University College London

Secure Implementation of ECDSA Signatures in Bitcoin

by

DI WANG

Supervisor: Dr. Nicolas T. Courtois

MSc in Information Security

This report is submitted as part requirement for the MSc in Information Security at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

September 17, 2014

Abstract

Bitcoin, a decentralized peer-to-peer cryptocurrency system, has recently gained a lot of popularity among users. In a Bitcoin transaction, users communicate depending on cryptographic proof rather than trust on third parties. Therefore, as one of the main building blocks of Bitcoin, Elliptic Curve Digital Signature Algorithm is used to prove Bitcoin ownership, which plays a pivotal role in Bitcoin transaction. The security of Bitcoin mostly relies on the security of ECDSA algorithms.

This thesis covers the steps of understanding how ECDSA works, and how it is used to create a new Bitcoin transaction. Also the speed performance in signature generation is measured and improved. Based upon it, different versions of security countermeasures are implemented to protect against side channel attacks. Finally, all the versions including both non-protective and protective implementations are tested and evaluated.

Acknowledgements

First and foremost, I would like to express my deepest gratitude for my supervisor, Dr. Nicolas T. Courtois, without whom I could not finish this project. His invaluable guidance and suggestions helped me overcome many obstacles.

I also thank my cooperator, Jiahui Yang, for completing this project together with me, without your help and encouragement, I cannot make such progress on this thesis.

Finally, a massive thank to my family for their emotional and financial support to encourage me doing my best.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Bitcoin is a decentralized peer-to-peer cryptocurrency launched in 2009. When it first appeared, it was not treated seriously in financial market, until 2011, a Japanese company MtGox started to exchange Bitcoins against money, Bitcoin arose media's attention and an increasing number of miners got involved in mining Bitcoins. During Cyprus banking crisis, Bitcoin price got a boost, and Figure 1.1 shows the market price of Bitcoin in USD. Clearly, the price of Bitcoin increased dramatically, and at the end of 2013 and beginning of 2014, it reached to the peak at a price of 1,151 USD.

Figure 1.1: Market Price of Bitcoin in USD from 2012-2014

1.1 Motivation and Goal

Because of the popularity of Bitcoin, one of its main building blocks, Elliptic Curve Digital Signature Algorithm (ECDSA) is hotly discussed as well. Since Bitcoin is decentralized, transactions will not go through to a third party so that two willing parties communication depend on cryptographic proof instead of trust. Therefore ECDSA, a digital signature scheme based on elliptic curve cryptography, is used to prove Bitcoin ownership and sign Bitcoin transactions. ECDSA keys used to generate Bitcoin addresses and sign transactions are derived from the certain parameters. NIST has been recommending *secp256r1* to be used [43], however, unlike most applications, Bitcoin protocol chooses *secp256k1* which is almost never used to generate private and public key pairs.

Due to this characteristic of Bitcoin protocol, ECDSA algorithm used in Bitcoin has become a hot topic among cryptographers. And it motivates me to know how ECDSA algorithms are used and implemented in Bitcoin protocol, and how to make it more secure.

In this thesis, several goals need to be achieved based on a Java version of implementation:

- Create an ECDSA prototype to produce signatures and then verify it, the implementation should use *secp256k1* curves for key pair generation.
- Manually create a new Bitcoin transaction from a given raw transaction, then integrate it with the prototype ECDSA algorithm to sign and verify it.
- Improve the signing performance of ECDSA prototype.
- Add security countermeasures against side channel attacks (simple power analysis, differential power analysis and fault analysis) based on the improved version and evaluate their security and speed performance in signature generation.

1.2 Structure of the Thesis

The thesis is organized as follows. In chapter 2, technical background of Bitcoins, elliptic curve cryptography (ECC) and side channel attacks are presented, related work regarding to alternative public key cryptography in Bitcoin, performance improvement in ECC and ECDSA as well as side channel countermeasures are reviewed.

In chapter 3, algorithms and parameters needed for java implementation such as ECDSA algorithms, Bitcoin transaction processes and methods for performance and security improvement are specified.

In chapter 4, detailed implementation descriptions on ECDSA prototype, ECDSA in projective coordinates to improve signing speed and side channel countermeasures are presented.

The testing results measuring signing time of the new implemented versions are illustrated and evaluated in chapter 5. Finally, future work and conclusion are provided in chapter 6.

Chapter 2

Background and Related Work

2.1 Bitcoin Overview

Bitcoin is a decentralized peer-to-peer digital currency based on public-key cryptography first proposed by Satoshi Nakamoto [49] in 2008, and fully operated in 2009. It is a proof-of-work based cryptocurrency which allows miners to mint Bitcoin through computation. Different from other traditional financial currencies, willing parties make secure transactions relying on cryptographic protocols rather than trust on third parties. Since there are no central authorities to keep records of transactions, they are confirmed by consensus procedure and stored in a distributed manner. Thus, privacy of users in public transactions is protected using pseudonyms called Bitcoin addresses.

Bitcoin has quite a lot of nice properties [3, 14] , making it become the largest cryptocurrency worldwide:

- **Partially anonymous** : Although all transactions are publicly known, it is possible for users to use Bitcoin addresses as pseudonyms. Also it encourages that for one particular user in different transactions, different Bitcoin addresses should be used. This way can maintain the anonymity in some degree.
- **Transferable**: There is no central authority in Bitcoin transaction, so all transactions are made public to be validated. Since no financial network exists, Bitcoin is transferable.
- **Payment irreversible**: Once owner spend their Bitcoins, he will no longer have the

ability to retrieve them without the recipient's consent. This feature makes sure that asking the chargeback to inverse the transaction is no longer possible.

- **Cheap and fast**: Different from modern banking transaction, Bitcoin transaction does not ask for transaction fees unless payer requires. What is more, Bitcoin transaction will be validated in a few minutes, which is a lot faster than transferring money via banks.
- **Secure**: For digital cash schemes, double spending is a common attack. However, in Bitcoin, it can be avoided by validating Bitcoin transaction in public. In addition, Bitcoin can prevent this attack if the hash function and digital signature algorithm is secure enough.
- **Not inflationary**: Different from regular flat currency, there is a limitation of total number of Bitcoins. According to the original specification, only 21 million Bitcoins can be created. Therefore, inflation is no longer a problem in Bitcoin world.

2.1.1 Main Building Blocks

Time stamping

Time stamping is a use of electronic timestamp to prove the temporal sequence among events. It is first proposed by Haber and Stornetta in 1991 [26] certifying when an electronic document created or last modified.

Since Bitcoin transactions are publicly announced, timestamp system is needed for participants to agree on the order of the transaction. The principle of timestamp is that information stored in the blocks is arranged in a chain, and hash of block of items is timestamped while each timestamp includes previous timestamps in its hash value. With the timestamp, data has existed at the time can be proved.

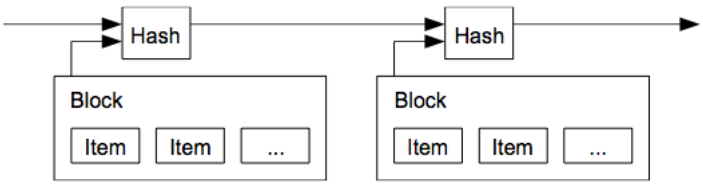

Figure 2.1: Timestamp in Bitcoin

Proof of work

To validate a Bitcoin transaction, proof-of-work is used which is originally proposed in Adam Back's Hashcash [1]. It is a cryptographic scheme that is easy to verify but hard to produce the defined form of result without a large amount of computational work. The principle of proof of work is that for many hash functions, finding an input to generate a value with a predefined leading substring is a low probability event and requires a lot of trail and error. In Bitcoin, finding such suitable inputs is used in block creation which is called mining (Section 2.1.2). The proof-of-work block in Bitcoin contains transaction to be validated, hash of previous block (implements the timestamp) and a nonce. The hash algorithm used is SHA256, and different nonces will be tried until the SHA256 hash value of the block satisfied with the requirements. This will consume a lot of computational power and it is a proof of the miners' work.

Besides validating the transaction to ensure the block integrity, proof-of-work protocol is also used to regulate the Bitcoin supply and reward miners [45].

Merkle trees

Merkle tree is an important element in Bitcoin. It is a binary tree of hashes proposed by Ralph Merkle [39] that is used to verify data integrity efficiently and securely.

Figure 2.2 [45] is an example of a Merkle tree. In this tree, every non-leaf node is the 'concatenate then hash' value of its child node, and the leaves are computed over data blocks. The final hash value - n in the Merkle tree - is called Merkle root which will be stored in the block header. Since the non-leaf node is expected to have two child nodes, the missing child node is a special case of Merkle tree, just like child node n_1. In this kind of situation, the solution in Bitcoin is straightforward. When forming the nodes in a row, if there are an odd number of nodes, the last node will be duplicated.

One of the strengths of Merkle tree is that there is no need to recompute the hash of all data if one data block changes. For example, if the block d_{00} changed, only one branch from n_{00}, n_0 and finally root n will be recomputed. It requires a much smaller number of hash computations and makes the process more efficient.

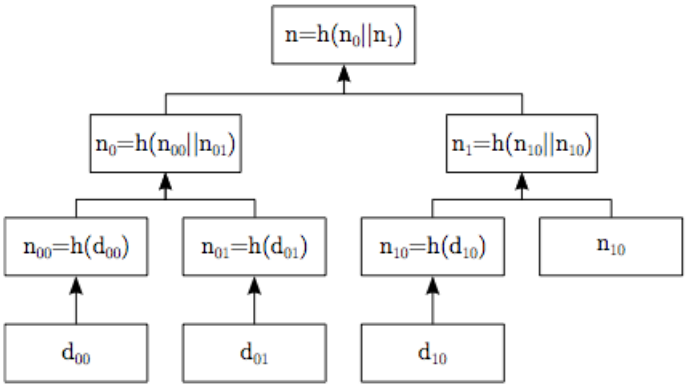

Figure 2.2: Merkle Tree

Signatures and Hashes

In Bitcoin transaction, two cryptographic primitives are used to prevent malicious users breaking the system.

A **digital signature** is used to make sure that the information is signed by the claimed person as well as to test whether the information is modified by some malicious people. The signature process contains signature generation and signature verification. Given a message, the signatory generates a signature by using his private key, and the verifier can use signatory's public key to verify the message's authenticity. Figure 2.3 [35] shows the whole process. In fact, instead of signing on the message directly, a cryptographic hash function is applied to the original message to produce a message digest for performance reason.

Digital Signature Algorithm (DSA) was the first digital signature scheme accepted legally by government [31] and proposed by NIST in August 1991. This algorithm is a variant of ElGamal Signature Algorithm. ECDSA is a digital signature scheme based on public key cryptosystem ECC (Section 2.2), instead of working in a subgroup of $Z_p{}^*$ in DSA, ECDSA works in the group of elliptic curve$E(Z_p)$. It has been standardized by many standard committees such as ISO, ANSI, IEEE and FIPS [30]. The specific signing and verification of ECDSA will be detail in Section 3.1.3.

Hash function is any function that maps data with arbitrary length to a fixed-size, hard-to-inverse value. A little modification on the inputs will produce outputs with big difference. Therefore, hash functions can be used to ensure data integrity.

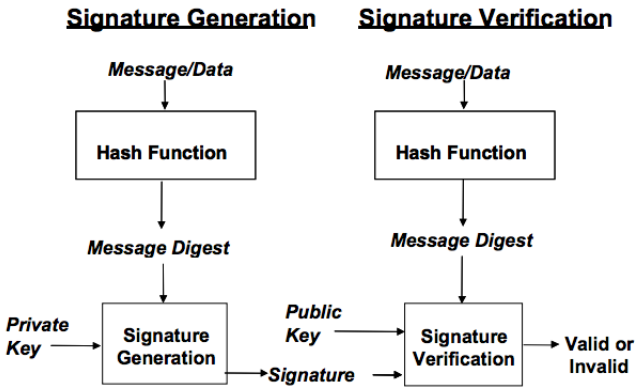

Figure 2.3: Digital Signature Process

Hash functions are adopted by Bitcoin system mainly in 1) Bitcoin addresses generation and 2) transactions and blocks generation. Bitcoin addresses are generated by hashing the public key of ECDSA using hash algorithms SHA256[1] and RIPEMD160[2] , while RIPEMD160 is used after SHA256. As for Bitcoin transaction and blocks generation, two consecutive SHA256 hashes are used. What is more, SHA1[3] is also used in transaction signature generation and verification with ECDSA algorithms.

2.1.2 Bitcoin Architecture

Blocks and Block Chain

Blocks in Bitcoin hold recent transaction contents that have not been recorded in previous blocks. Each block contains transaction information, proof of work and reference to the hash of previous block. The block header structure is shown in Table 2.1.

Description	Version	Previous Hash	Merkle Root	Timestamp	Bits	Nonce
Size (Bytes)	4	32	32	4	4	4

Table 2.1: Block Header Structure

Previous hash is the hash value of previous block header. Merkle root, as explained before, is used to verify transaction integrity. Timestamp is the time when the block was generated. Bits

[1]For more details about SHA256, see FIPS N. 180-2: Secure hash standard.

[2]For more details about RIPEMD160, see Preneel B, Bosselaers A, Dobbertin H. The cryptographic hash function RIPEMD160, 1997.

[3]For more details about SHA1, see Eastlake D, Jones P. US secure hash algorithm 1, 2001.

designate the difficulty of the proof of work, and different nonces are tried to meet Bitcoin proof of work requirements.

These series of blocks form a block chain, which is a shared public ledger. The sequence of the blocks in the block chain is based on the time that the transactions confirmed, and every block is built on top of previous one. To create new blocks, miners can choose blocks including transactions they agree with to extend. However, if some of other miners choose to work on different blocks, the chain will have more branches, shown in Figure 2.4 [55]. To regulate these competing branches, Bitcoin has the longest chain rule, that is only the longest chain will be accepted, and others will be dropped. In the figure, blocks in black form the longest chain and short chains in purple will be dropped.

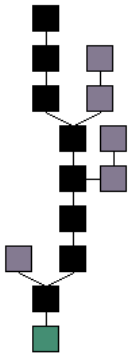

Figure 2.4: Longest Chain Rule Example

Transaction

Transaction in Bitcoin is the process of transferring Bitcoin ownership from one Bitcoin address to another. A Bitcoin address is a 160-bit hash of ECDSA public key and stored in Bitcoin wallet together with its related private key. Bitcoin wallet stores one or more Bitcoin addresses and each one can be used only once.

One Bitcoin transaction contains zero or more inputs and outputs. An input is reference to outputs of another previous transactions, and the values of transactions are added up and used in the current transaction. An *input* normally contains three parts: **Previous tx** is hash of previous transaction, **Index** is referenced transaction output, and **ScriptSig** contains a signature and a public key. The ECDSA signature is generated by signing the hash of the transaction, and public key belongs to the payer. Both the signature and public key prove the

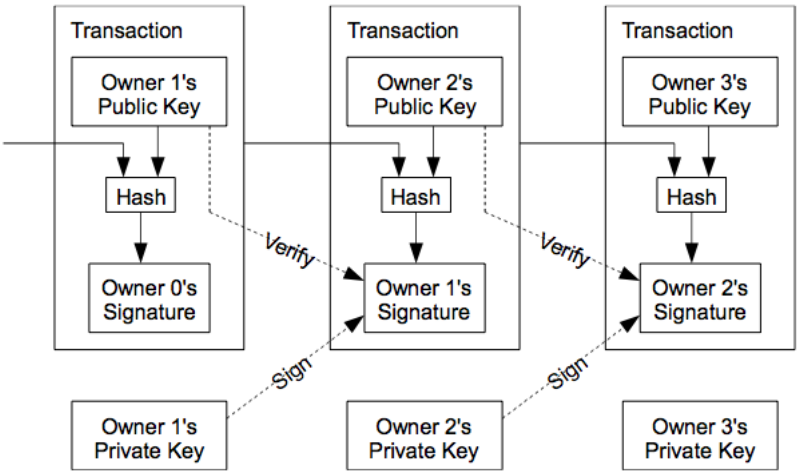

Figure 2.5: Bitcoin Transaction Overview

transaction is created by the owner of the Bitcoin address. An *output* has two parts: one is **value**, which is the number of Satoshis that are to be transferred, the other is **scriptPubKey**, specifying Bitcoin addresses of the one of more payees.

Satoshi gave a simplified description of how transaction functions, shown in Figure 2.5 [49]. Considering the middle transaction from owner 1 to owner 2, owner 1 uses his private key sign over the hash value of previous transaction together with owner 2's public key to create owner 1's signature. The signature can be verified using owner 1's public key. Once validated, the transaction is confirmed and put into the block.

Mining

Bitcoin mining has two main purposes, one is to create new Bitcoins, and the other is to make sure that every participant has a consistent view of the transaction log. Since inconsistent log of transaction may give opportunities to attackers to spend the same Bitcoins twice, mining allows verified transactions to be added into a block of transactions, and conflicting and invalid transactions will be ignored to avoid double spending.

To mine a block, miners need to perform computational proof of work using cryptographic hash function SHA256. The block header is hashed by SHA256 twice, which is SHA256 (SHA256 (block header)), to generate a hash value. If the hash value starts with a certain number of zeroes, the block is successfully mined and will be added into the official block chain. Since the

proof of work is hard to perform, trying different nonces contained in the block header to meet such requirements is a rare event, and around every ten minutes, a block is successfully mined [53].

2.1.3 Transaction Malleability

Transaction malleability in Bitcoin has been existed for a long time and did not raise public attention until MtGox, once the largest Bitcoin exchange company in Japan, announced suspending withdrawals because of this attack.

Transaction malleability in Bitcoin system is an attack to change the unique ID of a transaction and later both the original and modified versions are confirmed by the Bitcoin network. Here the unique ID (also called transaction ID) is the cryptographic hash of the entire transaction including the signature described above to identify the transaction. All the information involved is protected by the signature and cannot be modified by attackers. However, because of scripting language that Bitcoin uses, signatures can be modified in some way but still considered as valid. The changes in signature will lead to change of the hash value, that is the transaction ID. If client or exchange software makes Bitcoin transactions by only identifying transaction hashes, it will cause a lot of problems if one transaction has two different hashes [51].

Ken Shirriff described details why it is possible to modify the transaction hashes in his blog [52]. ScriptSig and scriptPubKey use script language which is a stack-based, non-Turning complete language using single byte opcodes [18]. Given both original and modified hashes of a transaction, their corresponding input and output scripts in raw transaction are identical, but there are several differences highlighted in red in the full scriptSig shown in Figure 2.6 [52].

The original transaction scriptSig specifies that 48 bytes of data is pushed onto stack by using opcode **PUSHDATA**, and the modified scriptSig uses **OP_PUSHDATA2** (0x4d) to specify the number of bytes to be pushed is its next two bytes 48 00. That is, both of the two situations did the same thing - pushing 48-byte signature onto the stack to be executed. Therefore, both scriptSigs are considered valid, and they produce different transaction hashes, which makes malleability possible. But in real Bitcoin transactions, using OP_PUSHDATA2 is very rare.

Modifying the push operations in scriptSig is the most common form to carry out malleability attack, besides this method, Pieter Wuille summarized other eight sources of malleability in

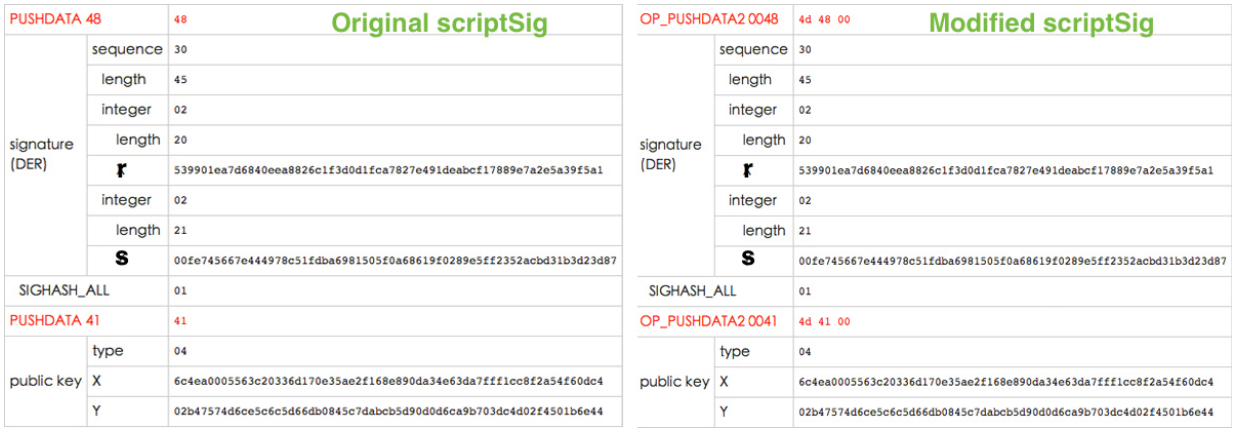

Figure 2.6: Original and Modified scriptSig

Bitcoin together with rules to combat them [59].

2.2 Elliptic Curve Cryptography (ECC)

2.2.1 Mathematical Preliminary

Modular Arithmetic

Modular arithmetic is the arithmetic of congruence first proposed by Friedrich Gauss in 1801 [24]. Numbers in modular arithmetic are operated not on a line, but wrapped around on a circle. A good example is the clock. There are only twelve numbers from 1 to 12 on a clock face, and when time gets to 15 o'clock, it actually shows 3 o'clock.

Modular arithmetic fixes an Integer $N > 1$ called modulus ($N = 12$ of hours, and $N = 60$ of seconds or minutes on a clock), and the value after modulus operation always stays less than the modulus.

The basic operation of modular arithmetic is reduction modulo N. The result of the reduction is the reminder r of dividing a by N given an integer a, denoted by $r \equiv a \ (mod \ N)$. For two integers a and b, if their difference is divisible by N, they are said to be congruent modulo N, denoted by $a \equiv b \ (mod \ N)$. The operations including addition, subtraction, multiplication in modular arithmetic can be performed using integer operations first, and then reduce the result modulo N.

Group

A **group** $(G, \circ)$ is a set G with a binary operation $\circ$ (multiplication/$\cdot$ or addition/$+$) that operates on two elements in G to generate the third element also within set G satisfying associativity, identity, and invertibility conditions.

Associativity is that given three elements a,b,c in set G, the equation $a \circ (b \circ c) = (a \circ b) \circ c$ holds. Identity is that there is an element e in set G such that all elements in G satisfies $e \circ a = a \circ e = a$. And invertibility is for any element a in G there is another element b in G such that $a \circ b = b \circ a = e$.

A **finite group** is a group containing a finite number of elements, and the number is called **order** (cardinality). Within a finite group G, if a non-empty set $S \in G$, then S is a **subgroup** of G, and the identity element of G should also be in the subgroup S.

If a group is generated by one single element g, it is called **cyclic group**, while g is **generator** of the group. All elements are generated by repeatedly applying the binary operation or inverse to g. Every cyclic group is an abelian group since the operation in cyclic group is communicative.

Ring and Finite Field

In mathematics, a **ring** is a set of elements together with addition and multiplication operations satisfying the following properties:

- The ring under addition operation is an abelian group.
- The ring under multiplication operation is closed and associative.
- For all a,b,c in the ring, the equation $a \cdot (b + c) = a \cdot b + a \cdot c$ holds.

A **field** is a ring that the nonzero elements form an abelian group under multiplication operation. If a field has a finite number of elements, it is called a **finite field**. It only exists when the order of the field is a prime power $q = p^k$(p is a prime number, k is an integer larger than zero). For each prime power q, there is exactly one up to an isomorphism finite field whose order is q, denoted by $GF(p)$ or F_q.

In elliptic curve cryptography, two kinds of finite fields are used. One is F_p of p elements where p is a prime, the other is F_{2^m} with 2^m elements.

2.2.2 ECC

Public Key Cryptography

Public key cryptography is cryptography that uses a key pair - a public key and a private key to encrypt and decrypt messages. These two keys are mathematically linked and the public key is known to everybody to encrypt messages, while private key is only known to the recipient for decryption. Besides encrypting messages, public key cryptography is also used in message authentication. The private key is used to process a message to produce a digital signature, and every one who has the corresponding public key can verify the validity of this signature.

Public key algorithms are based mathematical problems such as integer factorization and discrete logarithm problem which makes it easy to generate a public-private key pair but computational infeasible to get the private key from the public one.

ECC is one of the public key cryptosystems based on elliptic curves over finite fields introduced by Neal Koblitz [33] and Victor Miller [40] in 1985, and the following sections concentrate on introducing this algorithm.

Elliptic Curves Over Prime Fields

Since Bitcoin system uses elliptic curve over prime fields F_p, this section will introduce some basics about elliptic curves on over finite field F_p, where p is a prime number greater than 3.

The elliptic curve E over F_p can be expressed by the Weierstra equation: $y^2 = x^3+ax+b\ (mod\ p)$ where $a, b \in F_p$ and $4a^2 + 27b^2 \neq 0$. All the points satisfying the equation together with the identity element $\mathcal{O}$ (point of infinity) form group.

The domain parameters on the curve over F_p are a sextuple, expressed as $T = \{p, a, b, G, n, h\}$, where the integer p specifying the finite field F_p, a,b are constants defining the equation, G is the base point on the curve, prime n is G's order, and integer h is its the cofactor. Different curves will have different domain parameters to form different elliptic curve groups used in cryptography.

Point Operations

The binary operation of point on the curve is normally denoted as an addition operation, and addition of points on curve has geometric interpretation, shown in Figure 2.7 [31].

Given two points on the curve $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, a line trough P and Q will intersect the curve with a third point, then the addition of these given two points $R = (x_3, y_3)$ is the third point's symmetric point of x axis.

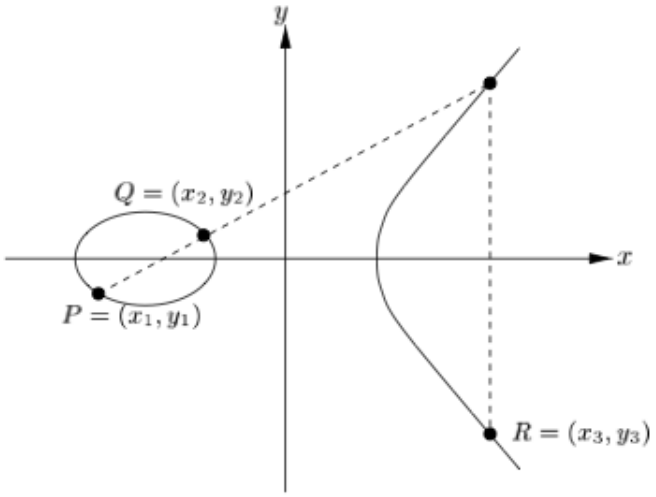

Figure 2.7: Point Addition on Elliptic Curve

The double of a point $P = (x_1, y_1)$, here we call point doubling, is described in Figure 2.8[31]. To get the point doubling result $R = (x_3, y_3)$, first draw a tangent line at the point P, and this line intersects with the curve at the second point, then the point R is the second point's symmetric point of x axis.

Similarly, the multiple of a point $P = kQ$ also called point multiplication is just repeated point additions and point doublings. However given a point P on the curve and its multiple Q, it is difficult to find a unique integer k such that $P = kQ$. This is called **Elliptic Curve Discrete Logarithm Problem** (ECDLP), which is a special case of Discrete Logarithm Problem. The security of elliptic curve cryptography relies on ECDLP. If the ECDLP is computationally intractable, the elliptic curve cryptography is considered to be cryptographically strong [9].

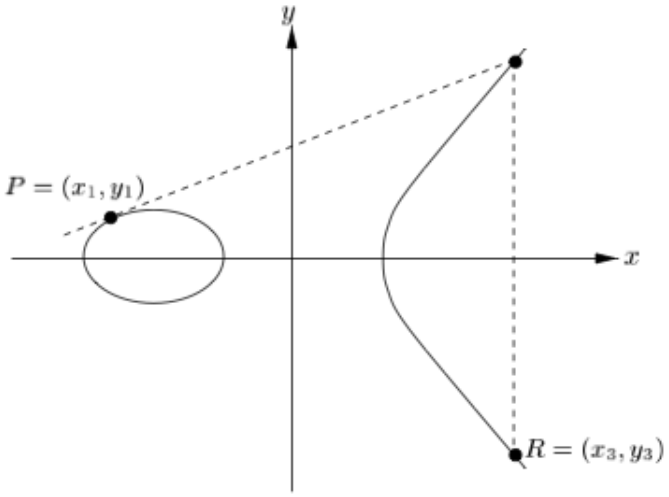

Figure 2.8: Point Doubling on Elliptic Curve

ECC Algorithm

Same with other public key cryptosystems, there are three algorithms in Elliptic Curve Cryptography, shown in Table 2.2.

Key Generation	Private key: a random number d in the range of $[1, n-1]$ Public key: $Q = dP$ where P is a point on the curve
Encryption	Given a message m, and its corresponding point M on the curve. The *Ciphertext* $c = (c_1, c_2)$, $c_1 = kP$, $c_2 = M + kQ$, where k is chosen randomly in the range of $[1, n-1]$
Decryption	*Plaintext* is $M = c_2 - dc_1$

Table 2.2: ECC Algorithms

2.3 Side Channel Attacks

Side channel attack was first introduced to cryptographic community by P. Kocher in 1996 [34]. The first side channel attack exploited is timing attack, and later Kocher et al. proposed differential power analysis (DPA) and simple power analysis (SPA) [36]. This kind of attack is to explore the information from physical implementation of a system, rather than weaknesses of cryptographic algorithms themselves. For example, instead of seeing the ciphertext or both

ciphertext and plaintext to crack the keys, side channel attacks concentrate on additional inputs or outputs of the device such as timing information, power consumption, and electromagnetic radiation.

Timing information refers to the time performing one operation. For different cryptosystems, it takes different amount of time to process different inputs. Attackers can collect enough timing information required to perform private key operations to break a cryptosystem.

Power consumption attacks, similar to timing attacks, are to measure power consumption during the encryption. The obtained power consumption can be combined with other cryptanalysis techniques to crack the private key of the system. Power consumption attack can be roughly divided into simple power analysis (SPA) and differential power analysis (DPA). SPA simply interprets power consumption into visual representation during the operation of a device or system, and such information may leak important information about the system. Many cryptographic primitives are tested to be vulnerable under such attacks, but they could be avoided by some countermeasures [2] discussed later. DPA needs not only visual information but also extra more statistical analysis such as data dependencies on power consumption to crack the system, and compared with SPA, it is more complex and harder to prevent.

Besides passive side channel attacks, there are also active side channel attacks and one example is fault analysis. Fault analysis (FA) is based on generating faults to the system to extract keys. Faults can be introduced by changing voltage, tampering with the clock, changing the temperature and so on. To identify such faults to carry out attacks, the same message is encrypted twice and the results are compared.

Although there are a lot of ways to conduct side channel attacks, several basic, high-level countermeasures have been proposed to fight against them. The countermeasures can be divided into two categorizes. One is to avoid or reduce leaking such information; the other is to eliminate the relationship between the information and secret data.

For timing attacks, adding delays on the operations seems to be the simplest proposal, but the real implementation may bring some difficulties. If all operations are made to perform using the same amount of time, it will reduce the efficiency of the system. Adding random delays on operations can reduce the possibilities to crack the system, but it is still possible to attack it by collecting more data pieces.

As for power consumption attacks, a method of preventing SPA and DPA is power consumption

balancing. This method requires dummy registers to be installed to make sure the total power consumption is balanced. This method changes the power consumption into a constant and eliminates data dependency between inputs and private keys. Another common methods is to add noise to power consumption measurements. Similar with adding delays on timing attacks, adding noise can also increase the number of samples required to crack the system.

To prevent differential fault attacks, just run every encryption twice to check the two results, output the encryption result only if they are the same. However this method will cost more computation time for the system.

2.4 Related Work

2.4.1 Alternative of Public Key Cryptography in Bitcoin

In 2013, Vitalik Buterin adopted Lamport Signatures to replace elliptic curve cryptography used in Bitcoin system [10]. This idea is inspired by assuming the existence of quantum computers. Shor's algorithm is useful in factoring numbers in quantum computers, and the modified version can reduce the runtime to crack elliptic curve cryptography significantly. In a Bitcoin transaction, the public key is exposed in a used Bitcoin address, which makes elliptic curve cryptography easy to crack. Therefore, the Lamport's one time signature can be used to construct the public key to replace the one in elliptic curve cryptography. Construction of Bitcoin addresses still remains unchanged, but the public key used consists of 320 RIPEMD-160 hashes instead of the elliptic curve point.

Joseph Binneau and Andrew Miller proposed another cryptocurrency called Fawkescoin in 2014 [6]. It utilizes Bitcoin-like mechanisms but for transactions, Bitcoin's ECDSA signature is replaced by using hash-based Guy Fawkes signature, which is similar to Buterin's method. However, Lamport's signature scheme will produce signatures with thousands of bits, and is unwieldy in Bitcoin system.

Binneau and Miller's Guy Fawkes signature based proposal constructs secure signatures using only a hash function and a secure timeline service which make Fawkescoin more efficient. The security of the transaction relies on the fact that the message transferred is made public before x is published, where x is a random, secure value only known by the owner to compute the hash value $H(x)$ as the address.

Up to now, there are no obvious flaws of ECDSA used in Bitcoin protocol. The replacement of public key cryptography shows it is possible to construct Bitcoin like cryptocurrency before public key cryptography is invented. And if public key algorithms in Bitcoin protocol compromised, such backup plans can be used as a response to the security issues.

2.4.2 Performance Improvement on ECC/ECDSA

Improving the performance of ECDSA is some sense similar with improving the efficiency of elliptic curve cryptography, since the point multiplication is the most time-consuming operation.

To calculate the point multiplication, the simplest method is called double-and-add using binary representation. Generalizations of this binary method such as k-ary method, signed window method could be used to compute the point multiplication of elliptic curves over a finite field [38]. These algorithms process a block of digits simultaneously and needs a few precomputations based on the size of the block. Gordon detailed some fast exponentiation methods that could be used in ECC in [25]. Itoh et al. [29] proposed formulas for computing repeated doublings in projective coordinates that are compatible with window method, so that the number of field multiplications and field additions are both reduced. Recently, Emilia Kasper presented a method to improve the performance of elliptic curve library in openSSL [32]. The author chose *secp224r1* curve to be used in the 64-bit implementation. To reduce cost during the point operation, standard precomputation technique is used to compute multiples of a given point before the scalar point multiplication is performed, and the final total number of point doubling and point addition in point multiplication is cost down. Also, the author performed interleaved multiplication by precomputing linear combinations for a fix point G to reduce doubling and addition.

2.4.3 Side Channel Attacks and Countermeasures in ECC/ECDSA

The goal of side channel attacks on ECC is to get the secret scalar which is similar to attacks on ECDSA in Bitcoin. And there are quite a lot different ways to fight against such attacks.

Montgomery's method introduced in 1987 [41] is said to be resistant to timing attacks and simple power analysis [28, 44], but Brumley and Tuveri proposed a remote timing attack in OpenSSL's Montgomery ladder implementation of ECDSA using elliptic curves over binary

fields [8]. Luther Martin did not think it is a serious attack since it is a rare situation which only works on ECDSA used in OpenSSL' TLS over binary fields [37] . Since ECDSA in Bitcoin uses curves over prime fields, this attack can be ignored. However, after FLUSH+RELAOD attack proposed, Yarom and Falkner recovered ECDSA secret scalar in OpenSSL with Montgomery ladder implementation using the attack. By getting the scalar, attackers could forge an unlimited number of signatures. The authors suggested that Montgomery ladder method should be avoided in the implementation of elliptic curve protocols in OpenSSL since it is no longer secure at all.

Another way to protect against simple side channel attacks similar to Montgomery ladder called double-and-add-always is proposed by Coron in 1999 [16], and Coron mounted a differential power attack on elliptic curve cryptography and proposed three countermeasures against it: randomizing the private scalar, blinding the point P and randomizing projective coordinates.

But presented in [22], randomizing private scalar is vulnerable under carry-based attack, and [23] shows both scalar randomization and point blinding are not resistant to the doubling attack. But fortunately a method called random key splitting [12] makes elliptic curve cryptography resistant to doubling attack by splitting the secret scalar into two random values.

As for fault analysis, coherence check described by Dominguez Oviedo A. [19] could be used in Montgomery ladder to withstand differential fault analysis. What is more, Montgomery ladder is resistant to C safe-error attack which is also a kind of fault analysis in ECC [41]. Combined curve check proposed by Blomer J. et al. [5] could withstand the sign change fault attacks by introducing a reference curve to detect faults. Ciet and Joye [13] also introduce two methods point validation and curve integrity check. Point validation checks whether a certain point is on the curve to protect against invalid point attacks. While curve integrity check detects fault injections on curve parameters when point multiplication is calculated.

Putting side channel attack into Bitcoin protocol, Benger et al. proposed an attack to recover private key from OpenSSL ECDSA algorithm [4]. The analysis is based on the curve *secp256k1* used in Bitcoin system. The authors combined the FLUSH+RELOAD attack proposed by Yarom and Falkner [60] and lattice techniques. And they were able to recover private keys in elliptic curve with high probability using less than 256 signatures.

This attack requires obtaining several signatures for a designate private key, so one of the mitigation is to limit the number a private key used. Bitcoin system also uses *secp256k1*,

and it recommends a new private key should be used for each transaction. However, this recommendation is not always followed [47]. Therefore, Bitcoin transaction can also be affected by this attack, and this side channel attack could give guidance to avoid it when implementing Bitcoin ECDSA algorithms.

2.4.4 Deterministic ECDSA

In 2010, the ECDSA private keys of Sony's PlayStation 3 were leaked because of the bad random number generator they used [21]. The random scalar generated in signing process was later proved to be a constant number. And recently, some flaws were found in Android devices when generating random values, leading to Bitcoin lost for devices running Bitcoin software [58].

These two security incidents in ECDSA are all due to bad number generators used. To avoid using such random number generators, a deterministic ECDSA could increase the security of the implementation. The deterministic ECDSA is proposed by Pornin in RFC 6979 [46]. Deterministic here means, instead of selecting a random scalar k in signing process, k is fixed with the same message and private key during signature generation but it is indistinguishable with random generated ones. The generation of k uses the hash of the message $h(m)$ and private key as input to a deterministic pseudorandom number generator HMAC-DRBG, and output of the generation is used to yield k. The detailed algorithm is described in [46]. The security can be assured as long as HMAC behaves as a pseudorandom function.

Chapter 3

Algorithm Specification

This chapter intends to present the preparation of related parameters and algorithms before the implementation of ECDSA and Bitcoin transaction as well as its performance and security improvement.

Firstly, the core of the system implementation is to know the parameters and algorithms of how prototype ECDSA is defined, and what curve is used in Bitcoin. The subsequent implementation is based on it. Secondly, since transaction of Bitcoin is based on digital signatures to prove Bitcoin ownership, the processes of Bitcoin transaction hashing and signing should also be made clear. Thirdly, algorithms of most efficient version of ECDSA combining projective coordinates and window method are listed. At last, to add more security countermeasures against side channel attacks, the algorithms of defending against, for example, simple power analysis, differential power analysis, and fault analysis that are compatible with the designed ECDSA are summarized.

3.1 ECDSA Parameters and Algorithms

3.1.1 Curve

The Bitcoin system uses ECDSA by adopting the curve 256-bit domain parameters *secp256k1*. Its Weierstra expression is $y^2 = x^3 + 7$, and values of the corresponding domain parameters are shown in Table 3.1 [50] in hexadecimal format.

Parameter	Value
p	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFC2F
a	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
b	00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007
G	02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
n	FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141
h	01

Table 3.1: Parameters of *secp256k1*

3.1.2 Point Operation Parameters and Algorithms

In Section 2.2.2, binary operations to find a point on elliptic curves over finite field F_p is geometrically described. Since calculation of point multiplication is based on point addition and point doubling, given two distinct points to add or one point to double, the algebraic expressions are defined as follows in Equation 3.1 and Equation 3.2:

- Point addition: $R(x_3, y_3) = P(x_1, y_1) + Q(x_2, y_2)$

$$\begin{cases} x_3 &= (\lambda^2 - x_1 - x_2) \ mod \ p \\ y_3 &= (\lambda(x_1 - x_3) - y_1) \ mod \ p \end{cases} \tag{3.1}$$

  where $\lambda = (y_2 - y_1/x_2 - x_1) \ mod \ p$

- Point doubling: $R(x_3, y_3) = 2P = P(x_1, y_1) + P(x_1, y_1)$

$$\begin{cases} x_3 &= (\lambda^2 - 2x_1) \ mod \ p \\ y_3 &= (\lambda(x_1 - x_3) - y_1) \ mod \ p \end{cases} \tag{3.2}$$

  where $\lambda = ((3x_1^2 + a)/2y_1) \ mod \ p$

With the parameters of point addition and doubling, point multiplication can be calculated, and the most basic method implementing it is double-and-add algorithm in Algorithm 1.

.

Algorithm 1 Double-and-add Algorithm

Require: d,P

Ensure: $Q = dP$

1: **if** $d_1 = 1$ **then**
2: $Q := P$
3: **end if**
4: **for** $i = 1$ to n **do**
5: $Q := 2Q$
6: **if** $d_i = 1$ **then**
7: $Q := Q + P$
8: **end if**
9: **end for**
10: **return** Q

There are many different implementations of point multiplication algorithm for different purposes, and double-and-add method can be replaced by other methods to improve the performance of ECDSA algorithm or protect against some types of side channel attacks.

3.1.3 ECDSA Algorithms

The implementation of ECDSA in Bitcoin system should follow the processes of key generation, signing and verification algorithms strictly. In ECDSA, the signature generation and verification is similar to DSA, but the key generation is based on ECC algorithm.

Key Pair Generation

The key pair in ECDSA is generated based on the domain parameters, and in Bitcoin system, the domain parameters are listed in Section 3.1.1. Given the elliptic curve E over Z_p with number of points that is divisible by the large prime n,

1. Choose a point $P(x_p, y_p)$ on the curve and a random integer $d \in [1, n-1]$.
2. Compute $Q(x_q, y_q) = dP$, make sure the point Q is also on the curve.
3. Public key is (E, P, n, Q), and private key is d.

Signature Generation

Given a message m to be signed and the private key d,

1. Choose a random integer $k \in [1, n-1]$.
2. Compute $(x_1, y_1) = kP$, convert x_1 into integer and $r = x_1 \ mod \ n$. (Return to step 1 if $r = 0$)
3. Compute $s = k^{-1}(SHA1(m) + dr)$. (Return to step 1 if $s = 0$)
4. Signature is (r, s) pair.

Signature Verification

Given the signature pair (r, s) on message m and public key (E, P, n, Q),

1. Check that integers $r, s \in [1, n-1]$.
2. Compute $w = s^{-1} \ mod \ n$.
3. Compute $u_1 = SHA1(m)w \ mod \ n$, $u_2 = rw \ mod \ n$.
4. Compute $(x_0, y_0) = u_1P + u_2Q$, convert x_0 into integer and $v = x_0 \ mod \ n$.
5. Compare v and r, accept the signature only if $v = r$.

3.2 Bitcoin Transaction Process

3.2.1 Transaction Signing

The process of signing and verifying transactions in Bitcoin system is quite complex. Runeks [48] and Ken Shirriff [54] described the transaction process in very detail by providing examples. With these examples, given a previous raw transaction to redeem Bitcoins from its outputs, the new transaction can be constructed manually. And the process can be divided into two forms: unsigned transaction and signed transaction.

The new unsigned transaction

Just as described in Section 2.1.2, the raw transaction contains inputs including previous transaction hash, index and scriptSig as well as outputs including values and scriptPubKey. To redeem Bitcoins, all elements in new transaction are derived from the previous raw transaction. The unsigned transaction which will be used to generate the signed one containing 13 elements (only suitable for one input and one output) shown in Table 3.2.

Index	Unsigned tx element	Length (bytes)
1	Version field	4
2	Number of inputs	1
3	**Previous transaction hash**	32
4	Output index to redeem	4
5	Length of scriptSig	1
6	**scriptPubKey to be redeemed**	25
7	Sequence field	4
8	Number of outputs	1
9	Amount to be redeemed	8
10	Length of actual scriptPubKey	1
11	**Actual scriptPubKey**	25
12	Lock time field	4
13	Hash code type	4

Table 3.2: Elements in an Unsigned Transaction

The third element *previous transaction hash* is generated by double hashing all elements involved in previous transaction using SHA256 algorithm. Since in current transaction the data has not been signed, the sixth element, instead of being scriptSig, is the scriptPubKey of the previous transaction to be redeemed. A thing need to mention is, *scriptPubKey to be redeemed* only appears once only if there are more outputs.

The new Signed transaction

The unsigned transaction is easy to obtain given the previous raw transaction. However, the transformation from unsigned to signed transaction is quite complex. The final scriptSig is a

main part in signed transction, and it is generated by, firstly create a public/private ECDSA key pair and double hash the new unsigned transaction using SHA256. Next generate the signature by using the private key to sign on the hash. Then the final scriptSig will be a concatenating of length of signature, signature, one byte hash code type, length of public key and public key.

Compared with unsigned transaction, there are only three elements changed in the signed transaction highlighted in red in Table 3.3. The new generated *final scriptSig* will replace the sixth element *scriptPubKey to be redeemed*, and its corresponding length will be changed as well. Different with *scriptPubKey to be redeemed* for multiple outputs, *final scriptSig* should appear in every output. In addition, the last element *hash code type* in unsigned transaction will be removed.

Index	Signed tx element	Length (bytes)
1	Version field	4
2	Number of inputs	1
3	Previous transaction hash	32
4	Output index to redeem	4
5	**Length of final scriptSig**	1
6	**Final scriptSig**	**Normally 138**
7	Sequence field	4
8	Number of outputs	1
9	Amount to be redeemed	8
10	Length of actual scriptPubKey	1
11	Actual scriptPubKey	25
12	Lock time field	4
13	**N/A**	**N/A**

Table 3.3: Elements in a Signed Transaction

3.2.2 Transaction Verification

The verification process of Bitcoin is quite simple compared with the signing process. It just uses the ECDSA algorithm mainly to verify that the inputs of the new transaction are authorized to receive the specified values from the previous transaction's referenced outputs. There are three parameters needed during verification process, the signature and public key are contained in

scriptSig, and the original message needed to be verified is double hash value of the unsigned transaction.

3.3 Performance Improvement

According to our investigation, the most efficient version of ECDSA algorithm uses projective coordinates with the window method for point multiplication, and detailed steps and measurements are in my cooperator's thesis.

Equation 3.1 and 3.2 are addition and doubling formulas used in affine coordinate system, but inversions (given x to find y such that $xy = 1$ where $x, y \; in \; F_p$) cause point addition, doubling and point multiplication to be highly inefficient [27]. To avoid inversion operations, another coordinate system without using inversions in point addition and doubling is expected to be used. Projective coordinate system is such a system to represent points over prime elliptic curve $y^2 = x^3 + ax + b$. It uses a triple (X, Y, Z) to represent a point in affine coordinate $(x, y) = (X/Z, Y/Z)$. There are many different ways for point addition and doubling with different operation counts, and the formulas chosen to be used are displayed in Appendix A. The implementation should follow the algorithms strictly.

Window method for point multiplication could compute some of the points in advance to reduce the number of computations. The algorithm is shown in Algorithm 2.

In this algorithm, a new parameter w is introduced representing the window size and 2^w values of dP will be precomputed where $d = 0, 1, 2 \ldots 2^w - 1$. Normally the best choice of value w is 4 for curves recommended by NIST [17]. The process of window method is similar to double-and-add algorithm, but it is more efficient because there are fewer point additions by performing precomputations.

.

Algorithm 2 Window Method Algorithm

Require: d,P

Ensure: $Q = dP$

1: **if** $d_1 > 0$ **then**
2: $Q := d_1 P$
3: **end if**
4: **for** $i = 1$ to n **do**
5: $Q := 2^w Q$ (Repeated doubling)
6: **if** $d_i > 0$ **then**
7: $Q := Q + d_i P$(Precomputed value $d_i P$)
8: **end if**
9: **end for**
10: **return** Q

3.4 Side Channel Countermeasures

Up to now, there are no practical methods of timing attacks on ECDSA algorithm in Bitcoin protocol, but the elliptic curve cryptography is vulnerable under some basic side channel attacks such as simple power analysis, differential power analysis and fault analysis. The implementation of ECDSA in Bitcoin is expected to be against such basic attacks. Followings are some of countermeasures against side channels that are suitable for implementation.

3.4.1 Simple Power Analysis

For simple power analysis, there are three methods suitable for our implementation: double-and-add-always, Montgomery ladder and unified operation.

Double-and-add-always. It is an improvement of double-and-add method in Algorithm 1 by adding dummy point additions. For each bit of the private scalar, both addition and doubling are performed so that operations are independent from the value of scalar.

.

Algorithm 3 Double-and-add-always Algorithm

Require: d,P

Ensure: $Q = dP$

1: $Q[0] = \mathcal{O}$
2: $Q[1] = \mathcal{O}$
3: **if** $d_1 = 1$ **then**
4: $\quad Q[0] := P$
5: **end if**
6: **for** $i = 1$ to n **do**
7: $\quad Q[0] := 2Q[0]$
8: $\quad Q[1] := Q[0] + P$
9: $\quad Q[0] := 2Q[d_i]$
10: **end for**
11: **return** $Q[0]$

Montgomery ladder. This method shown in Algorithm 4 performs point multiplication by using the form of double-and-add method but it computes the same number of point doubling and additions in a fixed pattern regardless of the secret scalar and no dummy operations involved.

Unified operation. This method unifies the doubling and addition into one operation to eliminate difference between point doubling and addition. Brier and Joye [7] proposed one single formula that suitable for both operations so that point doubling is not needed any more, the unified formula is shown in Appendix B.

.

Algorithm 4 Montgomery Ladder Algorithm

Require: d,P

Ensure: $Q = dP$

1: $Q[0] = \mathcal{O}$
2: $Q[1] = \mathcal{O}$
3: **if** $d_1 = 1$ **then**
4: $\quad Q[0] := P$
5: $\quad Q[1] := 2P$
6: **end if**
7: **for** $i = 1$ to n **do**
8: $\quad Q[1 - d_i] := Q[0] + Q[1]$
9: $\quad Q[d_i] := 2Q[d_i]$
10: **end for**
11: **return** $Q[0]$

3.4.2 Differential Power Analysis

To avoid techniques statistically analyzing the secret scalar, several basic countermeasures are proposed against differential power analysis, for example, scalar blinding, base point blinding and random scalar splitting.

Scalar blinding chooses a random number k of n bits when computing $Q = dP$, then compute $d' = d + k * \#N$ where N refers to the number of points on the curve, then $Q = d'P = (d + k * \#N)P = dP$ since $\#NP = \mathcal{O}$.

Base point blinding is similar to scalar blinding and it blinds the base point P by computing $Q' = d(P + R)$ while the value of dR is stored secretly.

Random scalar splitting is to split the scalar into two random k_1 and k_2 where $k = k_1 + k_2$ so that every execution the scalar is random. Another splitting is to choose a random r, so $k = \lfloor k/r \rfloor + (k \ mod \ r)$.

3.4.3 Fault Analysis

According to the review of fault analysis countermeasures in section 2.4.3, some of the methods do not fit into our ECDSA implementation, for example, point validation checks whether point to be multiplied lies on the curve or not. However in ECDSA, the base point is constant and always lies on the curve, so this method does not applicable.

Double-and-add-always is not immune to a C safe-error (computational safe-error) introduced by Yen and Kim et al. [57] because when inducing errors in operation $Q[1] := Q[0] + P$ of Algorithm 3 there is no difference if the scalar bit b_i is zero. In such a way the scalar can be easily obtained. However Montgomery ladder method is resistant to this attack because there are no dummy operations involved so that any errors induced will cause an incorrect result.

Coherence Check. Another countermeasure against differential fault analysis called Coherence Check [19] is effective only when Montgomery ladder is used. This is due to the fact that the difference between $Q[0]$ and $Q[1]$ is always P in Algorithm 4. If the difference value is checked during and after the point multiplication, whether errors induced or not can be detected.

Chapter 4

Implementation

This chapter covers the details of Java implementation based on algorithms specified in Chapter 3. Java methods of ECDSA prototype using affine coordinates and improved version using projective coordinates are summarized, and modified code pieces achieving security goals against side channel attacks are presented.

4.1 Programming Language and Environment

The implementation environment is quite simple without complex setups. The system used to implement the prototype of ECDSA algorithm and Bitcoin transaction is MAC OS X version 10.9.4, the programming language is Java [42], and the programming tool is Eclipse Standard 4.4 for MAC OS X 64-bit [56], and what is more, the implementation imports an external jar file Bouncy Castle 1.5.0 [11] in Java which is a collection of APIs especially for cryptography.

4.2 ECDSA Prototype

The implementation of ECDSA prototype algorithm has three classes: PointMultiplication, GenerateKey and ECDSA. PointMultiplication class mainly performs the point addition, point doubling and point multiplication. GeneratKey is a class generating public/private key pair, while ECDSA class is to sign the message and verify the generated signature. Table 4.1 shows the main packages used in the implementation of these three classes with short descriptions.

Class	Package Imported	Package Imported
Common Packages	java.math.BigInteger	Provides classes for arithmetic operations of very large numbers
Common Packages	java.security.spec.EllipticCurve	A class containing necessary values to represent an elliptic curve
Common Packages	java.security.spec.ECPoint	A class specifying a point on the elliptic curve in affine coordinate representations
Common Packages	java.security.SecureRandom	Used to generate cryptographically strong random numbers
GenerateKey class	java.security.KeyPair	A simple holder for a public and private key pair
GenerateKey class	java.security.KeyPairGenerator	A class used to generate a pair of public key and private key
GenerateKey class	java.security.PrivateKey java.security.PublicKey	Used for representing a private key and a public key respectively
GenerateKey class	java.security.interfaces.ECPrivateKey	An interface generating signatures on messages using ECDSA
GenerateKey class	java.security.interfaces.ECPublicKey	An interface verifying signatures on signed messages using ECDSA
GenerateKey class	java.security.spec.ECFieldFp	A class defining an elliptic curve over prime finite field
GenerateKey class	java.security.spec.ECParameterSpec	A class specifying domain parameters used in ECC
GenerateKey class	org.bouncycastle.jce.ECPointUtil	A bouncycastle utility class used for elliptic curve decoding
GenerateKey class	org.bouncycastle.util.encoders.Hex	A bouncycastle class used for convert hexadecimal strings
ECDSA class	java.security.MessageDigest	Provides the functionality of hash algorithm such as SHA1 and SHA256

Table 4.1: Descriptions of Imported Packages

4.2.1 Point Multiplication (PointMultiplication class)

PointMultiplication class contains methods of calculating point multiplication that will be used in both ECDSA signature signing and verification. There are three methods: *AddPoint ()*, *DoublePoint ()*, and *ScalarMulti ()*.

The *AddPoint ()* and *DoublePoint ()* methods are implemented using the equations in Section 3.1.2 to calculate the new point. Before performing the addition operation in *AddPoint ()* method, equality of two input points P and Q will be checked. If they are equal, *DoublePoint ()* method is called. The equality of the two points and Infinity point are checked respectively as well. If one of the points, for example P is a point at infinity, the *AddPoint ()* will return the value of Q directly without performing the addition equations.

ScalarMulti () method is implemented using double-and-add method by calling *AddPoint ()* and *DoublePoint ()*. This method will be changed later for performance and security considerations.

Return type	Method	Explanation
BigInteger	*SelectK ()*	This method is to generate a secret random 256-bit value k of type BigInteger within the range of $[1, n-1]$,where the prime n is the order of the base point G. If the generate value k is not coprime with p defining the field, regenerate a new one.This method returns the random k .
BigInteger	*SHA1 (byte [] m)*	This method generates the hash value of a message m by using SHA1 algorithm. This method returns hash value with type BigInteger of the input message m.
BigInteger[]	*Sign (byte [] m, BigInteger privatekey)*	This method is implemented following the algorithm outlined in Section 3.1.3 signature generation. The message to be signed and the private key (generated using *KeyGeneration ()*) are used to produce the signature. The method calls *SelectK ()*, *SHA ()* and *ScalarMulti ()*to generate a random value k, SHA1 hash value of the message and point multiplication result respectively. All the other operations are done by using related Java BigInteger classes. This method returns an array of type BigInteger signature .
void	*Verify (byte [] m, BigInteger [] signature, ECPoint publickey)*	This method is implemented following the algorithm outlined in Section 3.1.3 signature verification. Given the original message m, the returned signature from *Sign ()* and public key (generated using *KeyGeneration ()*), the signature validity is checked and verified. This method calls *ScalarMulti ()* and *AddPoint ()* to do point multiplication and point addition. All the other operations are done by using Java BigInteger classes. This method outputs the verification result of the generated signature. If signature is valid, output 'Valid signature' and otherwise 'Invalid signature'.

Table 4.2: ECDSA Class Description

4.2.2 Key Generation (GenerateKey class)

GenerateKey class generates a key pair particularly used in Bitcoin transaction. The curve domain parameters in *secp256k1* are initialized to specify the elliptic curve. There is only one method *KeyGeneration ()* returning a key pair array with the value of private key, x coordinate of public key and y coordinate of public key. Instead of following the process of ECDSA key generation, the key pair is generated using the java.security.KeyPairGenerator class.

4.2.3 Signature Generation and Verification (ECDSA class)

Signature generation and verification processes are all performed in ECDSA class. Besides the *Sign ()* and *Verify ()*, there are two more methods *SelectK ()* generating a random number k used in signature generation and *SHA1 ()* computing SHA1 hash value of a message. Table 4.2 gives details about the inputs, outputs or return value of each method.

4.3 Bitcoin Transaction Process

The process of generation Bitcoin transaction follows the sequence in Section 3.2, and the implementation details are presented in my cooperator's thesis.

4.4 ECDSA in Projective Coordinates

ECPoint class in Java API only support points on the curve represented in affine coordinates, so PointMultiplication class is modified to support representation in projective coordinates. The new created class called pointMultiplication is just a rewrite of ECPoint class including point addition, doubling and multiplications operations. Window method is used in point multiplication. Table 4.3 and Table 4.4 are a summary of constructors and methods in pointMultiplication class respectively.

Constructor	**Description**
pointMultiplication (BigInteger X, BigInteger Y, BigInteger Z)	Creates a point represented using a tripe (X, Y, Z) to replace the affine coordinates (x, y) .

Table 4.3: Constructor of pointMultiplication Class

Return type	Method	Description
BigInteger	*getX()*	This method returns the affine x -coordinate , where$x = X/Z$.
BigInteger	*getY()*	This method returns the affine y -coordinate , where$y = Y/Z$.
boolean	*isInfinity()*	This method returns true if the point is point at infinity, otherwise return false.
pointMultiplication	*AddPoint (pointMultiplication a)*	This method is implemented using point addition formula in projective coordinates in Appendix A, and it computes addition of two different points. If two points are equal, *DoublePoint ()* method will be called. This method returns a new point which is addition of two different points, or return to the value of one point if the other point is infinity point.
pointMultiplication	*DoublePoint ()*	This method is implemented using point doubling formula in projective coordinates in Appendix A, and it computes double value of a point. This method returns a new point which is doubling of a given point.
pointMultiplication	*ScalarMulti (BigInteger kin, pointMultiplication point)*	This method is to calculate point multiplication by calling *AddPoint ()* and *DoublePoint ()* methods using window method in Algorithm 2. The window size $w = 4$ and 16 values $0P, 1P, 2P \ldots 15P$ are computed in advance. This method returns a new point which is the point multiplication value given a scalar and a point.

Table 4.4: Method Description of pointMultiplication Class

4.5 Side Channel Countermeasures

There are many methods to fight against side channel attacks, and here four different versions are implemented with different speed performance and security.

4.5.1 Version 1- Double-and-add-always + Scalar Blinding

The first version implements Coron's two methods double-and-add-always together with scalar blinding. The method is Scalar () in Listing 4.1.

The double-and-add-always replaces window method on projective coordinate representation. The modification is simple by adding dummy additions on double-and-add method for every bit of the private scalar. Also, instead of performing point multiplication on the private scalar directly, a 20-bit random value r is chosen to blind the scalar, the value of $r * \#NG = \mathcal{O}$ where $\#N$ is order of base point G, which will not make difference on the final value.

Listing 4.1: Code Extract of Double-and-add-always + Scalar Blinding

```
/**
         * This method computes the point multiplication, it is only used during
         * signing phase.
         */
        public pointMultiplication Scalar(BigInteger kin, pointMultiplication point)
            {
                // choose a secure number r of 20 bits.
                SecureRandom sr = new SecureRandom();
                BigInteger r = new BigInteger(20, sr);
                // computes k = kin+r*N (N is the order of G)
                BigInteger k = kin.add(r.multiply(ECDSA.key.N));
                // following computes k*G=(kin+r*N)*G=kin*G
                String K = k.toString(2);
                pointMultiplication[] q = new pointMultiplication[2];
                // initialize q
                q[0] = new pointMultiplication(zero, zero, null);
                q[1] = new pointMultiplication(zero, zero, null);
                if (K.substring(0, 1).equals("1")) {
                        q[0] = point;
                }
                // double and add always method
                for (int i = 1; i < K.length(); i++) {
                        q[0] = q[0].DoublePoint();
                        q[1] = q[0].AddPoint(point);
                        int di = Integer.parseInt(K.substring(i, i + 1));
                        q[0] = q[di];
                }
                return q[0];
        }
```

4.5.2 Version 2- Window-add-always + Scalar Blinding

The second version implements the so called window-add-always and scalar blinding methods on projective coordinates. The window method and double-and-add-always is combined so that it is possible to perform dummy addition operations, and I call this variant of window method as window-add-always. Same with the previous version against side channel attack, scalar blinding is used as well. This implementation is in Listing 4.2.

Listing 4.2: Code Extract of Window-add-always + Scalar Blinding

```
/**
         * This method computes the point multiplication, it is only used during
         * signing phase.
         */
public pointMultiplication Scalar(BigInteger kin) {
                //select a secure random r of 20 bits
                SecureRandom sr = new SecureRandom();
                BigInteger r = new BigInteger(20, sr);
                // compute K=kin+r*#N(N is the order of G)
                BigInteger K = kin.add(r.multiply(ECDSA.key.N));
                // following computes k*G=(kin+r*N)*G=kin*G
                String k = K.toString(16);
                pointMultiplication[] q = new pointMultiplication[2];
                int a =Integer.parseInt(k.substring(0,1),16);
                if (a>0) {
                        q[0] = precompute[a];
                }
                // for i=1 to n
                for (int i = 1; i < k.length(); i++) {
                        // q=2^4P=16P=DoublePoint(DoublePoint(DoublePoint(DoublePoint
                            (q)))).
                        q[0] = q[0].DoublePoint();
                        q[0] = q[0].DoublePoint();
                        q[0] = q[0].DoublePoint();
                        q[0] = q[0].DoublePoint();
                        int di = Integer.parseInt(k.substring(i, i + 1),16);
                        // for every bit of the scalar, always perform addition
                            operation
                            q[1] = q[0].AddPoint(precompute[di]);
                            if(di>0){
                                    q[0]=q[1];
                            }
                            else
                                    q[0]=q[0];}
```

```
                return q[0];
        }
```

4.5.3 Version 3- Montgomery Ladder + Random Scalar Splitting

The third version of the implementation against side channel attacks, combines Montgomery ladder method in Algorithm 4 and the random scalar splitting method. The Montgomery does not use dummy operations but for every bit of the scalar, fixed pattern operations are performed. As for random scalar splitting a random number rr of 200 bits is chosen to perform the scalar multiplication first $rr * P$, then $(k - rr) * P$ computed again to calculated the final result $Q = rr * P + (k - rr) * P = k * P$.

Listing 4.3: Code Extract of Montgomery Ladder + Random Scalar Splitting

```
public  pointMultiplication ScalarMulti(BigInteger kin,
                        pointMultiplication point) {
                // choose a secure random r
                SecureRandom sr = new SecureRandom();
                BigInteger rr = new BigInteger(200, sr);
                String r = rr.toString(2);
                // compute R = r*point
                pointMultiplication[] R = new pointMultiplication[2];
                R[0] = new pointMultiplication(zero, zero, null);
                R[1] = point;
                for (int i = 0; i < r.length(); i++) {
                        int di = Integer.parseInt(r.substring(i, i + 1));
                        R[1 - di] = R[0].AddPoint(R[1]);
                        R[di] = R[di].DoublePoint();
                }
                // compute b=(k-r)*point
                String b = kin.subtract(rr).toString(2);
                pointMultiplication[] B = new pointMultiplication[2];
                B[0] = new pointMultiplication(zero, zero, null);
                B[1] = point;
                for (int i = 0; i < b.length(); i++) {
                        int di = Integer.parseInt(b.substring(i, i + 1));
                        B[1 - di] = B[0].AddPoint(B[1]);
                        B[di] = B[di].DoublePoint();
                }
                // compute Q = r*point+(k-r)*point=k*point
                pointMultiplication Q = B[0].AddPoint(R[0]);
                return Q;
        }
```

4.5.4 Version 4- Montgomery Ladder + Scalar Blinding

The last version of implementation does not introduce any new methods, it is just a combination of version 3 with Montgomery ladder and version 2 with scalar blinding, with code extract shown in Listing 4.4.

Listing 4.4: Code Extract of Montgomery Ladder + Scalar Blinding

```
public  pointMultiplication ScalarMulti(BigInteger kin,
                        pointMultiplication point) {
                Long startTime = System.currentTimeMillis();
                //select a secure random r of 20 bits
                SecureRandom sr = new SecureRandom();
                BigInteger r = new BigInteger(20, sr);
                // compute K=kin+r*#N(N is the order of G)
                BigInteger K = kin.add(r.multiply(ECDSA.key.N));
                // following computes k*G=(kin+r*N)*G=kin*G
                String k = K.toString(2);

                pointMultiplication[] Q = new pointMultiplication[2];
                Q[0] = new      pointMultiplication(zero, zero,null);
                Q[1] = point;
          // for i=0 to n
          // Montgomery ladder method
                for (int i = 0; i < k.length(); i++) {
                        int di = Integer.parseInt(k.substring(i, i + 1));
                        Q[1 - di] = Q[0].AddPoint( Q[1]);
                        Q[di] = Q[di].DoublePoint();
                }
                return Q[0];

        }
```

4.6 Difficulties

During the implementation of prototype ECDSA, Bitcoin transaction and performance improvement, several problems were encountered because a minor mistake will influence the correctness of the result. Following listed a summary of such problems.

Modular used

One of the keys to successfully implementing ECDSA prototype is to follow the mathematical algorithms. Point operations, signature generation and verification all need to perform modular operations, however for simplification, they are always omitted in the formula. So it takes me a long time to check errors derived from improper modular usage. Also checking if a point on the curve over F_p, the point should satisfy the equation $y^2 = x^3 + ax + b \ (mod \ p)$ while the modular p should be never forgotten.

Double Hash

In the creation of Bitcoin transaction, all elements in previous transaction should be double-hashed using SHA256 and generate a hash used in current transaction. The type of current transaction is String, so I created a method *SHA256 ()* which returns a String type hash value as well. However, the double hash result is incorrect by calling *SHA256 ()* twice. This is because the second time calling *SHA256 ()* is actually hashing the hexadecimal representation of the first hash instead of the binary data that the hex represents. Therefore, I changed the return data type to byte array, and convert the hash value to String after it is double hashed.

Complex Data Type Conversion

When manually creating Bitcoin transaction, the trickiest thing is conversion of data types. For example, the type of ECDSA public key and generated signature are BigInteger, but the generated scriptSig is of type String, therefore the BigInteger type will firstly be converted into type byte array, and then type String, which makes the whole implementation error prone.

Window Method Improvement

Window method contains a step to compute repeated point doublings, and in current implementation, window size 4 decides four point doublings are needed, this is assumed to be a costing part. Many attempts tried to modify the *DoublePoint ()* method so that only one point doubling is used, but unfortunately all attempts failed. And this difficulty is the only unsolved one.

Chapter 5

Results and Evaluation

In this chapter, different implementations on performance are tested under 2.4 GHz Intel Core i7 processor. Since side channel countermeasures will reduce program speed, the balance between speed and security will be measured and evaluated. The following sections cover the testing results and their comparisons.

5.1 Transaction

The implementation uses Runeks' raw transaction (in Appendix C) to produce a new transaction, Figure 5.1 shows the output of the new transaction including unsigned transaction, double hash value of it and signed transaction using a new randomly generate ECDSA key pair. The signature generated from the hash of unsigned transaction is proved to be valid.

5.2 Performance Comparisons

The added countermeasures against side channels are only used in ECDSA signature generation, therefore the performance comparisons only concentrates on signing processes. Besides time of signing, operations are cut into small pieces to be measured. For each test of the implementation, average number and time of 50 operations are measured to get a more precise result.

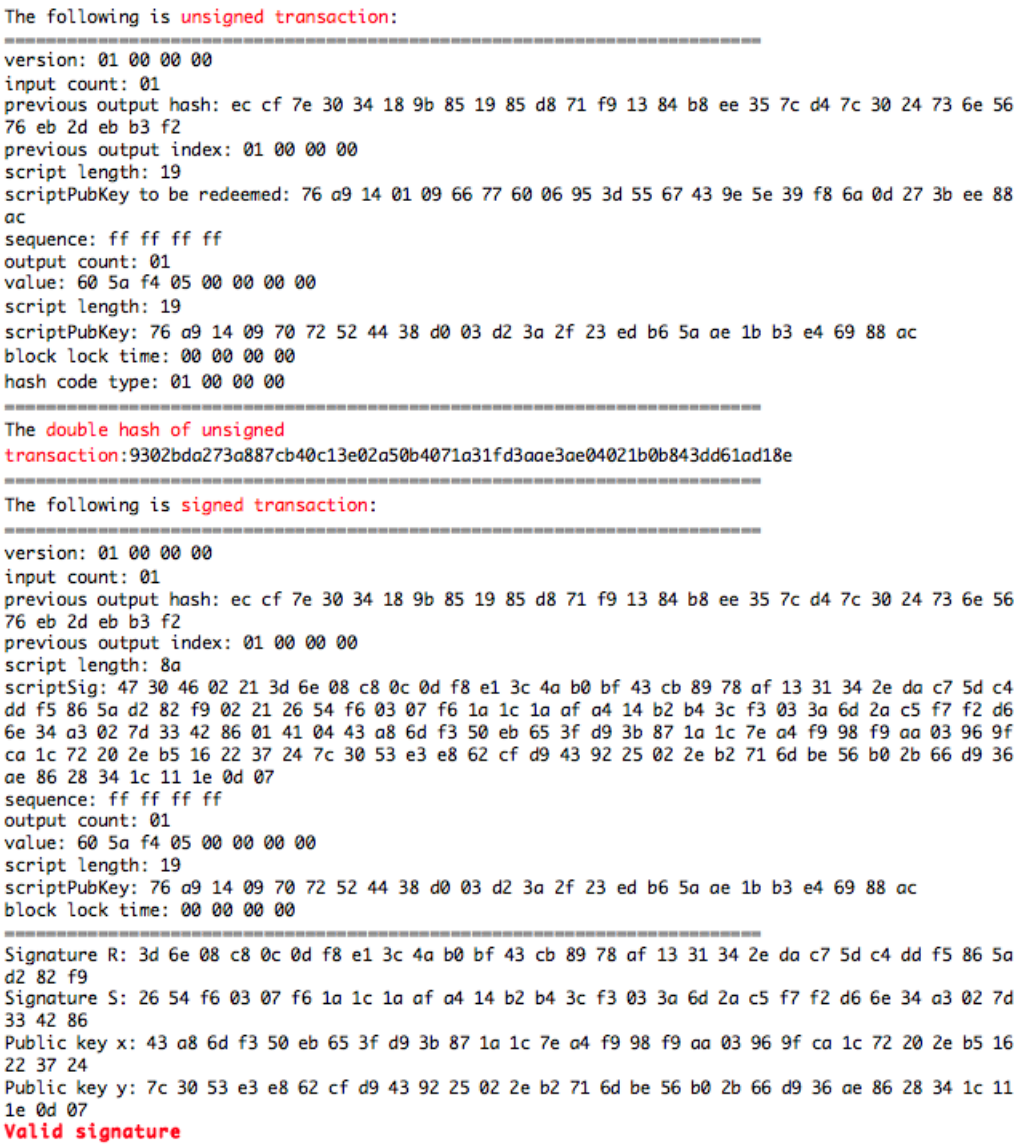

Figure 5.1: Transaction Results

5.2.1 Non-protective Implementation

This thesis presents two non-protective implementations on ECDSA algorithm, one is the prototype using affine coordinates with double-and-add method for point multiplication, and the other is an efficient implementation using projective coordinates with window method.

Table 5.1 shows the number of additions and doublings, average time of single addition, doubling and multiplication together with the signing time. Since time of one addition and one doubling is small, they are measured using microseconds (us), and the time of point multiplication is measured in milliseconds (ms). One point multiplication consists of repeated addition and

doubling operations, so the calculated time of point multiplication $\#ADD * 1ADD + \#DBL * 1DBL \approx 1PM$, and it holds in our implementation. The most time consuming operation in signing is point multiplication, so there is no obvious difference between the time of signing and time of one point multiplication.

	#ADD	1ADD	#DBL	1DBL	Calculated PM	1PM	SIGN
Prototype	127	28 us	256	32 us	11.75 ms	12.64 ms	13.78 ms
Improved	59	17 us	256	15 us	4.81 ms	5.42 ms	6.53 ms

Table 5.1: Operation Counts and Average Time of Prototype and Improved ECDSA

From prototype ECDSA implementation to improved ECDSA represented in projective coordinates, it indeed has significant improvement in point multiplication and signing time, and the detailed analysis as well as progress description is presented in my cooperator's thesis.

Description	# of operations	ADD	DBL
10 more I in ADD	3M+11I	212 us	—
10 more I in DBL	4M+11I	—	213 us
10 more M in ADD	13M+1I	38 us	—
10 more M in DBL	14M+1I	—	43 us
Measured time of 1ADD	3M+1I	28 us	—
Measured time of 1DBL	4M+1I	—	32 us
Average time of 1I	1I	18.4 us	18.1 us
Average time of 1M	1M	1.0 us	1.0 us

Table 5.2: Time Calculations of Multiplication and Inversion in ECDSA Prototype

For prototype using affine coordinates, calculation of point additions using Equation 3.1 contains 3 field multiplications and 1 inversion ($3M + 1I$), while point doubling using Equation 3.2 has 4 field multiplications and 1 inversion ($4M + 1I$). To test the time of one inversion, I added ten more inversions in each of the operation so that the addition and doubling contains $3M + 11I$ and $4M + 11I$. Then average time of one inversion could be calculated. The measurement of one field multiplication uses the same method. The calculation result is shown in Table 5.2. Obviously, inversion consumes most of the time in both doubling and addition, and the field multiplication time in addition and doubling are only 1.0 ms.

As for projective coordinates, there are no inversions involved, and calculation of point addition and doubling requires $12M+2S$ and $7M+5S$ [15] respectively (where S refers to field squaring). To calculate the time of field multiplication and squaring, ten more field multiplications and squarings are measured respectively. From the measurement, one field multiplication consumes almost the same time with one field doubling, and there is no difference in addition and doubling operations. In Table 5.3, the calculated time in addition and doubling is generated from the average time of one field multiplication and squaring. Compared the calculated time with measured one, there time is consistent (with at most 2 microseconds difference). The time highlighted in red in the table is comparison for addition and blue for doubling.

Description	**# of operations**	**ADD**	**DBL**
10 more S in ADD	12M+12S	29 us	—
10 more S in DBL	7M+15S	—	26 us
10 more M in ADD	22M+2S	28 us	—
10 more M in DBL	17M+2S	—	27 us
Measured time of 1ADD	12M+2S	17 us	—
Measured time of 1DBL	7M+5S	—	15 us
Average time of 1I	1S	1.2 us	1.1 us
Average time of 1M	1M	1.1 us	1.2 us
Calculated time of 1ADD	12M+2S	15.6 us	—
Calculated time of 1DBL	7M+5S		13.9 us

Table 5.3: Time Calculations of Multiplication and Squaring in Improved ECDSA

5.2.2 Protective Implementation

The four protective implementations against side channels are all based on the improved version in projective coordinates. Since the protection requires a lot of extra operations, the efficiency of point multiplication and signing is decreased. And these four versions have different speed performance and security. I measured the number of point additions and doubling, average time of point addition, doubling, multiplication as well as signing time of the four protective versions shown in Table 5.4.

	#ADD	1ADD	#DBL	1DBL	Calculated PM	1PM	SIGN
Version 1	256	17 us	256	15 us	8.20 ms	9.36 ms	10.23 ms
Version 2	64	19 us	256	16 us	5.31 ms	5.84 ms	7.13 ms
Version 3	456	17 us	455	13 us	13.67 ms	15.12 ms	16.26 ms
Version 4	256	20 us	256	15 us	8.98 ms	9.74 ms	11.39 ms

Table 5.4: Operation Counts and Average Time of Four Protective Versions

Seen from the table, the number of addition and doubling increased obviously compared with non-protective implementations. This is because adding extra addition and doubling operations could make performance indistinguishable for each bit of the scalar to protect against simple power analysis attack. Among these extra operations, ones in version 1 and 2 are dummy addition operations, but Montgomery ladder method used in version 3 and 4 performs in a fix pattern to avoid using dummy operations.

One thing needs to be mentioned is, in version 3, average addition and doubling number reaches to 456 and 455 respectively. This is because random scalar splitting method requires generating a random scalar r to perform one more point multiplication. In my implementation, the length of r is 200 bits, so point multiplication rP has 200 additions and 200 doublings. The number of addition and doubling of the other point multiplication $(k-r)P$ will depend on the length of $k-r$. Finally, one more addition is used to calculate $kP = rP + (k-r)P$, and this is the reason why there is one more addition than doubling.

The protective implementations only modify the algorithm in point multiplication, and calculation of point addition and doubling remain unchanged. Therefore number of field multiplication and squaring is still $12M + 2S$ in addition and $7M + 5S$ in doubling. Table 5.5 shows the average time of one multiplication and one squaring in each version. All the timing is measured in microseconds, and there are no apparent differences between these four versions in terms of field squaring and multiplication time in addition and doubling operation. And the calculated time of addition and doubling is consistent with the measured time (with at most 3 microseconds difference).

Description	# of Operations	V.1 ADD	V.1 DBL	V.2 ADD	V.2 DBL	V.3 ADD	V.3 DBL	V.4 ADD	V.4 DBL
10 more S	12M+12S	27 us	—	30 us	—	27 us	—	30 us	—
10 more S	7M+15S	—	26 us	—	28 us	—	24 us	—	26 us
10 more M	22M+2S	28 us	—	31 us	—	26 us	—	32 us	—
10 more M	17M+5S	—	27 us	—	26 us	—	25 us	—	27 us
Measured time of 1ADD	12M+2S	17 us	—	19 us	—	17 us	—	20 us	—
Measured time of 1DBL	7M+5S	—	15 us	—	16 us	—	13 us	—	15 us
Average time of 1S	1S	1.0 us	1.1 us	1.1 us	1.2 us	1.1 us	1.1 us	1.0 us	1.1 us
Average time of 1M	1M	1.1 us	1.2 us	1.2 us	1.0 us	0.9 us	1.2 us	1.2 us	1.2 us
Calculated time of 1ADD	12M+2S	15.2 us	—	16.6 us	—	15 us	—	16.4 us	—
Calculated time of 1DBL	7M+5S	—	13.9 us	—	13 us	—	13.9 us	—	13.9 us

Table 5.5: Time Calculations of Multiplication and Squaring in Four Protective Versions

5.2.3 Summary

In this section, an integrated table is presented according to data produced from previous sections. And the four security countermeasures are evaluated as well.

The unified table makes it better to compare these six implementations. The time of inversion, squaring and multiplication, instead of being presented into point addition and doubling separately, is averaged. Among all these implementations, field multiplication consumes almost same time with around 1.0 us, and average time in field multiplication and squaring in projective coordinates are similar as well.

Besides different performance in these four protective versions, the security against side channel attacks is also evaluated in Table 5.6. These four implementations use overlap methods to withstand basic side channel attacks. The countermeasures are not aim to fight against all kinds of attacks, but to make ECDSA algorithm more secure against different attacks.

Version 1 implements two classic methods double-and-add-always as well as scalar blinding to fight against simple power analysis and differential power analysis. The signing time is slowed down mainly because the dummy addition operations and a random 20-bit number generation. Double-and-add-always can only protect against simple side-channel attacks, but it fails against

	Prototype	Improved	Version 1	Version 2	Version 3	Version 4
#ADD	127	59	256	64	456	256
1ADD	28 us	17 us	17 us	19 us	17 us	20 us
#DBL	256	256	256	256	455	256
1DBL	32 us	15 us	15 us	16 us	13 us	15 us
1I	18.25 us	—	—	—	—	—
1S	—	1.15 us	1.05 us	1.15 us	1.1 us	1.05 us
1M	1.0 us	1.15 us	1.15 us	1.1 us	1.05 us	1.2 us
1PM	12.64 ms	5.42 ms	9.36 ms	5.84 ms	15.12 ms	9.74 ms
SIGN	13.78 ms	6.53 ms	10.23 ms	7.13 ms	16.26 ms	11.39 ms
AUTHOR	—	—	Coron, J.S. [16]	Di Wang / Coron J.S. [16]	Montgomery, P. L. [41]/Ciet, M.& Joye, M. [12]	Montgomery, P. L. [41]/ Coron, J.S. [16]
YEAR	—	—	1999	2014/1999	1987/2003	1987/1999
SPA	—	—	Good	Poor	Good	Good
DPA	—	—	Fair	Fair	Good	Fair
FA	—	—	Poor	Poor	Fair	Fair

Table 5.6: A Summarized Table of Six Implementations

doubling attack [23], and makes C safe-error possible. As for scalar blinding, 20-bit r is not enough to protect against doubling attack, and the scalar may be leaked under carry-based attack [22], but it could make sign-change attack (a kind of fault analysis) more difficult [20].

Version 2 is the most efficient version using window-add-always and scalar blinding. This version in theory should be against simple power analysis and differential power analysis. In window method, unlike binary scalar bits with only two values, there are 16 possibilities if the window size is 4 because 4 bits to represent one value. To make operations of value 0 indistinguishable from other 15 values, window-add-always method is used to ensure extra addition operation is performed when value is zero. However, the possibility of the value being zero is only 1/16, so this method is not strong enough, but it could make simple power analysis impossible.

Version 3 combining Montgomery ladder and random scalar splitting costs a large amount of time in signing but it is regarded as the most secure version among the four. The splitted scalar requires twice of the processing time in point multiplication, which makes the process inefficient, but it is resistant to differential power analysis, and also the doubling attack. As for Montgomery ladder, it shows protection against simple power analysis since it has a regular behavior for each scalar bit, and it is also resistant to C safe-error which is a kind of fault analysis.

Version 4 is a combined version balancing performance and security. It uses a relatively secure method Montgomery ladder, and adopts timesaving scalar blinding to fight against differential power analysis.

Each version has its own strengthens and weaknesses, so they should be carefully evaluated before deciding which one to use to satisfy the demands.

Chapter 6

Conclusion and Future work

In chapter 5, the performance of each version implementing side channel countermeasures had been tested and their security had been evaluated, as a result, not all implementations are strong enough against some side channel attacks. Therefore in this chapter, a plan of further development and an idea of the further work will be proposed, and a review of objectives as well as summary of all different areas in this project will be presented.

6.1 A Review of the Project

The main goals of the project proposed in Section 1.1 reflect the expected features of the implementation, with results and evaluation shown in Chapter 5. Following Table 6.1 shows a simple summary.

Goals	Outcome
Create ECDSA,prototype for signature generation and verification	✓
Manually create a,Bitcoin transaction, integrate it with ECDSA prototype	✓
Improve,performance of ECDSA prototype signature generation algorithm	✓
Adding security countermeasures,against side channel attacks	
Simple power analysis (SPA)	✓
Differential power analysis (DPA)	✓
Fault analysis (FA)	×

Table 6.1: A Summary of Achievements

Most of the goals were successfully achieved in this thesis, but for security countermeasures against fault analysis, although four versions of implementation in some extent could protection against some kinds of fault analysis, there is no specific methods implemented to protect against it. Version 3 and 4 using Montgomery ladder is mainly used to protect against simple power analysis, and fortunately it also protects against C safe-error. Therefore, implementing extra methods against fault analysis will be put into further work in Section 6.2.

6.2 Future Work

The project is in continuous development at the time of writing this thesis, and possible ideas for extending the Bitcoin program's flexibility as well as improving the implementation in terms of speed performance and security considerations will be presented in this section.

The current implementation regarding to manually creating Bitcoin transaction is only compatible with one input and one output, so it is possible to extend the transaction creation with more inputs or outputs to make the transaction more flexible.

In terms of performance improvement, one difficulty unsolved in Section 4.7 was when implementing window method with window size 4, using four repeated doublings consumed some amount of time. Then it is possible to modify doubling function so that four operations could unify into one to save more time. What is more, the BigInteger class in Java API has a low speed when dealing with very large numbers such as ECDSA key pairs and signatures, so another way of improving the signing speed is to rewrite the BigInteger class.

As for security considerations, deterministic ECDSA could avoid bad number generators used when randomly selecting the scalar. Then the function generating secure random scalar in current implementation can be replaced by one to calculate the scalar deterministically using a given message and the private key.

Coherence check, which is a differential fault analysis countermeasure based on Montgomery ladder, is not added in current implementation. If coherence check is implemented in version 4, this version will be more complete and secure to protect against main side channel attacks.

Besides the above two methods strengthening ECDSA implementation, there are still many high level methods that could be used. For example, ensuring one private key will never be used for more than once in Bitcoin transaction could be an effective method. If this is realized into

current implementation, one possible way is that every used key should be saved and compared with the new generated one, but this method will cost a lot of memory.

The last thing needs to be mentioned is, the implementation against specific side channel attacks needs evaluation on it security against other kinds of attacks. This is because an adoption on one countermeasure may benefit other kinds of attacks or introduce new vulnerabilities.

6.3 Conclusion

In conclusion, this project is completed successfully with regard to its initial goals although it may fail to reach the original expectation to some extent. Through the steps working on the step, how ECDSA algorithms work in Bitcoin protocols, how Bitcoin transactions created are all made clear. In addition, to improve ECDSA security against side channel attacks, four more versions are provided based on the performance-improved version. These four versions are against different kinds of side channel attacks like simple power analysis, differential power analysis and fault analysis with different speed performance. According to different demands, different versions could be selected to use.

The failure of reaching the best performance remains a big window for later enhancement, and the experience gained during the implementation is valuable, which will provide a good point for future work.

Bibliography

[1] A. Back. Hashcash – a denial of service counter-measure. 2002.

[2] H Bar-El. Introduction to side channel attacks. *Discretix Technologies LTD 43*, 2003.

[3] S Barber, X. Boyen, E. Shi, , and E. Uzun. Bitter to better –how to make bitcoin a better currency. *Financial Cryptography and Data Security*, pages 399–414, 2012.

[4] N. Benger, J. van de Pol, and N.P. Smart. Ooh aah ... just a little bit: A small anount of side channel can go a long way. *IACR Cryptology ePrint Archive 2014: 161*, 2014.

[5] J. Blomer, M. Otto, and J. P. Seifert. Sign change fault attacks on elliptic curve cryptosystems. *Fault Diagnosis and Tolerance in Cryptography*, pages 36–52, 2006.

[6] J. Bonneau and A. Miller. A cryptocurrency without public-key cryptography. 2014.

[7] E. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. *Public Key Cryptography*, pages 335–345, 2002.

[8] B.B. Brumley and N. Tuveri. Remote timing attacks are still practical. *Computer Security –ESORICS 2011*, pages 355–371, 2011.

[9] BSI. Technical guideline tr-03111, elliptic curve cryptography. 2012.

[10] V. Buterin. Bitcoin is not quantum-safe, and how wen can fix it when needed. *Bitcoin Maganzine*, 2013.

[11] Bouncy Castle, 2014.

[12] M. Ciet and M. Joye. (virtually) free randomization techniques for elliptic curve cryptography. *Information and Communications Security*, pages 348–359, 2003.

[13] M. Ciet and M. Joye. Elliptic curve cryptosystems in the presence of permanent and transient faults. *Designs, codes and cryptography*, 26(1):33–43, 2005.

[14] CoinDesk. Why use bitcoin?

[15] Cryptography/Prime Curve/Standard Projective Coordinates. Wikibooks. 2011.

[16] J.S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. *Cryptographic Hardware and Embedded Systems*, pages 292–302, 1999.

[17] Elliptic curve point multiplication. *Wikipedia*, 2014.

[18] C. Decker and R. Wattenhofer. Bitcoin transaction malleability and mtgox. *arXiv preprint arXiv: 1403.6676*, 2014.

[19] A. Dominguez Oviedo. On fault-based attacks and countermeasures for elliptic curve cryptosystems. 2008.

[20] J. Fan, X. Guo, and E. De Mulder. State-of-the-art of secure ecc implementations: a survey on known side-channel attacks and countermeasures. *Hardware-Oriented Security and Trust (HOST)*, pages 76–87, 2010.

[21] J. Fildes. iphone hacker publishes secret sony playstation 3 key. *BBC News Technology*, 2011.

[22] P.A. Fouque, D. Real, and F. Valette. The carry leakage on the randomized exponent countermeasure. *Cryptographic Hardware and Embedded Systems- CHES 2008*, pages 198–213, 2008.

[23] P.A. Fouque and F. Valette. The doubling attack- why upwards is better than downwards. *Cryptographic Hardware and Embedded Systems- CHES 2003*, pages 269–280, 2003.

[24] C. Gauss. *Disquisitiones arithmeticae. G. Flescher, Leiozig, 1801, English transaction by A. Clark.* 1966.

[25] D.M. Gordon. A survey of fast exponentiation methods. *Journal of algorithms 27.1*, pages 129–146, 1998.

[26] S. Haber and W. S. and Stometta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.

[27] Y. Hitchcock, E. Dawson, and A. Clark. Implementing an efficient elliptic curve cryptosystem ever gf (p) on a smart card. *ANZIAN Journal:44*, pages 354–377, 2003.

[28] J.W. hung, S.G. Sim, and P. J. Lee. Fast implementation of elliptic curve define over gf (pm) on calmrisc with mac2424 coprocessor. *Cryptographic Hardware and Embedded Systems –CHES 2000*, pages 57–70, 2000.

[29] K. Itoh, M. Takenaka, and N. Torii. 'fast implementation of public key cryptography on a dsp tms320c6201. *Cryptographic Hardware and Embedded Systems*, pages 61–72, 1999.

[30] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security 1.1*, pages 36–63, 2001.

[31] D. B. Johnson and A. J. Menezes. Elliptic curve dsa (ecdsa): an enhanced dsa. *Proc. Of the 7th Conference on USENIX Security Symposium (SSYM)*, page 13, 1998.

[32] E. Kasper. Fast elliptic curve cryptography in openssl. *Financial Cryptography and Data Security*, pages 27–39, 2012.

[33] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation48(177)*, pages 203–209, 1987.

[34] P. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss and other systems. *Advances in Cryptology – CRYPTO'96*, 1996.

[35] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. *Advances in Cryptology-CRYPTO'99.*, pages 388–397, 1999.

[36] P. Kocher, J. Jaffe, and B. Jun. Introduction to differential power analysis. *Journal of Cryptographic Engineering 1.1*, pages 5–27, 2011.

[37] L. Martin. The remote timing attack against openssl's ecdsa. *Voltage Security*, 2011.

[38] A.J. Menezes, P. C. Van Oorschot, and S.A. Vanstone. Handbook of applied cryptography. *CRC press*, 2012.

[39] R.C. Merkle. A digital signature based on a conventional encryption function. *Advances in Cryptology – CRYPTO'87*, 1988.

[40] V.S. Miller. Use of elliptic curves in cryptography. *Advances in Cryptology-CRYPTO'85 proceedings*, pages 417–426, 1986.

[41] P.L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, pages 243–264, 1987.

[42] Oracle Technology Network. Java technology reference. 2014.

[43] NIST. Recommended elliptic curves for federal government use. Technical report, July 1999.

[44] K. Okeya, H. Kurumatani, and K. Sakurai. Elliptic curves with the montgomery form and their cryptographic applications. *Public Key Cryptography*, pages 238–257, 2000.

[45] K. Okupski. Bitcoin developer specification – working paper. 2014.

[46] T. Pornin. Deterministic usage of the digital signature algorithm (dsa) and elliptic curve digital signature algorithms (ecdsa). 2013.

[47] D. Ron and A. Shamir. Quantitative analysis of the full bitcoin transaction graph. *Financial Cryptography and Data Security*, pages 6–24, 2013.

[48] Runeks. How to redeem a basic tx. 2012.

[49] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system. 2008.

[50] SECG SEC. 2: Recommended elliptic curve domain parameters. 2000.

[51] K Shirriff. The bitcoin malleability attack graphed hour by hour. *Ken Shirriff's blog*, 2014.

[52] K Shirriff. Bitcoin transaction malleability, looking at the bytes. *Ken Shirriff's blog*, 2014.

[53] K. Shirriff. Bitcoins the hard way: Using the raw bitcoin protocol. *Ken Shirriff's blog*, 2014.

[54] K. Shirriff. Bitcoins the hard way: Using the raw bitcoin protocol. *Ken Shirriff's blog*, 2014.

[55] R. Skudnov. Bitcoin clients. *Instructor 3.12: 32*, 2012.

[56] Eclipse standard 4.4, 2014.

[57] Y. Sung-Ming, S. Kim, and S. Lim. A countermeasure against one physical cryptanalysis may benefit another attack. *Information Security and Cryptology – ICISC 2001*, pages 414–427, 2002.

[58] L. Tung. Security flaw leaves android bitcoin wallets vulnerable to theft. *ZDNet*, 2013.

[59] P. Wuille. Dealing with malleability. *BIP0062*, 2014.

[60] Y. Yarom and K.E. Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. *IACR Cryptology ePrint Archive 2013: 448*, 2013.

Appendix A

Point Addition and Doubling in Projective Coordinates

For points represented in projective coordinates, addition and doubling of point operations can be calculated as follows.

A.1 Point Addition

Given two different points (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2) unequal to point at infinity, addition of these two points (X_3, Y_3, Z_3) can be calculated by:

$U = U_1 - U_2$ where $U_1 = Y_2 * Z_1, U_2 = Y_1 * Z_2$

$V = V_1 - V_2$ where $V_1 = X_2 * Z_1, V_2 = X_1 * Z_2$

$W = Z_1 * Z_2$

$A = U^2 * W - V^3 - 2V^2 * V_2$

Then

$X_3 = V * A$

$Y_3 = U * (V^2 * V_2 - A) - V^3 * U_2$

$Z_3 = V^3 * W$

A.2 Point Doubling

Given one point (X, Y, Z) unequal to point at infinity, doubling of the point (X', Y', Z') can be calculated by:

$W = a * Z^2 + 3 * X^2$

$S = Y * Z$

$B = X * Y * S$

$H = W^2 + 8 * B$

Then

$X' = 2 * H * S$

$Y' = W * (4 * B - H) - 8 * Y^2 * S^2$

$Z' = 8 * S^3$

Appendix B

Unified Formula for Both Point Addition and Doubling

Given two points (x_1, y_1) and (x_2, y_2) on the curve using parameters *secp256k1*, whether they are equal or not, both point addition and doubling can be calculated as follows [7]:

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

$y_3 = -\lambda x_3 - \mu$

where

$\mu = y_1 - \lambda x_1$

$\lambda = (x_1^2 + x_1 x_2 + x_2^2 + a)/(y_1 + y_2)$

Appendix C

Raw Transaction

The implemented Bitcoin transaction uses the raw transaction in [47] shown below, and the result is produced redeeming the second output of the transaction.

```
01000000

01
26c07ece0bce7cda0ccd14d99e205f118cde27e83dd75da7b141fe487b5528fb00000000
8b
48304502202b7e37831273d74c8b5b1956c23e79acd660635a8d1063d413c50b218eb6bc8a022100a10a3a7b5aa
a0f07827207daf81f718f51eeac96695cf1ef9f2020f21a0de02f01410452684bce6797a0a50d028e9632be0c2a
7e5031b710972c2a3285520fb29fcd4ecfb5fc2bf86a1e7578e4f8a305eeb341d1c6fc0173e5837e2d3c7b178aa
de078
ffffffff

02

b06c191e01000000
19
76a9143564a74f9ddb4372301c49154605573d7d1a88fe88ac

00e1f50500000000
19
76a914010966776006953d5567439e5e39f86a0d273bee88ac

00000000
```

Appendix D

Source Code

D.1 ECDSA class

```
public class ECDSA {
        // get the value of sextuple T = (p,a,b,G,n,h) in curve secp256k1
        public static GenerateKey key = new GenerateKey();
        public BigInteger N = key.N;
        public BigInteger p = key.p;
        public BigInteger zero = BigInteger.ZERO;
        public BigInteger one = BigInteger.ONE;
        public EllipticCurve curve = key.curve;
        public ECPoint G = key.G;
        static NumberFormat formatter = new DecimalFormat("#0.00");
         PointMultiplication PM =new PointMultiplication();
        /**
         * This method generates a secure random number k in [1,n-1].
         */
        public BigInteger SelectK() throws NoSuchAlgorithmException {

                SecureRandom sr = new SecureRandom();
                BigInteger K = new BigInteger(256, sr);
                K = K.mod(N.subtract(one));
                while (K.equals(zero) || !(K.gcd(p).compareTo(one) == 0)) {
                        K = new BigInteger(256, sr);
                        K = K.mod(N.subtract(one));
                }
                return K;

        }

        /**
```

```
     * This method signs a message m given the private key.
     */
    public BigInteger[] Sign(byte[] m, BigInteger privatekey)
                    throws NoSuchAlgorithmException {
            Long startTime = System.currentTimeMillis();
            BigInteger K = SelectK();
            BigInteger dm = SHA1(m);
            // calculate the public key curve point Q.
            ECPoint Q = PM.ScalarMulti(K, G);
            // calculate R = x_q mod N
            BigInteger R = Q.getAffineX().mod(N);
            // calculate S = k^-1 (dm + R*privateKey) mod N
            BigInteger Kin = K.modInverse(N);
            BigInteger mm = dm.add(privatekey.multiply(R));
            BigInteger S = (Kin.multiply(mm)).mod(N);
            // if R or S equal to zero, resign the message
            if (R.equals(zero) || S.equals(zero)) {

                    K = SelectK();
                    Q = PM.ScalarMulti(K, G);
                    R = Q.getAffineX().mod(N);
                    Kin = K.modInverse(N);
                    mm = dm.add(privatekey.multiply(R));
                    S = (Kin.multiply(mm)).mod(N);
            }

            BigInteger[] Signature = { R, S };
            Long endTime = System.currentTimeMillis();
            Long totalTime = endTime - startTime;
            System.out.println("Runing Time of singning in ECDSA:"
                            +totalTime*1000 + "us");
            return Signature;

    }

    /**
     * This method calculates the hash value by using SHA-1 algorithm.
     */
    public BigInteger SHA1(byte[] m) throws NoSuchAlgorithmException {
            MessageDigest mDigest = MessageDigest.getInstance("SHA1");
            byte[] result = mDigest.digest(m);
            return new BigInteger(result);
    }

    /**
     * This method verifies a signature on message m given the public key.
     */
```

```
public void verify(byte[] m, BigInteger[] signature,
                ECPoint publickey) throws NoSuchAlgorithmException {
        Long startTime = System.currentTimeMillis();
        // if R and S is not in [1,n-1], signature invalid.
        if (signature[0].compareTo(N) == 1 || signature[0].compareTo(one) ==
            -1
                        || signature[1].compareTo(N) == 1
                        || signature[1].compareTo(one) == -1) {
                System.out.println("SIGNATURE WAS NOT IN VALID RANGE");
        }
        // calculate w = S^-1 mod N
        BigInteger w = signature[1].modInverse(N);
        BigInteger h = SHA1(m);
        // calculate u1= hw mod N and u2=Rw mod N
        BigInteger u1 = (h.multiply(w)).mod(N);
        BigInteger u2 = (signature[0].multiply(w)).mod(N);
        // calculate the curve point (x1,x2)=u1*G+u2*publicKey
        ECPoint p1 = PM.ScalarMulti(u1, G);
        ECPoint p2 = PM.ScalarMulti(u2, publickey);

        ECPoint pt = PM.AddPoint(p1, p2);
        // calculate V = x1 mod N
        BigInteger V = pt.getAffineX().mod(N);
        // if R=V, signature valid, otherwise invalid
        if (V.equals(signature[0]))
                System.out.println("Valid signature");
        else
                System.out.println("Invalid signature");
        Long endTime = System.currentTimeMillis();
        Long totalTime = endTime - startTime;

        System.out.println("Runing Time of verification:"
                        + formatter.format(totalTime) + "ms");
}

/**
 * This method generates a signature, verifies it and calculate the running
 * time of the whole process.
 */
public static void main(String[] arg) throws NoSuchAlgorithmException,
                InvalidAlgorithmParameterException, NoSuchProviderException {

        ECDSA ecdsa = new ECDSA();
        Format format = new Format();
        String message = "ECDSA TEST";
        byte[] m = message.getBytes();
        BigInteger[] keypair = key.KeyGeneration();
```

```
                BigInteger privatekey = keypair[0];
                ECPoint publickey = new ECPoint(keypair[1], keypair[2]);
                System.out.println("private key is: " + privatekey.toString(16));
                System.out.println("The private key is: "
                                + format.format(privatekey.toByteArray()));
                System.out.println("x of public key is: "
                                + publickey.getAffineX().toString());
                System.out.println("y of public key is: "
                                + publickey.getAffineY().toString());
                BigInteger[] signature = ecdsa.Sign(m, privatekey);
                System.out.println("the value of R is:" + signature[0]);
                System.out.println("the value of S is:" + signature[1]);
                ecdsa.verify(m, signature, publickey);
        }

}
```

D.2 GenerateKey class

```
public class GenerateKey {

        // public static void main(String[] arg) throws NoSuchAlgorithmException,
        // NoSuchProviderException, InvalidAlgorithmParameterException{

        // Define the parameters of sextuple T = (p,a,b,G,n,h) of curve Secp256k1
        public static BigInteger N = new BigInteger(
                        "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE
                        BAAEDCE6AF48A03BBFD25E8CD0364141",
                        16);
        public static BigInteger p = new BigInteger(
                        "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
                        FFFFFFFFFFFFFFFFFFFFFFFFFFFFFC2F",
                        16);
        public static BigInteger a = new BigInteger(
                        "00000000000000000000000000000000
                        0000000000000000000000000000000000000000",
                        16);
        public static BigInteger b = new BigInteger(
                        "00000000000000000000000000000000
                        0000000000000000000000000000000000000007",
                        16);

        public static EllipticCurve curve = new EllipticCurve(new ECFieldFp(p),// p
                        a, // a
                        b);// b
        public static ECPoint G = ECPointUtil
```

```
                        .decodePoint(
                                        curve,
                                        Hex.decode("0279BE667EF9DCBBAC55A062
                                        95CE870B07029BFCDB2DCE28D959F2815B16F81798"))
                                            ;// G
        static ECParameterSpec ecSpec = new ECParameterSpec(curve, G, // G
                        N, // n
                        1); // h

        /**
         * This method generates a secure random ECDSA public/private key pair.
         */
        public BigInteger[] KeyGeneration()
                        throws InvalidAlgorithmParameterException,
                        NoSuchAlgorithmException, NoSuchProviderException {
                // generate the key pair randomly
                KeyPairGenerator g = KeyPairGenerator.getInstance("EC", "SunEC");
                g.initialize(ecSpec, new SecureRandom());
                KeyPair pair = g.generateKeyPair();
                // get the private key
                PrivateKey priKey = pair.getPrivate();
                BigInteger s = ((ECPrivateKey) priKey).getS();
                // get the public key
                PublicKey pubKey = pair.getPublic();
                ECPoint w = ((ECPublicKey) pubKey).getW();
                BigInteger[] keypair = { s, w.getAffineX(), w.getAffineY() };
                BigInteger px = w.getAffineX();
                BigInteger py = w.getAffineY();
                // if the length of x and y in public key is not 32 bytes, regenerate
                // the key pair
                if (px.toByteArray().length != 32 || py.toByteArray().length != 32) {
                        return KeyGeneration();
                }
                return keypair;
        }
}
```

D.3 PointMultiplication class (In affine coordinates)

```
public class PointMultiplication {

        public static BigInteger zero = BigInteger.ZERO;
        public static BigInteger two = new BigInteger("2");

        /**
         * This method performs points addition operation given two points p1 and
```

```
     * p2.
     */
    public ECPoint AddPoint(ECPoint p1, ECPoint p2) {
            ECPoint p3 = new ECPoint(zero, zero);

            // p3 is an Infinity point;
            if (p2.getAffineY().equals(zero)
                            || p1.getAffineY().equals(zero)
                            || (p2.getAffineX().equals(p1.getAffineX()) && p2.
                                getAffineY()
                                                .equals(p1.getAffineY().negate())))
                    p3 = ECPoint.POINT_INFINITY;

            // if two points are equal -- DoublePoint method
            else if ((p1.getAffineX().equals(p2.getAffineX()))
                            && (p1.getAffineY().equals(p2.getAffineY())))
                    p3 = DoublePoint(p1);
            // if one of point is an infinity point
            else if (p1.equals(ECPoint.POINT_INFINITY))
                    p3 = p2;
            else if (p2.equals(ECPoint.POINT_INFINITY))
                    p3 = p1;
            // if points are not equal
            // calculate the slope s = (p2y-p1y)/(p2x-p1x) mod p
            BigInteger s1 = p2.getAffineY().subtract(p1.getAffineY()); // s1=yp2-
                yp1
            BigInteger s2 = p2.getAffineX().subtract(p1.getAffineX()); // s2=xp2-
                xp1
            BigInteger s = s1.multiply(s2.modInverse(ECDSA.key.p)).mod(
                            ECDSA.key.p);// slope = (s1/s2) mod p
            // calculate p3x =(slope^2-p1x-p2x) mod p
            BigInteger p3x1 = s.pow(2);// p3x1 = s^2
            BigInteger p3x = p3x1.subtract(p1.getAffineX())
                            .subtract(p2.getAffineX()).mod(ECDSA.key.p);// p3x =(
                                p3x1-p1x-p2x)mod p

            // calculate p3y = (slope(p1x-p3x)-p1y) mod p
            BigInteger p3y1 = p1.getAffineX().subtract(p3x); // p3y1 = p1x-p3x
            BigInteger p3y = s.multiply(p3y1).subtract(p1.getAffineY())
                            .mod(ECDSA.key.p);// p3y = (slope*p3y1-p1y) mod p

            p3 = new ECPoint(p3x, p3y);
            return p3;
    }
    /**
     * This method performs points doubling operation given one point p.
     */
```

```
        public ECPoint DoublePoint(ECPoint P) {

                ECPoint p3 = new ECPoint(zero, zero);

                // calculate the slope = (3*P x^2 +a)/2*Py mod p
                BigInteger s1 = P.getAffineX().pow(2).multiply(new BigInteger("3"))
                                .add(ECDSA.key.a); // s1 = 3*Px^2 +a
                BigInteger s2 = P.getAffineY().multiply(new BigInteger("2"));// s2=2*
                    Py
                BigInteger s = s1.multiply(s2.modInverse(ECDSA.key.p)); // s = (s1/s2
                    )mod p
                // calculate p3x = (slope^2 - 2*Px) mod p
                BigInteger p3x1 = s.pow(2);// p3x1 = s^2
                BigInteger p3x2 = P.getAffineX().multiply(new BigInteger("2"));//
                    p3x2 = 2*Px
                BigInteger p3x = p3x1.subtract(p3x2).mod(ECDSA.key.p);// p3x =(p3x1-
                    p3x2) mod p
                // calculate p3y = (slope(Px-P3X)-Py) mod p
                BigInteger p3y1 = P.getAffineX().subtract(p3x);// p3y1 = Px-p3x
                BigInteger p3y = s.multiply(p3y1).subtract(P.getAffineY())
                                .mod(ECDSA.key.p);// p3y = (s*p2y1-Py) mod p
                p3 = new ECPoint(p3x, p3y);
                return p3;
        }
        /**
         * This method performs scalar multiplication operation given one point p
         * and an integer.
         */
        public ECPoint ScalarMulti(BigInteger kin, ECPoint G) {
                String K = kin.toString(2);
                ECPoint q = new ECPoint(zero, zero);
                q = ECPoint.POINT_INFINITY;
                if (K.substring(0, 1).equals("1")) {
                        q = G;
                }
                for (int i = 1; i < K.length(); i++) {
                        q = DoublePoint(q);
                        if (K.substring(i, i + 1).equals("1")) {
                                q = AddPoint(q, G);
                        }

                }
                return q;
        }
}
```

D.4 pointMultiplication class (In projective coordinates)

```
public class pointMultiplication {
        private BigInteger X;
        private BigInteger Y;
        private BigInteger Z;
        private BigInteger zinv;
        private static BigInteger zero = BigInteger.ZERO;
        private boolean infinity;
        // precomputed points
        static pointMultiplication[] precompute = {};

        /**
         * constructor1
         */
        public pointMultiplication(BigInteger x, BigInteger y, BigInteger z) {
                this.X = x;
                this.Y = y;
                if (z == null) {
                        this.Z = BigInteger.ONE;
                } else {
                        this.Z = z;
                }
                this.zinv = null;
        }
        /**
         * constructor2
         */
        public pointMultiplication() {
        }
        /**
         * This method gets affine x representation
         */
        public BigInteger getX() {
                if (this.zinv == null) {
                        this.zinv = this.Z.modInverse(ECDSA.key.p);

                }
                return X.multiply(this.zinv).mod(ECDSA.key.p);

        }
        /**
         * This method gets affine y representation
         */
        public BigInteger getY() {
                if (this.zinv == null) {
                        this.zinv = this.Z.modInverse(ECDSA.key.p);
```

```
                }
                return Y.multiply(this.zinv).mod(ECDSA.key.p);
        }
        /**
         * This method checks point at infinity
         */
        public boolean isInfinity() {
                if (X == zero && Y == zero) {
                        return true;
                }
                return false;
        }
        /**
         * This method calculates point addition
         */
        public pointMultiplication AddPoint(pointMultiplication a) {
                if (this.isInfinity()) {
                        return a;
                }
                if (a.isInfinity()) {
                        return this;
                }
                pointMultiplication R = new pointMultiplication(zero, zero, null);

                // U1 = Y2*Z1
                BigInteger U1 = a.Y.multiply(Z);
                // U2 = Y1*Z2
                BigInteger U2 = Y.multiply(a.Z);
                // V1 = X2*Z1
                BigInteger V1 = a.X.multiply(Z);
                // V2 = X1*Z2
                BigInteger V2 = X.multiply(a.Z);
                if (V1 == V2) {
                        if (U1 != U2) {
                                this.infinity = true;
                                return R;

                        } else
                                return DoublePoint();
                }
                // U=U1-U2
                BigInteger U = U1.subtract(U2).mod(ECDSA.key.p);
                // V=V1-V2
                BigInteger V = V1.subtract(V2).mod(ECDSA.key.p);
                // W=Z1*Z2
                BigInteger W = Z.multiply(a.Z).mod(ECDSA.key.p);
```

```
            // A1=U^2*W
            BigInteger A1 = U.pow(2).multiply(W);
            // A2=V^3
            BigInteger A2 = V.pow(3);
            // A3=2*V^2*V2
            BigInteger A3 = V.pow(2).multiply(V2).multiply(new BigInteger("2"));
            // A=A1-A2-A3
            BigInteger A = A1.subtract(A2).subtract(A3).mod(ECDSA.key.p);
            // x3 = V*A
            BigInteger x3 = V.multiply(A).mod(ECDSA.key.p);
            // y31 = V^2*V2-A
            BigInteger y31 = V.pow(2).multiply(V2).subtract(A);
            // y32 = V^3*U2
            BigInteger y32 = V.pow(3).multiply(U2);
            // y3 = U*y32-y31
            BigInteger y3 = U.multiply(y31).subtract(y32).mod(ECDSA.key.p);
            // z3=V^3*W
            BigInteger z3 = V.pow(3).multiply(W).mod(ECDSA.key.p);
            pointMultiplication P = new pointMultiplication(x3, y3, z3);
            return P;

    }
    /**
     * This method calculates point doubling
     */
    public pointMultiplication DoublePoint() {
            pointMultiplication R = new pointMultiplication(zero, zero, null);
            if (this.isInfinity()) {
                    return this;
            }
            if (Y.signum() == 0) {
                    this.infinity = true;
                    return R;
            }
            // W1 = a*Z^2
            BigInteger W1 = Z.pow(2).multiply(ECDSA.key.a);
            // W2 = 3*X^2
            BigInteger W2 = X.pow(2).multiply(new BigInteger("3"));
            // W=W1-W2
            BigInteger W = W1.add(W2).mod(ECDSA.key.p);
            // S=Y*Z
            BigInteger S = Y.multiply(Z).mod(ECDSA.key.p);
            // B=X*Y*S
            BigInteger B = X.multiply(Y).multiply(S).mod(ECDSA.key.p);
            // H=W^2-8*B
            BigInteger H = W.pow(2).subtract(B.multiply(new BigInteger("8")))
                            .mod(ECDSA.key.p);
```

```
                // X3=2*H*S
                BigInteger x3 = H.multiply(S).multiply(new BigInteger("2"))
                                .mod(ECDSA.key.p);
                // Y31 = W*(4*B - H) - 8*Y^2*S^2
                BigInteger y31 = B.multiply(new BigInteger("4")).subtract(H);
                // Y32 = 8*Y^2*S^2
                BigInteger y32 = Y.pow(2).multiply(S.pow(2))
                                .multiply(new BigInteger("8"));
                // Y3 = W*Y31 - Y32
                BigInteger y3 = W.multiply(y31).subtract(y32).mod(ECDSA.key.p);
                // Z3 = 8*S^3
                BigInteger z3 = S.pow(3).multiply(new BigInteger("8")).mod(ECDSA.key.
                    p);
                pointMultiplication P = new pointMultiplication(x3, y3, z3);
                return P;
        }
        /**
         * This method calcualtes point multiplication using window method
         */
        public pointMultiplication Scalar(BigInteger kin) {
                String K = kin.toString(16);
                pointMultiplication q = new pointMultiplication(zero, zero,null);
                int a = Integer.parseInt(K.substring(0, 1), 16);
                // if d1>0, q=d1P (precomputed value)
                if (a > 0) {
                        q = precompute[a];
                }
                // for i=1 to n
                for (int i = 1; i < K.length(); i++) {
                        // q=2^4P=16P=DoublePoint(DoublePoint(DoublePoint(DoublePoint
                            (q)))).
                        q = q.DoublePoint();
                        q = q.DoublePoint();
                        q = q.DoublePoint();
                        q = q.DoublePoint();
                        // if di>0,q=q+diP(precomputed value)
                        int di = Integer.parseInt(K.substring(i, i + 1), 16);
                        if (di > 0) {
                                q = q.AddPoint(precompute[di]);
                        }
                }
                return q;

        }
}
```