the loop is executed \(31/2 = 15\) times. If the check bit of the first argument is 1, second argument is shifted and XORed with the result from the previous step. The result of this multiplication is two words, so, it is kept in two registers as low and high word of the result. In the pseudocode, \text{tmp}[0] is the value of low word and \text{tmp}[1] is the value of high word of the result.

The assembler version of this code is much more efficient than its C version. This is because; ARM instruction set is especially rich for shift operations. Also, conditional execution plays an important role in assembler implementation. One other advantage of implementing this version of one-word multiplication in assembly is that there is no need to use extra memory for lookup table. Here are the timings for assembly implementation of one word multiplication.

The assembler version of the first iteration of while loop is as follows:

```
__asm{
    mov r4, #0
    mov r5, #0
    teq r5, r1, LSL #1
    eorcs r5, r5, r2, LSR #1
    eorcs r4, r4, r2, LSL #31
    eormi r5, r5, r2, LSR #2
    eormi r4, r4, r2, LSL #30
    ... }
```

- \text{MOV\{<cond>\} \{S\} \text{Rd, <shifter_operand>}}
MOV (Move) instruction is used to move a value from one register to another, put an immediate value into a register, or perform a shift without any arithmetic or logical operation.

If ConditionPassed (<cond>) then

\[
\text{Rd} = \text{<shifter_operand>}
\]

if \( S = 1 \) and \( \text{Rd} = \text{R15} \) then

\[
\text{CPSR} = \text{SPSR}
\]

else if \( S = 1 \) then

\[
\text{N Flag} = \text{Rd}[31]
\]

\[
\text{Z Flag} = \text{if Rd} = 0 \text{ then 1 else 0}
\]

\[
\text{C Flag} = \text{<shifter_carry_out>}
\]

\[
\text{V Flag} = \text{unaffected}
\]

- **TEQ{<cond>} Rd, <shifter_operand>**

TEQ (Test Equivalence) instruction is used to test if two values are equal.

If ConditionPassed (<cond>) then

\[
\text{<alu_out>} = \text{Rn EOR <shifter_operand>}
\]

\[
\text{N Flag} = \text{<alu_out>}[31]
\]

\[
\text{Z Flag} = \text{if <alu_out>} = 0 \text{ then 1 else 0}
\]

\[
\text{C Flag} = \text{<shifter_carry_out>}
\]

\[
\text{V Flag} = \text{unaffected}
\]

Here N, Z, C and V (Negative, Zero, Carry and Overflow) bits are condition code flags. After the instruction is executed, N is assigned the value of the last bit (31st bit) of the output. So, if it is 1, that means the number is negative, otherwise positive. Z is used to check if the output is zero or not. If it is zero, then Z is 1; else Z is 0. C is to check carry after the instruction is executed. If carry is 1, C gets 1; otherwise, C is 0. V bit is used to check if there is an overflow. It is 1, if there is an overflow; 0 if there is not. In this specific instruction, N, Z and C are affected.
The reason this instruction is used here is that the value of the first argument (content of r1) remains unchanged and the subsequent instructions are conditionally executed according to N and C condition flags. If C = 1, then the next two eorcs instructions, are executed. And if N = 1, then the eormi instructions are executed.

- **EOR{<cond>} {S}** Rd, Rn, <shifter_operand>

  The EOR (Exclusive-OR) instruction performs a bit-wise Exclusive-OR of the values of register Rn with the value of <shifter_operand>, and stores the result in the destination register Rd. The condition code flags are updated if there is a suffix S.

  If ConditionPassed {<cond>} then
  
  \[ \text{Rd} = \text{Rn} \text{ EOR} \text{ <shifter_operand>} \]
  
  if \( S = 1 \) and \( \text{Rd} = \text{R15} \) then
  
  CPSR = SPSR
  
  else if \( S = 1 \) then
  
  N Flag = Rd [31]
  
  Z Flag = if \( \text{Rd} = 0 \) then 1 else 0
  
  C Flag = <shifter_carry_out>
  
  V Flag = unaffected

  EORCS is the exclusive-OR instruction with the CS condition. It checks the carry bit (C) before executing the instruction. If C is 1, then it executes the instruction, if not, it doesn’t. This instruction is efficient because, when C is 0, there is no need to perform the instruction, as XORing with 0, doesn’t affect the result.

  EORMI is the exclusive-OR instruction with MI condition. It checks the negative bit (N) before executing the instruction. If N is 1, the instruction is executed, if not, it is not. Since XORing with 0 doesn’t affect the result, it is efficient not to perform the instruction if N= 0.
The while loop in C implementation has been unrolled in assembly. It is executed 15 times for 32-bit multiplication. The type and number of instructions in assembly are as follows:

**Table 3.5. 1-word multiplication in assembly**

<table>
<thead>
<tr>
<th>Operation Type</th>
<th># of instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>-</td>
</tr>
<tr>
<td>Store</td>
<td>1 (2 memory writes)</td>
</tr>
<tr>
<td>Other</td>
<td>81</td>
</tr>
</tbody>
</table>

The timings for assembly implementation of one word multiplication are in Table 3.6.

**Table 3.6. 1-word multiplication optimization results**

<table>
<thead>
<tr>
<th>1-WordMul</th>
<th>Signature Time (μs)</th>
<th>Field Mult Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before optimizations, in C</td>
<td>129886 μs</td>
<td>77 μs</td>
</tr>
<tr>
<td>After optimizations in assembly</td>
<td>107928 μs</td>
<td>60 μs</td>
</tr>
</tbody>
</table>

3.2.2.2. Reduction

In implementation, reduction function needs 3 arguments. These are a pointer to the output; m, the degree of polynomial; and poly[ ], an integer array keeps the locations of the middle coefficients of the polynomial, such that,
poly[0]=k1, poly[1] = k2, poly[2] = k3. In trinomial irreducible polynomial, there is only one middle term, k1.

For B-233 particular curve m is 233 and k1 is 74. Since these parameters are fixed, there is no need to pass them with reduction function calls. This reduces number of parameters to only one, which is C.

As the irreducible polynomial of our particular curve is trinomial, the tests and instructions for pentanomial polynomials are removed.

Loop unrolling technique can be applied in reduction routine. There is a 'for' loop, which is executed only 7 times. This way we eliminate the branching and testing entirely. Also, in the original implementation, there are some local calculations, depending on word and key length. Since word and key length is fixed for the particular curve, they can be precomputed and directly assigned to local variables, without any additional calculation. By this way, time consumed for unnecessary calculations are avoided. Also, there are conditional if statements depending on some of these values. As they are known, there is no need to spend time for comparing them. The product of two 233-bit numbers occupies 14 full words and half of the 15th word. In reduction process, some of the shift operations on the last word can be avoided. If the used bits are shifted, then remaining bits will be all zero.

It is also possible to consider assembler implementation of this function, because at many points performance can be improved. The assembler version of this algorithm can be implemented by mainly using exclusive-OR and shift operations, which are efficient to execute. The Table 3.7 shows the number and type of instructions in assembly implementation of reduction.
Table 3.7. Reduction in assembly

<table>
<thead>
<tr>
<th>Operation Type</th>
<th># of instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>3 (15 memory reads)</td>
</tr>
<tr>
<td>Store</td>
<td>3 (12 memory writes)</td>
</tr>
<tr>
<td>Other</td>
<td>37</td>
</tr>
</tbody>
</table>

After the optimizations on reduction code in C, compared to the original code, the speedup is \((129886-108366)/129886 = 0.166\), that is 16.6%. After these optimizations in C, we have implemented the optimized C code in assembly, and the overall gain in performance is \((129886-103056)/129886 = 0.206\), that is 20.6%.

Table 3.8. Reduction optimization results

<table>
<thead>
<tr>
<th>Reduction Algorithm</th>
<th>Signature Time((\mu s))</th>
<th>Field Mult. Time((\mu s))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before optimization in C</td>
<td>129886 (\mu s)</td>
<td>77 (\mu s)</td>
</tr>
<tr>
<td>After optimization in C</td>
<td>108366 (\mu s)</td>
<td>67 (\mu s)</td>
</tr>
<tr>
<td>After optimization in assembly</td>
<td>103056 (\mu s)</td>
<td>66 (\mu s)</td>
</tr>
</tbody>
</table>

3.3. ECDSA TIMINGS AFTER OPTIMIZATIONS

In this subsection we present the overall result after the optimization in ECDSA code. The optimized code is executed both on ARM7 and ARM9 processors. The clock rate for ARM7 is set 80MHz and for ARM9 200Mhz. Table 3.9 and Table 3.10 summarize the results for ARM7 and ARM9, respectively.
Table 3.9. ECDSA optimization results on ARM7

<table>
<thead>
<tr>
<th>ECDSA</th>
<th>Signature Time (μs)</th>
<th>Field Mult. Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before optimization</td>
<td>129886 μs</td>
<td>77 μs</td>
</tr>
<tr>
<td>After optimization</td>
<td>76624 μs</td>
<td>47 μs</td>
</tr>
</tbody>
</table>

Table 3.10. ECDSA optimization results on ARM9

<table>
<thead>
<tr>
<th>ECDSA</th>
<th>Signature Time (μs)</th>
<th>Field Mult. Time (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before optimization</td>
<td>36498 μs</td>
<td>22 μs</td>
</tr>
<tr>
<td>After optimization</td>
<td>26358 μs</td>
<td>17 μs</td>
</tr>
</tbody>
</table>

The overall improvement of the ECDSA code on ARM7 processor is 

\[
\frac{(129886-76624)}{129886} = 0.410, \text{ that is } 41\%.
\]

And on ARM9 processor, the speedup of the ECDSA code is 

\[
\frac{(36498-26358)}{36498} = 0.278, \text{ that is } 27.8\%.
\]
4. CONCLUSION

In this thesis we studied the optimization and porting of ECDSA on ARM7 processor, using a particular curve B-233 with a particular polynomial of the form \( x^m + x^k + 1 \) over GF(\(2^m\)). This curve has been recommended by NIST for U.S. government use. The optimized algorithm is overall 41.0% faster than the original program for general curves.

Our goal was to optimize signature generation in terms of time and space. We mainly worked on field multiplication, as it is the most time consuming component of signature process. Field multiplication consists of multiplication and reduction. Over the field GF(\(2^m\)), Karatsuba multiplication algorithm was used for multiplication and for reduction a reduction algorithm defined in thesis [15] was used. We optimized these two routines in C using general software engineering techniques such as loop unrolling, strength reduction, inlining etc. In addition to these, machine specific optimizations for writing efficient C for ARM were applied to these routines. In C, it is not possible to access some parts of the processor. That's why, we have implemented 8-word Karatsuba multiplication and reduction algorithm using ARM assembly language. For assembly implementation of these routines we have chosen to use different methods at some parts to take advantage of using assembly language. While implementing in assembly language, we have also applied some techniques to prevent potential stalls. We have developed certain techniques and strategies to work on particular curves.

We have executed the algorithm on ARM Software Development Toolkit, Version 2.50. In order to measure the actual performance; we have made use of features of the toolkit. The timings before and after the optimizations that are given throughout the thesis are the average values of several executions. Table 3.9 and Table 3.10 summarize the average signature generation timing overall on ARM7 and ARM9 processor respectively.
This work can be continued with optimization of other set of particular curves over the field GF($2^m$) by using the same techniques applied to our curve. The assembly routines written for 8-word Karatsuba multiplication can be applied to n-word Karatsuba multiplication functions for other particular curves or general curves.
BIBLIOGRAPHY


APPENDIX
APPENDIX: RECOMMENDED ELLIPTIC CURVES FOR FEDERAL GOVERNMENT USE (July 1999)

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

1.1. PARAMETER CHOICES

The parameter choices for an elliptic curve are discussed in the following subsections

1.1.1. Choice of Key Lengths

There are two main parameters for elliptic curve cryptography that are the elliptic curve E and a designated point G on E called the base point. The base point has order r, a large prime. The product f and r for some integer f (the cofactor) not divisible by r gives the number of points on the curve. The cofactor is taken as small as possible; therefore the private and public keys will be approximately the same length.

1.1.2. 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. The elements of GF(p) 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. m is also 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.

1.1.3. Choice of Basis

In binary fields, bit strings are interpreted in different ways depending on the basis choice for the field. There are two types of bases, which are polynomial and normal basis.

- A polynomial basis is specified by an irreducible polynomial modulo 2, called the field polynomial, p(t). The bit string \((a_{m-1} \ldots a_2 a_1 a_0)\) represents the polynomial
  \[ a_{m-1}t^{m-1} + \ldots + a_2 t^2 + a_1 t + a_0 t^0 \]
  over GF(2). The field arithmetic is implemented as polynomial arithmetic modulo p(t).

- A normal basis is specified by an element \(\theta\) of a particular kind. The bit string \((a_0 a_1 a_2 \ldots a_{m-1})\) represents the element
  \[ a_0 \theta + a_1 \theta^2 + a_1 \theta^{2^2} + \ldots + a_{m-1} \theta^{2^{m-1}} \]

There are many polynomial and normal bases from which to choose. The following procedures are commonly used to select a basis representation.

- Polynomial Basis: If an irreducible trinomial \(t^m + t^k + 1\) exists over GF(2), then the field polynomial \(p(t)\) is chosen to be the irreducible trinomial with the lowest-degree middle term \(t^k\). If no irreducible trinomial exists, then one selects instead a pentanomial \(t^m + t^a + t^b + t^c + 1\). The particular pentanomial chosen has the following properties: the second term \(t^a\) has the lowest degree \(m\); the third term \(t^b\) has the lowest degree among all irreducible pentanomials of degree \(m\) and second term \(t^a\); and the fourth term \(t^c\) has the lowest degree among all irreducible pentanomials of degree \(m\), second term \(t^a\), and third term \(t^b\).
• Normal Basis: Choose the Type T low-complexity normal basis with the smallest T.

For each binary field, the parameters are given for the above basis representations.

1.1.4. **Choice of Curves**

There are two kinds of curves:

- Pseudo-random curves: Their coefficients are generated from the output of a seeded cryptographic hash.
- Special curves: the coefficients and underlying field has been pre-selected to optimize the efficiency of the elliptic curve operations.

For each size, the following curves are given:

A pseudo-random curve over GF(p).
A pseudo-random curve over GF(2^m).
A special curve over GF(2^m) called a Koblitz curve or anomalous binary curve.

The pseudo-random curves are generated via the SHA-1 based method given in the ANSI X9.62 and IEEE P1363 standards.

1.1.5. **Choice of Base Points**

Any point of order r can serve as the base point. Though users may want to generate their own base points, each curve is supplied with a sample base point \( G = (G_x, G_y) \).

1.2. **CURVES OVER FINITE FIELDS**

For each prime \( p \), a pseudo-random curve

\[ E: y^2 = x^3 - 3x + b \pmod{p} \]
of prime order \( r \) is listed. (Thus, for these curves, the cofactor is always \( f = 1 \).) The following parameters are given:

- The prime modulus \( p \)
- The order \( r \)
- the 160-bit input seed \( s \) to SHA-1 based algorithm
- The output \( c \) of the SHA-1 based algorithm
- The coefficient \( b \) (satisfying \( b^2 \equiv -27 \pmod{p} \))
- The base point \( x \) coordinate \( G_x \)
- The base point \( y \) coordinate \( G_y \)

The integers \( p \) and \( r \) are given in decimal form; bit strings and field elements are given in hex.

1.3. CURVES OVER BINARY FIELDS

There are different kinds of binary curves. Here, we present information about our curve. For each field degree \( m \), the pseudo-random curve has the form:

\[
E: y^2 + xy = x^3 + x^2 + b
\]

The cofactor is 2 for each pseudorandom curve. The coefficients of the pseudo-random curves, and the base point coordinates for both curves, are given in representation of the polynomial basis and normal basis.

For each field degree \( m \), the following parameters are given:

Field Representation:
- \( T \): The normal basis type
- The field polynomial (a trinomial or pentanomial)

Pseudo-random curve:
- \( r \): The base point order
Pseudo-random curve (Polynomial Basis representation):

- \( b \): The coefficient
- \( G_x \): The base point \( x \) coordinate
- \( G_y \): The base point \( y \) coordinate

Integers such as \( T \), \( m \), and \( r \) are given in decimal form; field elements and bit strings are given in hexadecimal representation.

1.4. DEGREE 233 BINARY FIELD

\( T = 2 \)

\[ p(t) = t^{233} + t^{74} + 1 \]

Curve B-233

\( r = 6901746346790563787434755862277025555839812737345013555379383634485463 \)

Polynomial Basis:

\( b = 066\ 647\text{ede}e6c\ 332c7f8c\ 0923bb58\ 213b333b\ 20e9ce42\ 81fe115f\ 7d8f90ad \)

\( G_x = 0fa\ c9dfcbac\ 8313bb21\ 39f1bb75\ 5fedefbc\ 391f8b36\ f8f8eb73\ fd558b \)

\( G_y = 100\ 6a08a419\ 03350678\ e58528be\ bf8a0bef\ f867a7ca\ 36716f7e\ 01f81052 \)