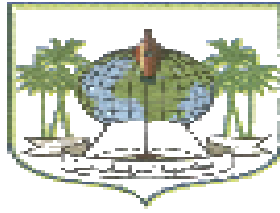


The Libyan Academy

Tripoli-Libya



School of Applied Science and Engineering

Department of Electrical and Computer Engineering

**Efficient Implementation of Elliptic Curve
Cryptography for Wireless Sensor Networks**

*Thesis submitted to the department of Electrical and computer Engineering in partial Fulfillment
of the Requirements for the Degree of Master in Engineering science*

By:

FATMA TAHER EL-HADI

Supervisor by:

Dr. Aboubaker El-Houni

Dr. El-Bahlul Fgee

Acknowledgements

I thank Allah for helping me to complete my thesis.

I would like to express my gratitude to my supervisor Dr. Fgee for his guidance and motivation towards the completion of this thesis.

I would also like to thank my supervisor, Dr. Aboubaker El-Houni.

Finally, I would like to thank my family and especially my husband for their patience and advice.

ABSTRACT

When deploying wireless sensor networks (WSNs) in public environments it may become necessary to secure their data storage and transmission against possible attacks. The nodes feature only small computational and energy resources, thus requiring efficient algorithms. As a solution for this problem the Elliptic Curve cryptosystem.

Strong public key cryptography is therefore commonly seen as infeasible on such devices. In this thesis an efficient and lightweight implementation of public-key cryptography algorithms relying on elliptic curves is proposed. Normal basis representation uses minimal RAM, random access memory. Then optimal normal base is chosen because its better suited for sensor node.

Testing results shows less memory size compared with existing software implementation.

good results are expected by using Elliptic Curve cryptosystem since it needs low memory and hard to penetrate. Therefore, a security can be achieved in the sensor networks environment.

TABLE OF CONTENTS

TITLE	PAGE
List of Tables	i
List of Figures	ii
CHAPTER 1: INTRODUCTION	1
1.1 Importance of Security	1
1.2 Related Work	2
1.3 Thesis Objective	3
1.4 Problem Statement	3
1.5 Proposed Solution	4
1.6 Thesis Layout	5
CHAPTER 2: SENSOR NETWORKS	6
2.1 Introduction	6
2.2 Important Application of Sensor Networks	8
2.3 Sensor Network Platforms	9
2.4 Sensor Node Technology	10
CHAPTER 3: SECURITY AND ECC	13
3.1 Security Requirements	13
3.1.1 Confidentiality	16
3.1.2 Authentication	16
3.1.3 Integrity	17
3.1.4 Freshness	17
3.1.5 Availability	18
3.1.6 Secure Management	18
3.2 Defense Mechanisms	18
3.3 Cryptographic Algorithms	19
3.3.1 Symmetric Encryption	20
3.3.2 Asymmetric Encryption	20
3.4 Elliptic Curve Cryptography	21
3.4.1 Mathematics of Elliptic Curve over Real Numbers	22
3.4.2 Mathematics of Elliptic Curve over Galois Fields	24
3.4.3 Elliptic Curve Discrete Logarithm Problem	26
CHAPTER 4: FUNDAMENTALS OF ELLIPTIC CURVE CRYPTOGRAPHY	28
4.1 A Normal Basis	28

TITLE	PAGE
4.1.1 Squaring Normal Basis Number	29
4.1.2 Multiplication	29
4.2 ONB mathematics and Subroutines	30
4.2.1 Type I Optimal Normal Basis	32
4.2.2 Type II Optimal Normal Basis	34
4.2.3 Multiplication Routine (opt_mul)	37
4.2.4 Inversion (opt_inv) Routine	38
4.3 Optimal Normal Basis Elliptic Curve Subroutines	40
4.3.1 Multiplication over Elliptic Curves	41
4.3.1.1 Balanced Representation	42
4.4 Choosing an Elliptic Curve	42
4.4.1 Nonsupersingular Curves	43
4.4.2 Embedding Data on a Curve	43
4.4.3 Solving Quadratic Equation in Binary Fields	44
4.4.4 The Trace Function	45
4.4.5 Quadratic Normal basis Routine.....	46
4.5 Simple Protocols	47
4.5.1 Random Bit Generator	48
4.5.2 Choosing Random Curves	48
4.5.3 Elliptic Curve Diffie-Hellman ECDH	48
4.5.4 El-Gamal	49
Chapter 5: Results of Implementation of Elliptic Curve Cryptography	51
5.1 Introduction	51
5.2 Software Used	52
5.3 Simulation Scenario	52
5.3.1 Diffie-Hellman	53
5.3.2 El-Gamal Protocol	69
5.4 Comments on the results	82
CHAPTER 6: CONCLUSION AND FUTURE WORK	83
6.1 Conclusion	83
6.2 Future Work	83
REFERENCES	84
APPENDICES	88

List of Tables

TITLE	PAGE
Table 3.1.Security requirements for some application with Potential security threats	15
Table 4.1 Antilog Table for 2^i for $m=4$	33
Table 4.2 Log Table for 2^i for $m=4$	34
Table 4.3 Power of $2^i \bmod 19$ (antilog table)	35
Table 4.4 Log base 2 of $i \bmod 19$ (log table)	35
Table 4.5 Λ vectors for $m=9$	36
Table 5.1 comparison between software implementation of ECC	52

List of Figures

TITLE	PAGE
Figure 3.1 Symmetric Cryptography	20
Figure 3.2 Asymmetric Cryptography	21
Figure 3.3 Plot of elliptic curve $y^2 = x^3 - 7x + 5$	22
Figure 3.4 Addition of elliptic curve points over a real number curve	23
Figure 3.5 Addition of a point to itself.	24
Figure 4.1 Subroutine schematic(1)	31
Figure 4.2 Subroutine schematic(2)	40
Figure 4.3 subroutine schematic(3)	46
Figure 4.4 subroutine schematic(4).....	47
Figure 5.1 Type I ONB, $m=162$ and random (curve, base point)	53
Figure 5.2 Type I ONB, $m=162$ and another random (curve, base point)	54
Figure 5.3 Type I ONB, $m=162$ and koblitz curve , base point	55
Figure 5.4 Type I ONB, $m=162$ and koblitz curve , another base point	56
Figure 5.5 Type I ONB, $m=172$ and random curve, Base point	57
Figure 5.6 Type I ONB, $m=172$ and another (random curve, Base point)	58
Figure 5.7 Type I ONB, $m=172$ and koblitz curve , base point	59
Figure 5.8 Type I ONB, $m=172$ and koblitz curve , another base point	60
Figure 5.9 Type II ONB, $m=173$ and random (curve, base point)	61
Figure 5.10 Type II ONB, $m=173$ and another random (curve, base point)	62
Figure 5.11 Type II ONB, $m=173$ and koblitz curve , base point	63
Figure 5.12 Type II ONB, $m=173$ and koblitz curve , another base point	64
Figure 5.13 Type II ONB, $m=174$ and random (curve, base point)	65
Figure 5.14 Type II ONB, $m=174$ and another random (curve, base point)	66
Figure 5.15 Type II ONB, $m=174$ and koblitz curve , base point	67
Figure 5.16 Type II ONB, $m=174$ and koblitz curve , another base point	68
Figure 5.17 Type II ONB, $m=174$ and public curve , base point	70
Figure 5.18 Type II ONB, $m=174$ and different public curve , base point	71

Figure 5.19 Type II ONB, $m=174$ and Koblitz curve , base point	72
Figure 5.20 Type II ONB, $m=174$ and public curve , different base point	73
Figure 5.21 Type II ONB, $m=173$ and Koblitz curve , base point	74
Figure 5.22 Type II ONB, $m=173$ and Koblitz curve , different base point	75
Figure 5.23 Type I ONB, $m=172$ and Koblitz curve , base point	76
Figure 5.24 Type I ONB, $m=172$ and Koblitz curve , different base point	77
Figure 5.25 Type I ONB, $m=162$ and public curve , base point	78
Figure 5.26 Type I ONB, $m=162$ and different public curve , base point	79
Figure 5.27 Type I ONB, $m=162$ and Koblitz curve , base point	80
Figure 5.28 Type I ONB, $m=162$ and koblitz curve , different base point	81

CHAPTER 1

INTRODUCTION

Advancements in micro electro mechanical systems and wireless networks have made possible the advent of tiny sensor nodes which are low cost small tiny devices with limited coverage, low power, smaller memory sizes and low bandwidth. Physically small sensor devices are able to communicate with each other within its radio communication range.

A wireless sensor network consists of large number of sensor nodes spread over a geographical area. Sensor networks have a wide variety of uses including monitoring environmental issues, or military surveillance.

1.1 Importance of Security

Sensor networks consist of nodes that sense their environment, process data, and communicate through wireless links. They are expected to support a wide variety of applications, many of which have at least some requirements for security.

To protect sensible data against threats and to ensure security properties like integrity, authenticity or confidentiality traditional network protocols rely on cryptographic primitives like encryption and decryption schemes.

Cryptographic algorithm for authentication and encryption can be implemented in two ways, using asymmetric or symmetric key encryption.

In wired data networks, nodes rely on pre-deployed trusted server to help establish trust relationships. However in WSN, these trusted authorities do not exist because sensor nodes have limited memory, CPU power, and energy, hence cryptographic algorithms must be selected carefully.

1.2 Related Work

Recently a lot of authors studied the efficiency of elliptic curve cryptosystems on microcontrollers which are used in wireless sensor motes. They focus on the TI MSP430 and the ATmega128 developed by Atmel. Gura first reported the efficiency of elliptic curve cryptosystems defined over prime fields on the ATmega128 in [1]. They used curves published in the recommendations of SEC [2]. The execution time of one ECDSA signature generation is 810 ms at 8MHz. Work by Uhsadel and Scott [3, 4] further improves speed results on the ATmega128. Malan refers in [5] to an ECC implementation based on binary fields on the ATmega128 microcontroller. It was implemented using a C++ and Java library. The running times were quite slow, a complete point multiplication was executed in 34 seconds. Shi and Yan showed in [6], that a sophisticated implementation of the inversion algorithm enables an improvement for binary curves. Their run times for a scalar multiplication was 12.2 seconds. In 2005 Blaß and Zitterbart implemented finite fields of characteristic two on the ATmega128 [7] clocked at 7MHz, too. The elements of the finite field of degree 113 were represented by normal bases and the multiplication of a fixed base point took about 6.7 seconds. In 2008 Szczechowiak and Seo reported new speed records for elliptic curve realizations

concerning binary fields [8, 9]. While Szczechowiak consumes 2.16 seconds for a scalar multiplication on a binary Koblitz curve.

1.3 Thesis Objective

In this thesis an efficient and lightweight (low size and low power consumption) implementation of public-key cryptography algorithms relying on Elliptic Curves is presented to solve the problem of security threats.

1.4 Problem Statement

In wireless sensor networks (WSN), there are major security problems because the communication is wirelessly and the devices are often easy to access. Therefore, an adversary can easily eavesdrop on communication or simply steal a node. An adversary can often read out any content that is stored on a node. Furthermore, the devices are very constrained in terms of memory, computing power, and energy supply. Most commonly utilized sensor hardware is based around a battery powered microcontroller like e.g. Atmels 8Bit ATMEGA128 offering only 7 MIPS with 4KByte RAM.

Symmetric algorithms are generally preferable to asymmetric algorithms in the field of wireless sensor networks, because they required memory less than asymmetric algorithms, so the energy consumption is less. However, when symmetric algorithms are used, there are some problems arise: key distribution and number of stored keys. When individual keys are used in a wireless sensor network with n nodes, each node has to store $(n - 1)$

keys. This is unsuitable for large wireless sensor networks, because large memory is needed to store all the keys. Moreover, the capture of a single node by an adversary may compromise the security of the entire network.

1.5 Proposed Solution

Based on the previous reasons, asymmetric algorithms are very valuable for key establishment and authentication in WSN. There exist several protocols for asymmetric cryptographic algorithms for WSNs. The most suitable asymmetric cryptography primitives for WSN, is Elliptic Curve Cryptography cryptosystem (ECC). ECC has attracted much attention as the security solution for wireless networks rather than RSA due to the small key size and low computational overhead. RSA needs much longer key lengths compared to elliptic curve cryptography to achieve the same security level (1024 bit vs. 160-bit).

Arithmetic operations in the finite field $GF(2^m)$ [10] are extensively used in elliptic curve cryptographic protocols. The binary field $GF(2^m)$ is a set of 2^m elements, where each of these elements can be represented by a bit string of length m . The field can also be considered as an m -dimensional vector space with m linearly independent elements over $GF(2^m)$ known as a basis. There are two common bases, polynomial basis and normal basis, to represent field elements of $GF(2^m)$. Normal basis uses minimal RAM. Then the optimal normal base is chosen because its better suited for sensor node.

The key for efficiency are memory optimizations like moving data from expensive RAM to ROM as well as a precomputation of base points for faster execution [11].

1.6 Thesis Layout

The thesis is organized as follows. Chapter 2 gives a brief description of sensor node, including the application of sensor networks. Chapter 3 surveys the security requirements in WSNs and algorithms that used, then gives some background information on Elliptic Curve Cryptography and supporting arithmetic. Chapter 4 presents the details of the basic mathematical theory behind the elliptic curve cryptosystem and describe the Diffie-Hellman and El-Gamal algorithms.

The results of our implementation are presented in Chapter 5. Finally, Chapter 6 presents conclusion the thesis and potential future work is presented.

CHAPTER 2

SENSOR NETWORKS

2.1 Introduction

Major advances in wireless communications and electronics technology have resulted in the emergence of low cost, low-power micro-sensors, actuators, embedded processors, and radio devices. Advances in VLSI technology have enabled the integration of all these technologies on a single device with a very small form factor [12]. As a result, new sets of computing devices with sensing, actuating, communicating, and processing capability are being developed. Because of increasing appetite to know more about and be aware of relevant happenings in the surrounding environment which driving to use of these devices in autonomous networks. Such networks are capable of reporting after monitoring, sensing and even actuating their environment.

These devices, referred to as wireless sensor nodes, will eventually be deployed in hundreds and thousands in wireless sensor networks. The emergence of such networks has opened up a lot of avenues for research in different fields such as distributed computing, sensor node design, communication technology, operating systems, communication protocols, security, and software reconfiguration.

Applications deployed on sensor networks are composed of many software components executing on individual sensor nodes. The objective is to perform software reconfiguration on individual sensor nodes to alter their functionality without human interference.

A wireless sensor network consists of a number of nodes spread across a geographical area. The nodes are typically fitted with sensors (for example light, temperature, pressure, and audio sensors) for monitoring their environment and have wireless communication capability [13]. The nodes also have some degree of intelligence for processing the data gathered by their sensors and the capacity to do so. The network is deployed over a geographical area in an ad hoc manner. Some nodes within the network could be placed at known locations, but in general, the nodes determine their positions themselves using various localization algorithms. They can also be fitted with actuators that perform some mechanical action. The nodes communicate with each other using wireless technologies like radio. In typical deployments, not all nodes are within wireless range of each other.

Wireless sensor networks are quickly gaining popularity due to the fact that they are potentially low cost solutions to a variety of real-world challenges. Their low cost provides a means to deploy large sensor arrays in a variety of conditions capable of performing both military and civilian tasks. But sensor networks also introduce severe resource constraints due to their lack of data storage and power. Both of these represent major obstacles to the implementation of traditional computer security techniques in a wireless sensor network. The unreliable communication channel and unattended operation make the security defenses even harder. Indeed, wireless sensors often have the processing characteristics of machines that are decades old (or longer), and the industrial trend is to reduce the cost of wireless sensors while maintaining similar computing power. With that in mind, many researchers have begun to address the challenges of

maximizing the processing capabilities and energy reserves of wireless sensor nodes while also securing them against attackers.

2.2 Important Applications of Sensor Networks

Due to their communication model, these networks have potential applications in many areas. Although the application areas are classified into environment, military and civil, these are not an exhaustive list of area of their application.

Some of their applications are:

1. Military Applications: Sensors are widely used in applications such as surveillance, communication from intractable areas to base stations. Since these are inexpensive and deployed in large numbers, loss of some of these sensors would not affect the purpose for which they were deployed.

2. Distributed Surveillance: Highly mobile sensor networks like the Underwater Autonomous vehicle Odyssey make it possible to transmit huge amounts of data at low powers [13].

3. WINS Wireless Sensing Networks: These networks contain large arrays of distributed sensors and the interpolation (by making use of multiple sensors on each node) of various sensed datum gives high quality information. These networks are primarily used in military land and for monitoring complex machinery processes [13].

4. Structure Monitoring: Structure monitoring systems detect, localize, and estimate the extent of damage. Civil engineering structures can be tested for soundness using sensors.

5. Pollution and Toxic Level Monitoring: These sensors collect data from industrial areas where toxic spills occur. These are useful in sensing Nuclear, Biological, and Chemical phenomena in environment and transmitting it to remote stations for analysis.

6. Sensors for Vision: These are collaborative self-organizing sensor networks which have many micro sensors built on a chip and implanted in the eye. This improves the vision of people with "no vision or limited vision" [13].

7. Smart Sensor Networks: These networks have a number of independent sensors. Each of the sensors makes a local decision and all the decisions are combined and weighed based on a specific algorithm and a global decision is taken.

8. Rainfall and Flood Monitoring: These networks have water level, wind and temperature sensors and the data is transmitted to a central database for analyzing and forecasting weather.

9. Other Applications: These involves habitat monitoring for determining biocomplexity. These include resource explorations such as mining and mineral analysis. Health applications involve tracking patients, monitoring drug administrations in hospitals. Great commercial opportunities exist in the household electronics and in realizing the smart home and office environments.

2.3 Sensor Network Platforms

Sensor platforms tend to be simple and relatively inexpensive so that they can be deployed in large numbers. An extremely small form factor is also preferred for many applications to prevent node destruction and tampering. In fact, the Smart Dust project at the University of California, Berkeley, seeks to develop wireless sensor nodes [13], to

include sensor, processor, radio, and battery, that are one cubic millimeter in size. Many deployment scenarios, particularly military ones, have the devices being abandoned once their power supplies are exhausted. Three important goals, therefore, are to keep the devices small, inexpensive, and as energy-efficient as possible. One cannot expect Moore's Law to lead to enhanced performance. Rather, technological improvements are likely to drive down size and cost while maintaining current capabilities.

Several sensor platforms have been developed in the last few years. Appendix A gives a listing of currently available platforms. These devices are characterized by 8-bit processors, around 100 kilobytes of program memory, and 2 to 10 kilobytes of random access memory (RAM).

On the other hand, there are some devices which have more memory. The amount of RAM affects energy consumption, because the RAM requires constant current to maintain stored data. The data stored in RAM must be maintained while the devices are in the sleep state, increased RAM results in an increase in sleep-state energy consumption and, therefore, reduces network lifetime.

2.4 Sensor Node Technology

One of the most widely used platforms for researching wireless sensor networks with limited resources is the Berkeley MICA mote, designed at the University of California, Berkeley [13]. The MICA mote has a 4MHz microcontroller, 4KB of RAM, 128KB of flash memory and a 916 MHz wireless radio transceiver with a transfer rate of 19.2 Kbps and range of 200 feet. It has a very small form factor (58mm X 32mm X 7mm) and is powered by two AA batteries. Daughtercards with various sensors and actuators are

available, including photo, temperature, humidity, infrared and barometric pressure sensors, accelerometers, magnetometers, microphones, and sounders.

There are different kinds of sensors:

- **The Berkeley MICA motes**

The Berkeley MICA motes run the TinyOS operating system, an open source, event driven and modular OS designed to be used with networked sensors. A TinyOS application is a statically compiled graph of components. Components have memory frames to store their state, and communicate with each other through used and provided interfaces that contain logically related commands and events. Components can post tasks to process longer running computations, which are executed in order by the scheduler. TinyOS comes with a library of OS components that handle task scheduling, radio communication, clocks and timers, ADC, I/O and EEPROM abstractions, and power management. Application developers can select a subset of these modules, extend or override them if necessary, and statically compile them into the final executable. A typical MICA system consists of tens to hundreds of motes forming an ad hoc multi-hop network and a base station that is typically known as a PC class computer.

- **Stargate Sensor**

Stargate is a powerful single board sensor with communications and sensor signal processing capabilities. The Stargate uses Intel's® latest generation 400MHz X-Scale® processor (PXA255) [14]. In addition to traditional single board computer applications, the Stargate directly supports applications designed around Intel's Open-Source Robotics initiative as well as TinyOS-based wireless sensor networks. Another powerful, yet compact sensor node is the OpenBrick device. It is small (220 x 165 x 42 mm), light

(about 1200 g), and includes three (3) RJ45 LAN connectors. It has a fan-less 533 Mhz x86 compatible VIA C3 processor and 256 MB SDRAM. Software can be installed on a Compact Flash or on a Hard Disk. It also has USB 2.0 connectivity. The device supports the Linux operating system.

- **Sensoria WINS NG Sensor**

This sensor node is designed by Sensoria Corporation, Rockwell Collins, and the Army Research Laboratory (ARL) [14]. Both the Rockwell Science Center and Sensoria Corporation have a Wireless Integrated Network Sensor node suitable for low-energy sensor networks. The Sensoria node uses a low-power pre-processor, the Zilog Z-180, and a low-power core processor, MIPS R4400, for performing signal processing functions. The Rockwell node uses a low-power Intel StrongARM SA-1110 microprocessor. Both the MIPS at 80 MHz and StrongARM at 133 MHz typically consume less than 300 mW when in the run mode. The StrongARM consumes less than 1mW in its sleep mode. the Sensoria WINS NG uses the Microsoft Windows CE operating system.

CHAPTER 3

SECURITY AND ECC

A Wireless Sensor Network consists of sensor nodes that capturing information from an environment, processing data and transmitting them via radio signals. WSNs expected to support a wide variety of applications, many of which have at least some requirements for security.

Security requirements of WSNs are similar to conventional computer networks, therefore parameters such as confidentiality, integrity, availability and authenticity must be taken into account in creation of a network environment. Due to limitations of WSNs (low computational power, limited data transmission and power Constraints), e.g. most sensor hardware used is based around a battery powered microcontroller like Atmels ATMEGA128 which has only 4KBytes of main memory (RAM), 4KByte of EEPROM and 128KByte of ROM, not all security solutions designed for conventional computer networks can be implemented directly in WSN. For a long time, it was believed that the public key cryptography was not suitable for WSNs because it was required high processing power, but through studies of encryption algorithms based on curves was verified the feasibility of that technique in WSN.

3.1 Security Requirements

In most of applications, sensor devices are spread over large areas, what difficult a individual control of network components. Moreover, wireless communication allows an

attacker can trigger attacks without having physical access to the device. Attacks on WSNs can be divided into three main types [15] such as:

- Attack of authentication and confidentiality: Consists of attacks who change, repeat or modify packages.
- Availability network Attack: Generally known as Denial of Service (DoS) attacks or negation of service, this attack involves the application of techniques that make the network unavailable.
- Attack on integrity: this type of attack the attacker's goal is to inject false data on the network, keeping the network available, but traveling fictitious data.

The security of WSNs can be classified into two broad categories: Operational security and Information security.

The operation-related security objective is that a network, as a whole, should continue to function even when some of its components are attacked (the service availability requirement).

The information-related security objectives are that confidential information should never be disclosed, the integrity and authenticity of information should always be assured.

Table 3.1 shows the security requirements for some application with potential security threats. The objectives are marked with a cross in Table 3.1 if they are violated in the corresponding scenario [16].

S=Service Availability, C=Confidentiality, I=Integrity, A=Authenticity

Table 3.1. Security requirements for some application with Potential security threats.

Application domain	Potential security threats	Properties violated			
		S	C	I	A
Military	Denial-of-service attacks	×		×	
	Eavesdropping of classified information		×		
	Supply of misleading information				×
Disaster Detection	Supply of misleading information				×
Environmental Monitoring	Suppose government-endorsed environmental sensors are installed near a factory to monitor air/water quality to make sure the factory's emission lies beneath the pollution threshold, however by feeding the sensors with wrong information, the factory allows itself to escape detection and let its polluting emission go unchecked.			×	
Intelligent Buildings	Biometrics-based access control mechanisms are compromisable if the biometric sensors can be bypassed or fooled.	×		×	
	Token-based access control mechanisms are compromisable if the token authentication protocol is insecure.				×
Health/Medical	Providing wrong physiological measurements of a patient to the career or doctor, a miscreant may cause potentially fatal diagnosis and treatment to be performed on the patient.			×	×
Transportation	There is no order in the city when traffic information can no longer be trusted because they can easily be spoofed.			×	×
Industry	Eavesdropping of commercial secrets by business rivals.		×		
	Intentional disruption of manufacturing processes as a result of misleading sensor readings caused by disgruntled employees or business spies.	×		×	
Space exploration	Space agencies invest billions into space exploration projects, it is only logical that they want to ensure all commands executed on their space probes are authorized, and all collected data encrypted and authenticated.	×	×	×	×

While it may seem that information security can readily be achieved with cryptography, there are two facts that make achieving the above objectives non-trivial in WSNs:

- As sensor nodes operate unattended they are potentially accessible, both geographically and physically, to any malicious party imaginable.
- Sensor nodes communicate through an open medium.

3.1.1 Confidentiality

Confidentiality means keeping information secret from unauthorized parties. A sensor network should not leak sensor readings to neighboring networks. The goal of confidentiality is required in sensors environment to protect information traveling between the sensor nodes of the network or between the sensors and the base station from disclosure, because an adversary may have an appropriate equipment which eavesdrop on the communication. In this case, the adversary overhears critical information such as sensing data and routing information [15].

3.1.2 Authentication

In a sensor network, an adversary can easily inject messages, so the receiver needs to make sure that the data used in any decision-making process are originated from the correct source.

As in conventional systems, authentication techniques verify the identity of the participants in a communication, distinguishing in this way legitimate users from intruders. In the case of sensor networks, it is essential for each sensor node and base

station to have the ability to verify that the data received was really sent by a trusted sender and not by an adversary that will use tricked legitimate nodes to make the receiver accepting false data. If such a case happens and false data are supplied into the network, then its behavior could not be predicted, and most of times the mission of WSN will not be accomplished as expected [15].

3.1.3 Integrity

Data integrity ensures the receiver that the received data is not altered in transit by an adversary. Lack of integrity could result in many problems since the consequences of using inaccurate information could be disastrous, for example, when treating with a patient, the measuring equipment should work property because if it gives wrong data then the life of the patient will be in danger. Integrity controls must be implemented to ensure that information is not altered in any unexpected way [16].

3.1.4 Freshness

One of the many attacks launched against sensor networks is the message replay attack where an adversary may capture messages exchanged between nodes and replay them later to cause confusion to the network. Data freshness implies that the data is recent, and it ensures that an adversary has not replied old messages [16].

To achieve freshness, network protocols must be designed in a way to identify duplicate packets and discard them to prevent potential mix-up.

3.1.5 Availability

Availability ensures that services and information can be accessed at the time they are required. In sensor networks there are many risks that could result in loss of availability such as sensor node capturing and denial of service attacks.

The availability of a sensor and sensor network may decrease for the following reasons:

Additional computation consumes additional energy. If no more energy exists, the data will no longer be available.

Additional communication also consumes more energy. Besides, as communication power increases the chance of a communication conflict or interference will increase.

A single point failure exists if we use the central point scheme such as a single sink or gateway. This greatly threatens the availability of the network [15].

3.1.6 Secure Management

Management is required in every system that is constituted of multi components, and handles sensitive information. In the case of sensor networks, we need secure management on base station level; since sensor nodes communication ends up at the base station, issues like key distribution to sensor nodes in order to establish encryption and routing information need secure management.

3.2 Defense Mechanisms

Different types of WSN applications require different security requirements. In an environment of temperature monitoring, where researchers collect data for research, it

may be that safety requirements are not very important, but the monitoring of radiation near a nuclear power plant requires authenticity assurance, confidentiality, availability and integrity. Various architectures have been developed to provide security in WSNs, among them are:

SPINS, TinySec MiniSec and the IEEE 802.15.4 include a security framework to meet the services of data integrity, confidentiality and authenticity.

SPINS (Security Protocols for Sensor Networks) developed by Perrig [17] consists of a set of security protocols that acts through encryption and message authentication codes.

The TinySec was designed and implemented in the TinyOS operating system to be a mechanism for providing confidentiality, integrity and authenticity of the data link layer. It uses the CBC mode of operation that may be combined with various block ciphers as RC5 and skipjack [18].

The MiniSec is a protocol layer of security to WSN using OCB (Offset Codebook) mode for operating the block cipher, which eliminates the need of adding filler to the clear text blocks [19].

The standard IEEE 802.15.4 provides integrity, access control, confidentiality and replay protection in the link layer. The cryptographic algorithm used in this standard is AES [20].

3.3 Cryptographic Algorithms

Most existing cryptographic algorithms can be classified as symmetric or asymmetric.

3.3.1 Symmetric Encryption

Symmetric encryption or secret key cryptography is the use of only a key, both in the encryption and decryption of data. By the year 1976 this was the only known method for the use of encryption [21], but to be effective you need a secure channel for communication in which a cryptographic key can be changed.

Figure 3.1 illustrates a communication through symmetric encryption. The text X is encrypted with the key k and sent to the receiver, which uses the key k to decrypt it. The key k is transported by a secure channel, for the possessing it safety. Because if any attacker gets it then the original text could be read easily by the attacker. AES and DES are two examples of algorithms that are parts of the class symmetrical.

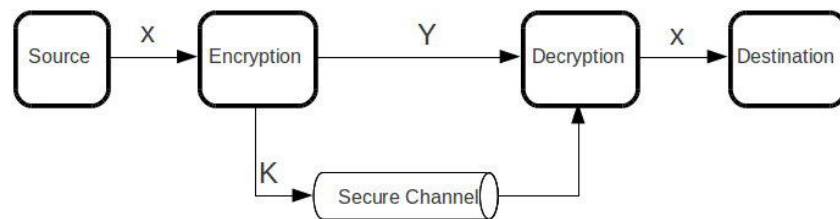


Figure 3.1: Symmetric Cryptography

3.3.2 Asymmetric Encryption

The public key cryptography or asymmetric cryptography came up with a radical change of paradigms. The public key algorithms are based on mathematical functions, instead of permutation and substitution [21]. Besides the single most important thing is that the public key cryptography is asymmetric, involving the use of two different keys, in

contrast to the conventional symmetric encryption, which uses only one key. The use of two keys has profound consequences in the areas of confidentiality, key distribution and authentication. The main distinguishing feature of asymmetric encryption is that it allows the establishment of a secure communication between individuals, without the requirement of the previous share a single cryptographic key.

In this class of cryptographic algorithms are used two different keys for encryption and decryption: a public key and its corresponding private key. In this model, in accordance with Figure 3.2, the receiver releases its public key to the sender can encrypt the message, but only the private key of the receiver, which is kept secret is able to decrypt it.

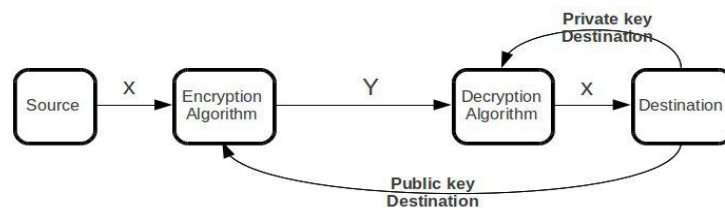


Figure 3.2: Asymmetric Cryptography

3.4 Elliptic Curve Cryptography

Elliptic curves are an algebraic structure whose use for cryptography. In 1985, Neal Koblitz [22] and V.S. Miller [23] independently proposed using elliptic curves for public key cryptosystems. They did not invent a new cryptographic algorithm with elliptic curves over finite fields, but they implemented existing algorithms, like Diffie-Hellman, using elliptic curves [21].

The magic of elliptic curves comes from the ability to take any two points on a specific curve, add them together, and get another point on the same curve. More importantly for

cryptography is the difficulty of figuring out which two points were added together to get there. For the right choice of various parameters, that difficulty is exponential with key length. While the cryptanalyst must use very advanced mathematics to even begin attempting to crack a code.

3.4.1 Mathematics of Elliptic Curves over Real Numbers

The elliptic curve is a set points over a field that satisfies a certain equation. First, let's start with what mathematicians call the “Weierstrass” form of an elliptic curve equation[24]:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \quad a_1, a_2, a_3, a_4, a_6 \in \mathbb{F} \quad (3.1)$$

Here a_i are the parameters of the curve. The variables x and y cover a plane. In fact, x and y can be complex, real, integers, polynomial basis, optimal normal basis, or any other kind of field element. That's part of what makes the math so deep.

Let's take the sample form of equation 3.1.

$$y^2 = x^3 + a_4x + a_6 \quad (3.2)$$

Plot the curve for $a_4 = -7$, $a_6 = 5$. and see where the points P and $-P$ lie on the curve is shown in Figure 3.3.

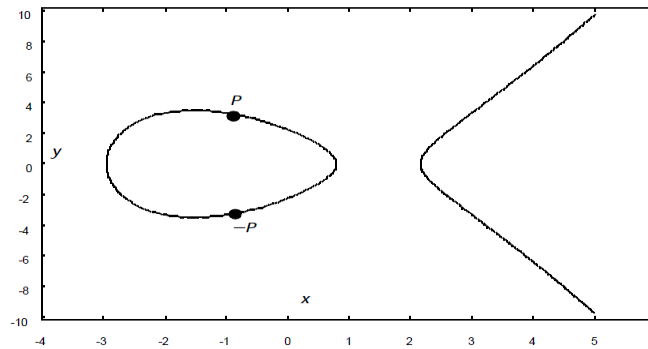


Figure 3.3: Plot of elliptic curve $y^2 = x^3 - 7x + 5$

The two points lie on top of each other. Now, let's do another obvious thing;

$$P + (-P) = \mathbf{O} \quad (3.3)$$

Where $\mathbf{O}$ the identity element. If a line is drawn between P and $-P$, the next point on the curve it hits is “the point at infinity.” In fact, The addition of any two points on an elliptic curve is defined by drawing a line between the two points and finding the point at which the line intersects the curve.

The negative of the intersection point is defined as the “elliptic sum”, this is shown in Figure 3.4.

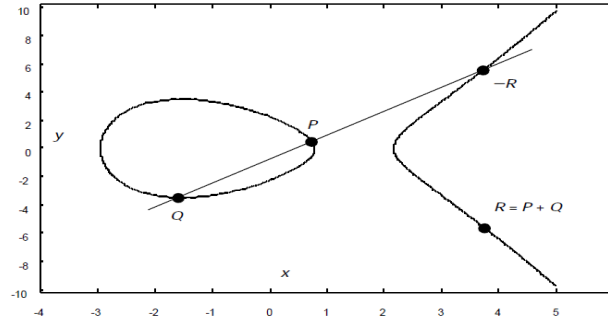


Figure 3.4: Addition of elliptic curve points over a real number curve

If $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ then:

$$R = P + Q = (x_3, y_3) \quad (3.4)$$

Where

$$x_3 = \theta^2 - x_1 - x_2 \quad (3.5)$$

$$y_3 = \theta(x_1 + x_3) - y_1 \quad (3.6)$$

$$\theta = \frac{y_2 - y_1}{x_2 - x_1} \quad \text{if } P \neq Q \quad (3.7)$$

or

$$\theta = \frac{3x_1^2 + a_4}{2y_1} \quad \text{if } P = Q \quad (3.8)$$

Adding a point to itself is a special case, also called “doubling”; the line used is the tangent to the curve at P . The Figure 3.5 illustrates this idea.

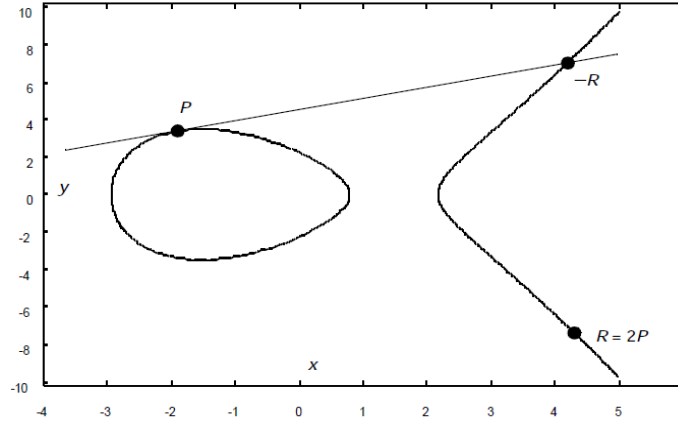


Figure 3.5: Addition of a point to itself.

3.4.2 Mathematics of Elliptic Curves over Galois Fields

The elliptic curves over F_2^n means that the constants are either polynomial or normal basis numbers. The version of equation 3.1 that used is [25]:

$$y^2 + xy = x^3 + a_2x^2 + a_6 \quad (3.9)$$

or this version:

$$y^2 + y = x^3 + a_4x + a_6 \quad (3.10)$$

The second form, equation 3.10, is called a “supersingular” curve. These forms have the advantage that they can be computed quickly. These properties make supersingular curves unsuitable for cryptography.

The curves of equation 3.9 are called “nonsupersingular”. Curves of this form are excellent for cryptographic applications. One must be careful in choosing the coefficients

to get maximum benefit of security. A poor choice can create a curve that is easier for the cryptanalyst to attack.

For equation 3.9 to be valid, a_6 must never be 0. However, a_2 can be 0. The rules are the same as before: Take any two points on the curve; draw a line between them; and the negative of the third point, which intersects both the curve and the line, is the “sum” of the first two points.

The rule to negate a point $P = (x,y)$ is:

$$-P = (x, y + x). \quad (3.11)$$

Because x and y are variables in the field, the $+$ operation is just an exclusive-or of all the bits of x with all the bits of y .

The rules for adding two points over F_2^n can be stated as follows [24]:

If $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ then:

$$R = P + Q = (x_3, y_3)$$

if $P \neq Q$:

$$\theta = \frac{y_2 - y_1}{x_2 - x_1} \quad (3.12)$$

$$x_3 = \theta^2 + \theta + x_1 + x_2 + a_2 \quad (3.13)$$

$$y_3 = \theta(x_1 + x_3) - y_1 \quad (3.14)$$

if $P = Q$:

$$\theta = x + \frac{y}{x} \quad (3.15)$$

$$x_3 = \theta^2 + \theta + a_2 \quad (3.16)$$

$$y_3 = x^2 + (\theta + 1) x_3 \quad (3.17)$$

3.4.3 Elliptic Curve Discrete Logarithm Problem

ECC schemes are public key mechanisms that provide the same functionality as RSA schemes. However, their security is based on the hardness of the different problem, namely Discrete Logarithm Problem (DLP).

The elliptic curve discrete logarithm problem can be stated as follows. Let P and Q be two points on an elliptic curve that $KP = Q$, where K is a scalar. Given P and Q , it is computationally infeasible to obtain K , if K is sufficiently large. K is the discrete logarithm of Q to the base P [25,26].

Hence, the main operation involved in ECC is point multiplication, i.e., multiplication of a scalar K with any point P on the curve to obtain another point Q on the same curve.

The security of the ECC rests on the difficulty of the elliptic curve discrete logarithm problem. Currently the best algorithm known to solve the Elliptic Curve Discrete Logarithm Problem have fully exponential running Time, in contrast to the sub exponential-time algorithms known for the integer factorization problem.

In this case, this means that the desired security level can be attained with significantly smaller keys in elliptic curve systems in a way better than RSA counterparts. For example: it is generally accepted that a 160-bit elliptic curve key provides the same level of security as a 1024-bit RSA key. One of the advantages of ECC is that the elliptic curve discrete logarithm problem is believed to be harder than both the integer factorization problem and discrete logarithm problem modulo p . This extra difficulty implies that ECC is one of the strongest public key cryptographic system known today.

However, the elliptic curve discrete logarithm problem is relatively easy for a small class of elliptic curves, known as supersingular elliptic curves and also for certain anomalous elliptic curves. In both cases, the weak instances of the problem are easily identified, and an implementation merely checks that the specific instance selected is not one of the class of easy problems.

CHAPTER 4

FUNDAMENTALS OF ELLIPTIC CURVE CRYPTOGRAPHY

When the implementing of elliptic curve crypto systems are needed, the first choice is the basis: either polynomial or normal. The tradeoff between them is a tradeoff between speed (polynomial) or space (normal). Polynomial basis gives more speed (low execution time) but takes twice as much RAM, because multiplication double the number of bits stored before being reduced modulo the basis function. Normal basis uses minimal RAM but takes longer to compute, because the inversion takes more time.

The normal basis is better suited for sensor network where memory consumption is critical, then the normal basis is chosen.

4.1 A Normal Basis

The mathematical symbols of normal basis may seem overly complex, but the implementation in computer hardware or software is very easy. Only AND, XOR, and ROTATE operations are needed. So, these are the fastest operations possible on any microprocessor and this is what makes optimal normal basis (ONB) so attractive. There are two types of normal basis that are useful for Elliptic Curve Cryptography. They are called Type I and Type II.

In a normal basis, the elements in a field are described using a polynomial [25]. If some element β in the field F_p^m is taken, the polynomial representation is:

$$\beta = a_n x^n + \dots + a_1 x + a_0 \quad (4.1)$$

where $n < m$, m is the field size

A normal basis can be formed using the set:

$$\{\beta^{p^{m-1}}, \dots, \beta^{p^2}, \beta^p, \beta\}. \quad (4.2)$$

An element, e , can be written in a normal basis with different operations as (for computers, use $p=2$):

$$e = e_{m-1}\beta^{2^{m-1}} + \dots + e_2\beta^{2^2} + e_1\beta^2 + e_0\beta \quad (4.3)$$

or

$$e = \sum_{i=0}^{m-1} e_i \beta^{2^i} \quad (4.4)$$

4.1.1 Squaring Normal Basis Number

For follow reasons the squaring a number in normal basis is just a rotation [25].

$$1. \left(\beta^{2^i}\right)^2 = \beta^{2^{i+1}} \quad (4.5)$$

$$2. \beta^{2^m} = \beta \quad (4.6)$$

So, squaring is very fast in a normal basis.

4.1.2 Multiplication

Multiplication over a normal basis is the same in any mathematical system, multiply coefficients and sum. The multiplication of two different elements A, B in F_{2^m} [27]:

$$C = A.B = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_i b_j \beta^{2^i} \beta^{2^j} \quad (4.7)$$

$$\beta^{2^i} \beta^{2^j} = \sum_{k=0}^{m-1} \lambda_{ijk} \beta^{2^k} \quad (4.8)$$

The λ_{ijk} coefficient is called “ the lambda matrix” or “multiplication table”

For a derivation of the following [27].

$$C_k = \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} a_{i+k} b_{j+k} \lambda_{ij0} \quad (4.9)$$

An “optimal” normal basis has the minimum number of nonzero λ_{ij} terms. This number is called the “complexity” of the multiplication table. For field F_{2^m} the optimal (minimum) complexity is $2m-1$.

The only difference between Type I and Type II is the way that find which bits in the λ matrix are set. For Type I ONB only one vector to store is needed; for Type II ONB two vectors to store is needed.

4.2 ONB mathematics and subroutines

In code there are some routines for generalized and variable length ONB mathematics:

- initialization of Type I and Type II

The `genlambda( )` used to generate λ matrix, and log table, `lg2_m`, for Type I. on the other hand, `genlambda2( )` generate λ matrix, and log table for Type II .

- Rotate operations (rot_left, rot_right).

In the rot_left() and rot_right () routines the bits are rotated left or right.

- Multiplication (opt_mul).
- Inversion (opt_inv).

In the following sections all the subroutines above and the theory behind normal basis multiplication and inversion will discuss. Figure 4.1 shows the subroutine schematic 1 [25] that illustrates ONB mathematics:

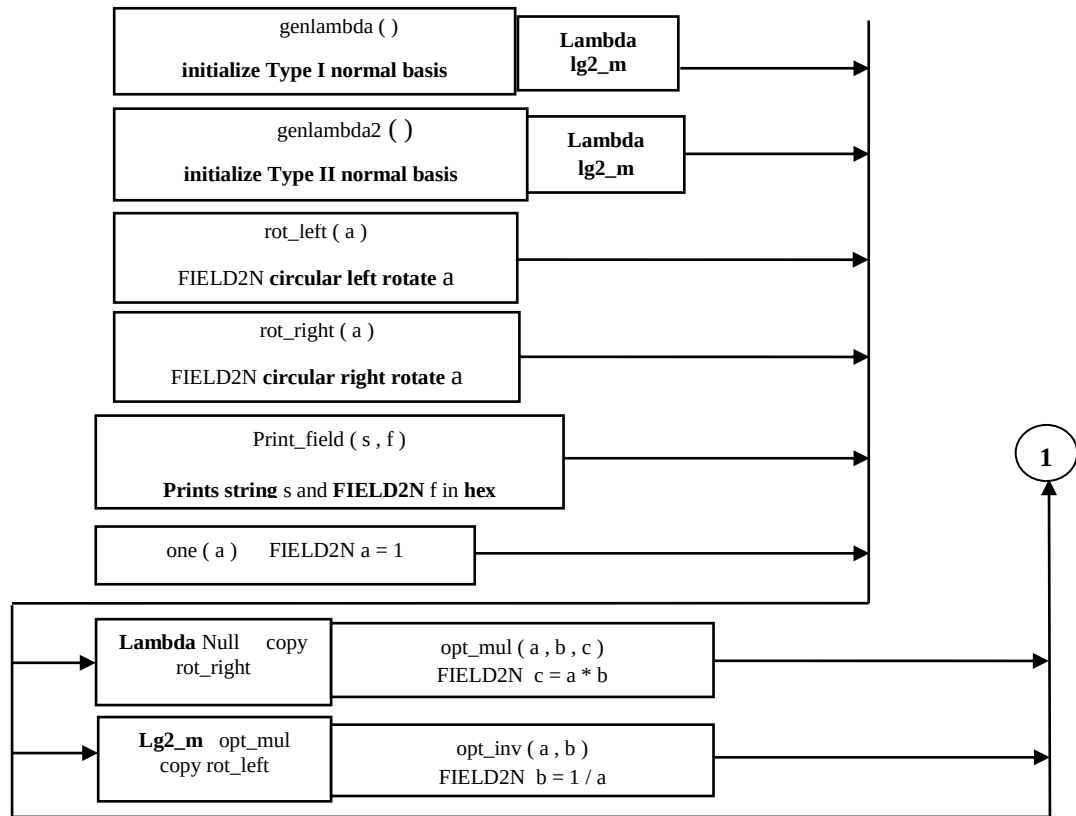


Figure 4.1: Subroutine Schematic 1

4.2.1 Type I Optimal Normal Basis

The rules for creating Type I ONB over the field F_{2^m} are:

1. $m+1$ must be prime.
2. 2 must be primitive in Z_{m+1} . (Z mean the set of integers)

Rules 2 means that 2 raised to any power in the range $0 \dots m-1$ modulo $m+1$ must result in a unique integer in the range $1 \dots m$.

Find the cross-product terms of $\beta^{2^i} \cdot \beta^{2^j}$ in equation 4.8 is needed to make the multiplication work. Solve the equation 4.8 [25]:

$$2^i + 2^j = 1 \mod m+1 \quad (4.10)$$

$$2^i + 2^j = 0 \mod m+1 \quad (4.11)$$

Let's start with $i=0$. $2^0 = 1$ and $2^m = 1$, since $m+1$ is prime. This last point comes from Fermat's Theorem [24]. The equation 4.10 cannot have a solution for $i=0$. Only equation 4.11 can. If the square root of $2^m = 1$, is taken:

$$2^{m/2} = \pm 1 \mod m+1 \quad (4.12)$$

But $2^0 = +1$, since 2 generates all the numbers mod $m+1$, equation 4.12 has only one choice:

$$2^{m/2} = -1 \mod m+1 \quad (4.13)$$

equation 4.11 thus has a solution for $i=0$, which is:

$$2^0 + 2^{m/2} = 0 \mod m+1 \quad (4.14)$$

Type I ONB needs a single vector to keep track of all the cross-product terms. The first entry in the table is at offset 0 and has value $m/2$. Don't need to store any more of the values to solutions of equation 4.11, because if multiply equation 4.14 by 2 the right-hand side is still equal 0. For every i , the nonzero λ_{ij} element from equation 4.11 will always be:

$$j = m/2 + i \bmod m . \quad (4.15)$$

Tabulate the value from equation 4.10 have done once and then use those to look up the correct shift amounts. The first value for $i=0$ is easy, since:

$$2 + 2^j = 1 \quad (4.16)$$

$$2^j = -1$$

So, $j = m/2$ from equation 4.13.

After that use antilog and log tables are needed, so $2^i \bmod m+1$ that found easily and j from $1 - 2^i \bmod m+1$. for example the antilog table for $m = 4$ Table 4.1 [25], multiply i by 2 modulo 5 for each entry.

Table 4.1 Antilog Table for 2^i for $m=4$

i	0	1	2	3	4
2^i	1	2	4	3	1

The log table, Table 4.2, takes each value of 2^i as the index and places as i the entry.

Table 4.2 Log Table for 2^i for $m=4$

2^i	1	2	3	4
i	0	1	3	2

To generation code for the single vector two things that needed. The first is a construction of log tables for $2^i \bmod m+1$. The second step is to create a lambda vector, which stores all the values of j for each value of i that satisfies equation 4.10. Because the lambda matrix is symmetric, only do half of it.

4.2.2 Type II Optimal Normal Basis

For a Type II optimal normal basis there are the same number of terms. A Type II optimal normal basis over F_{2^m} can be created if [25]:

1. $2m+1$ is prime

and either

2a. 2 is primitive in Z_{2m+1}

or

2b. $2m+1 \equiv 3 \pmod{4}$ and 2 generates the quadratic residues in Z_{2m+1} .

To generate a Type II ONB two field elements from two different fields can be used. First pick an element γ of order $2m+1$ in $F_{2^{2m}}$, that used to find β , which is in the field F_{2^m} . The element γ used to help create the λ matrix. From the sum of $\gamma + \gamma^{-1}$. This element gives the first element, β , of normal basis [27].

The cross-product terms of $\beta^{2^i} \beta^{2^j}$ are:

$$\beta^{2^i} \cdot \beta^{2^j} = (\gamma^{2^i} + \gamma^{-2^i})(\gamma^{2^j} + \gamma^{-2^j}) \quad (4.17)$$

By solving the above equation resulted the set of the following equations that used to build λ matrix:

$$2^i + 2^j = 1 \quad (4.18)$$

$$2^i + 2^j = -1 \quad (4.19)$$

$$2^i - 2^j = 1 \quad (4.20)$$

$$2^i - 2^j = -1 \quad (4.21)$$

This example to illustrate build the log and antilog tables. If $m=9$ (the length of the λ matrix) the antilog table, Table 4.3, takes an index, i , and returns $l = 2^i \bmod 2m+1$.

Table 4.3: Power of $2^i \bmod 19$ (antilog table)

I	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
2^i	1	2	4	8	16	13	7	14	9	18	17	15	11	3	6	12	5	10	1

The log table, Table 4.4, takes an index, l , and returns the value of i [25].

Table 4.4: Log base 2 of $i \bmod 19$ (log table)

I	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
$\log_2(i)$	0	1	13	2	16	14	6	3	8	17	12	15	5	7	11	4	10	9

The log table is built using $2^i \bmod m+1$ as the subscript, and the loop counter i is the value. To continue the example, let's start with $i=1$ in equations 4.18 through 4.21.

Writing down all four equations mod 19 and subtracting 2 from both sides gives the following:

$$2^j = -1 = 18 \Rightarrow j = 9$$

$$2^j = -3 = 16 \Rightarrow j = 4$$

$$-2^j = -1 \text{ is } 2^j = 1 \Rightarrow j = 0$$

$$-2^j = -3 \text{ is } 2^j = 3 \Rightarrow j = 13$$

Since the λ matrix has only nine ($m=9$) entries per column, the solutions for j must be in the range of $0 \dots 8$. Only two terms for j are less than 9. These are the two terms that needed. So the first nonzero entries in the λ matrix is: $\lambda_{1,0} = 1$ and $\lambda_{1,4} = 1$. All other entries, $\lambda_{1,j}$, must be 0. Continuing in this manner the two values of j for each value of i could be find, which give nonzero entries in the λ matrix. These values which marked with two vectors: Λ_0 and Λ_1 (in the code `Lambda[0][ ]` and `Lambda[1][ ]` respectively). Each position in the Λ table corresponds to a value of i in cross-product term $\beta^{2^i} \beta^{2^j}$, and each entry is the matching value of j . The results are shown in Table 4.5 [25].

Table 4.5 Λ vectors for $m=9$

i	Λ_0	Λ_1
0	1	
1	4	0
2	4	7
3	6	8
4	2	1
5	6	5
6	5	3
7	2	5
8	8	3

One key point to save main memory is moving all larger unchangeable data (λ matrix) from RAM to ROM which is generally supported on sensor node. Using a special compiler directive ROM as a modifier to data declarations forces compiler, linker and finally the device programmer to place this data into flash-ROM regions next to ordinary program code. Later on small fragments of this region can be copied temporarily from ROM to RAM using the `memcpy` command.

The multiplication table λ_{ij0} which does not change during sensor lifetime. It was therefore very reasonable to:

1. precompute this table offline and distribute it to every sensor node prior to deployment
and
2. move λ_{ij0} out of valuable main memory to (flash-)ROM.

4.2.3 Multiplication Routine (`opt_mul`)

From equation 4.9 note that both subscripts a and b are shifted by the same value of k. this means that all the a coefficients and all the b coefficients can be shifted for any particular values of i and j to find one term for all the c coefficients. B is shifted once for each term and use the count of that offset as the index into the lambda vector table to find each proper shift of A. Also the bits are rotated actually. Going right(**rot_right routine**) the least significant bit needs to be placed into the most significant bit. On the contrary going left (**rot-left routine**) should be done. The first routine is actually a squaring operation, and the second routine is a square root operation in a normal basis.

For every sum two shifts of A are needed. So it is faster to compute all the shifts once and store them in a lookup table.

So, the basic idea of the multiply is really simple: Look up the two terms which are needed from the shifted table, XOR them, then AND that with the present B vector. XOR this term with the C vector as a partial sum, shift the B vector and repeat until all n partial terms have been summed with the result.

However, for a Type I ONB, there is no need for two lookup vectors. The only thing is which needed to perform the multiplication is (AND) of A shifted half its length with B and take the “trace” of the result, which gets summed with the single lookup vector result.

4.2.4 Inversion Routine (opt_inv)

There are several methods of computing inversions over normal basis.

The fundamental constant, 1, is represented as all bits set in a normal basis [25].

$$1 = \sum_{i=0}^{m-1} \beta^{2^i} = \beta + \beta^2 + \beta^4 + \dots + \beta^{2^{m-1}} \quad (4.22)$$

To compute an inverse, the rules are the same as normal math: $a \cdot a^{-1} = 1$. But “1” looks different. The first trick is to use Fermat’s Theorem [24]:

$$a^{-1} = a^{2^m-2}. \quad (4.23)$$

Equation 4.23 can be solved and transformed to [25]:

$$\left[\left(a^{2^{\frac{r_0-1}{2}-1}} \right)^{2^{\frac{r_0-1}{2}+1}} \right]^2 . a . \quad \text{where } m-1 = r_0 \quad (4.24)$$

If r_0 is odd, this means an additional squaring operation (rotation) and an extra multiply . Let $r_1 = r_0/2$. This tosses the last bit. The same expansion with r_1 can be done as did with r_0 . This works down a chain of multiples and squaring until get to r_l , where l is the number of bits in m . at that point, $r_l=1$, and the innermost term is:

$$a^{2^{1-1}} = a \quad (4.25)$$

The binary representation of m is:

$$m = \sum_{k=0}^l m_k 2^k = m_l 2^l + m_{l-1} 2^{l-1} + \dots + m_0 \quad (4.26)$$

Now each r_s in expansion formula is:

$$r_s = m_l 2^{l-s} + m_{l-1} 2^{l-1-s} + \dots + m_s. \quad (4.27)$$

In the inversion routine (**opt_inv**), the sliding window over the bit pattern from the first bit to the last should be done. Find $a^{2^{r_s-1}}$ from the bit pattern of r_{s+1} . If the last bit of r_{s+1} is set, an extra squaring and multiply are needed as in equation 4.24. If the last bit is clear, rotate the previous result by r_{s+1} and multiply. The variable r holds the sliding window values of the binary representation of powers, as in equation 4.27.

There are specific values of number bits for which an optimal normal basis exists. Appendix B shows a few values taken from an appendix of the IEEE P1363.

4.3 Optimal Normal Basis Elliptic Curve Subroutines

The Subroutine Schematic 2 is shown in Figure 4.2 illustrates implement the elliptic curve math [25]:

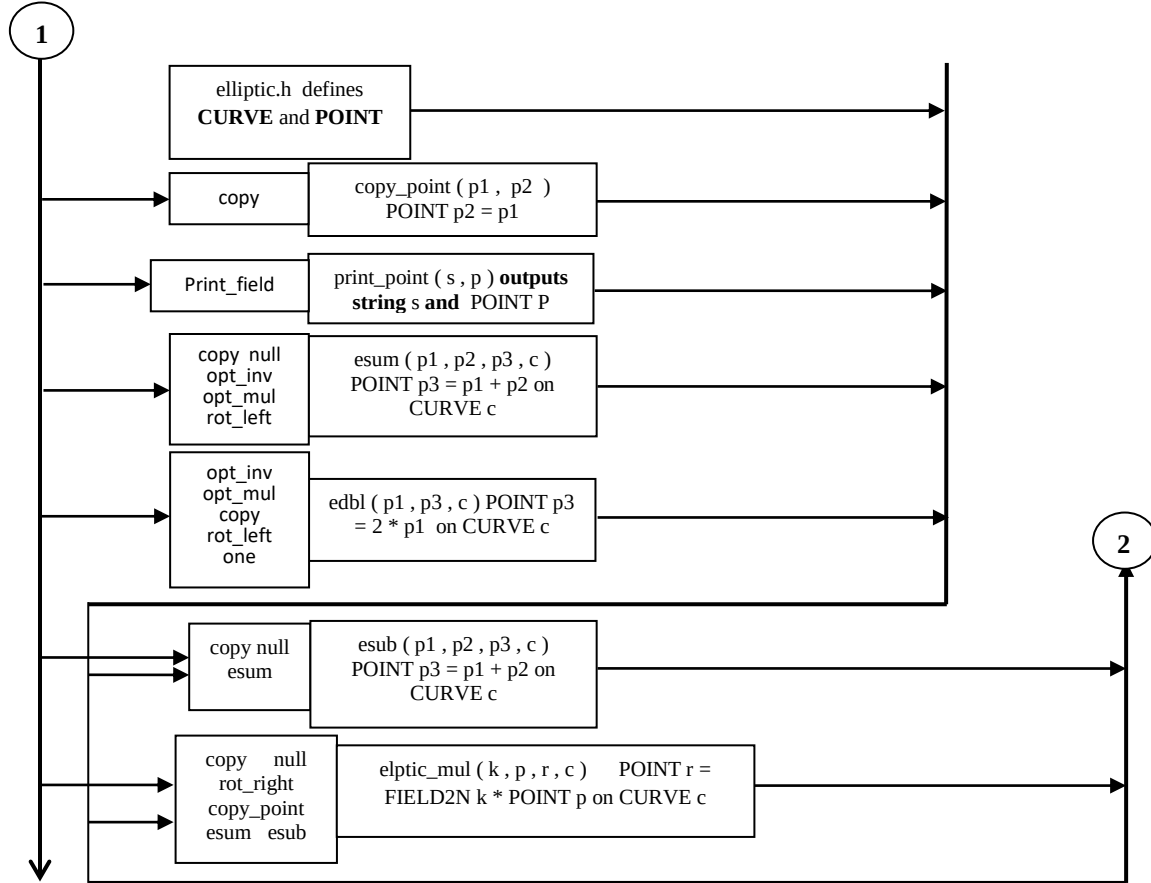


Figure 4.2: Subroutine Schematic 2

There are some routines used to create elliptic curve points and parameters and math operations for ONB:

- definitions for points and curves.

A points is defined as an (x,y) pair over the field .

A curve is defined from equation 3.14 in the previous chapter.

The choice of a_2 and a_6 does have cryptographic implications.

- Math operations.

1. Point Addition

The equations 3.12 through 3.14 are implemented to adding two points using optimal normal basis field elements.

2. Point Doubling

The equations 3.15 through 3.17 are implemented to double a point.

3. Point Subtracting

To subtract two points negate the second point and add it to the first point.

The equation 3.11 is implemented to negate the point.

4.3.1 Multiplication over Elliptic Curves

The idea of multiplying points refers to computing:

$$Q = k P \tag{4.28}$$

Where Q and P are points on an elliptic curve and k is an integer. To perform an elliptic multiply, the algorithm comes from [19] is used. The first assumption is converted the integer number k to the same number of bits. There are many ways to do this. The doubling and add method is used which Koblitz calls it a “balanced” expansion.

4.3.1.1 Balanced Representation

To create the balanced representation the simple way is used. This method is called

“Algorithm 2” in that paper [29]:

INPUT n .

OUTPUT balanced array S .

1. Set $k \leftarrow n$, $s \leftarrow \emptyset$
2. While $k > 0$
 - if k is odd
 - Set $u \leftarrow 2 - (k + 3)$
 - if $u < 0$, increment k
 - else
 - set $u \leftarrow 0$
3. prepend u to S
4. set $k \leftarrow k/2$

4.4 Choosing an Elliptic Curve

The beginning of this chapter talked about the first choice when the implementing of elliptic curve crypto systems are needed. The second choice is a curve. The supersingular curves unsuitable for cryptography because they can be computed quickly.

4.4.1 Nonsupersingular Curves

The nonsupersingular curves are excellent for cryptographic applications. These curves have the form in equation 3.9. The specific values of a_6 can give maximum strength curves. If a_2 is 0, then the calculations are a faster. When a_2 is nonzero, the curve is called a “twist” . An example from [28] is:

$$y^2 + xy = x^3 + x^2 + 1 \quad (4.29)$$

where $a_2=1$ and $a_6=1$, field sizes 101, 107, 109, 113, and 163 all have order 2 times a very large prime.

For the nontwist curve:

$$y^2 + xy = x^3 + 1 \quad (4.30)$$

fields 103, 107, and 131 all have order 4 times a very large prime [29].

4.4.2 Embedding Data on a Curve

The data (that comes from sensor) should be put onto a curve, and, if a random values are picked, the x value which is picked surely satisfies the equation of the curve.

Rewrite the equation 3.9 as follows:

$$y^2 + xy + f(x) = 0 \quad (4.31)$$

This is a simple quadratic equation.

There are many elements that do not satisfy equation 4.31 in a Galois Field of F_{2^m} . This creates a problem. If the data fill all the bits of the field, and these data can't be on the curve. In this case, the crypto system can't be used. So, the bits that should be in the field must be more than the bits that are in the secret for security reasons. This gives the chance to change some bits in the chosen value of x . It turns out that it does not matter which bits should be changed or how they should be changed. The “density” of points for any curve over a finite field is almost uniformly distributed. This means that any subsection of the bits can be seen as an integer and increment it using simple arithmetic. This method was first proposed by Koblitz [30] and works very well. The raw data are placed along with some random garbage, and then find out if that combination will be on the curve. If not, the garbage is incremented and a test is again performed. Once the values of y “yes”, there are two values of y if there is a solution to equation 4.31 are found, the point p is “on the curve” and it can be hidden.

4.4.3 Solving Quadratic Equations in Binary Fields

The equation 4.31 has two solutions [25], to solve it make a change of variable to eliminate x .

$$z^2 + z + c = 0 \tag{4.32}$$

where

$$c = f(x) \cdot x^{-2} \tag{4.33}$$

$$y = xz \tag{4.34}$$

The equation 4.32 only has a solution when the Trace of c is 0. The Trace function maps F_{2^m} to F_2 ; it gives us 1 bit as result.

Once the $\text{Trace}(c) = 0$ in equation 4.32, the $z+1$ is a solution to the equation.

Let z be one solution and z' be another solution:

$$z(z+1) = c \tag{4.35}$$

$$z'(z'+1) = c$$

Replace z' in the second equation with $z+1$ get the first equation. Once one solution is found the other is trivial. Then, putting z and z' in equation 4.34, get the two solutions for the quadratic equation and the data are embedded on the curve.

4.4.4 The Trace Function

In a field F_{2^m} , the Trace is defined as the sum of the $m-1$ squarings of an element, c , with itself [24].

$$\text{Tr}(c) = c + c^{2^1} + c^{2^2} + \dots + c^{2^{m-1}} \tag{4.36}$$

In a normal basis, equation 4.36 is called a “parity function” [25]. It tells whether the number of bits set is even or odd.

4.4.5 Quadratic Normal Basis Routine

Figure 4.3 shows the Subroutine Schematic 3 for solving quadratic equation and all the routines needed to embed data on curves [25].

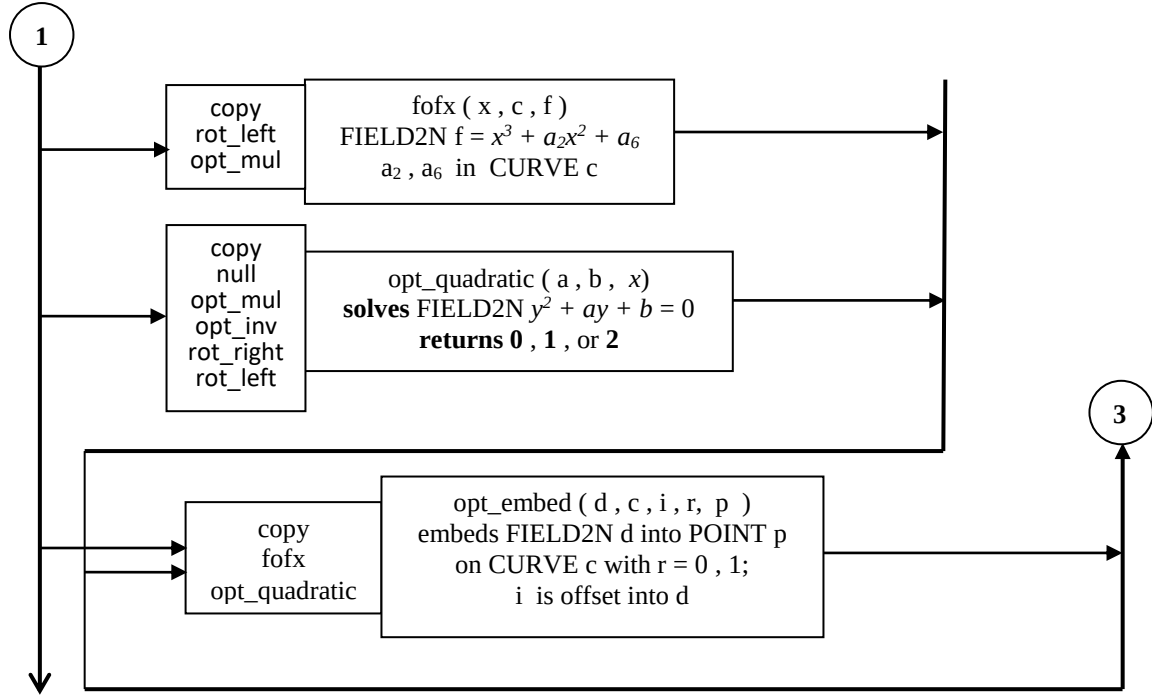


Figure 4.3 : Subroutine Schematic 3

The function **fofx**() used to compute the right-hand side of equation 3.9. Solving a quadratic equation is easy in a normal basis. The first step in **opt_quadratic** routine is solving the equation $y^2 + ay + b$, check if the coefficient of y is 0 then the solution is the square root of the constant b , this is just a right shift for a normal basis. The next step is to compute the constant c in equation 4.33. and compute two solutions for equation 3.32. The **opt_embed**() function embeds data onto a curve.

4.5 Simple Protocols

All routines that illustrate in Subroutine Schematic 1, the math of ONB finite fields, Subroutine Schematic 2, elliptic curves, and Subroutine Schematic 3, getting data onto a curve, used to implement the ECC cryptography algorithm. The Subroutine Schematic 4 that shown in Figure 4.4 [25] illustrates the implementing of **Diffie-Hellman** and **El-Gamal**.

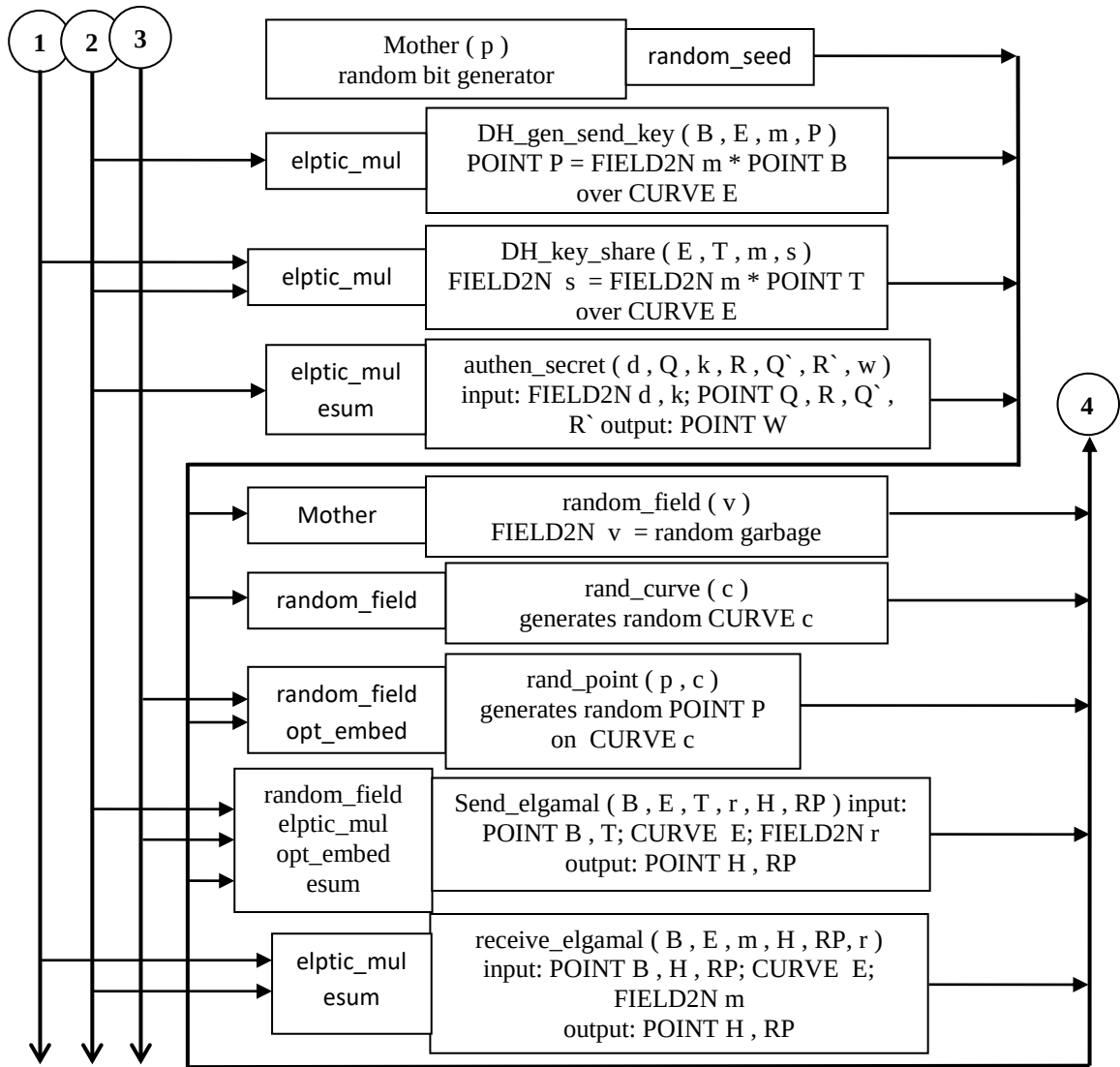


Figure 4.4 Subroutine Schematic 4.

4.5.1 Random Bit Generator

The random bit generator, **mother ()** routine, is used to generate a random bit pattern which fits in a field size, m , variable which used to create random curves. The routine was developed by Dr. Marsaglia [25].

4.5.2 Choosing Random Curves

The choice of curve can be random or it can be a very specific one. For moderate security random curves are perfectly adequate. For high security curves of known order are necessary. But choosing good curves is hard. For cryptography the order of the curve should be contain the largest prime number. For better security the known curves should be used.

rand_curve() routine generate a random curve with $a_2=0$ and a_6 as a random bit pattern.

4.5.3 Elliptic Curve Diffie-Hellman ECDH

The first published public key algorithm known by Diffie and Hellman is Diffie-Hellman key exchange [21]. The purpose of the algorithm is to enable two users to exchange a key securely that can then be used for subsequent encryption of messages. The algorithm itself is limited to the exchange of the keys. The Diffie-Hellman algorithm depends for its effectiveness on the difficulty of computing discrete logarithms [31].

Consider two parties (Alice and Bob) want to communicate with each other [21]. Both chose their own private key to be a random bit pattern in a mutually agreed upon field size. Let k_A be Alice's key and k_B belong to Bob. They first agree to use a specific curve, base point, B, and field size.

Bob computes:

$$P_B = k_B B \quad (4.37)$$

Over the chosen elliptic curve and sends it to Alice. Alice computes:

$$P_A = k_A B \quad (4.38)$$

And sends it to Bob. Then they both compute the shared secret:

$$P_s = k_A (k_B B) = k_B (k_A B) \quad (4.39)$$

4.5.4 El-Gamal

Taher ElGamal described a popular asymmetric encryption algorithm in [32] back in 1985. This algorithm is not mentioned in the IEEE P1363 draft. It is a very useful protocol for randomly generated curves and points, because it does not require knowledge of the order of the curve or the order of the base point. This algorithm allows asymmetric encryption and decryption of data using public and private keys.

Alice wants to encrypt and send secret data to Bob in a way only Bob can decrypt it. Similar to Diffie-Hellman both have agreed to a curve and a base point B [25].

Both chose their private key k_A and k_B .

Alice computes her public key:

$$P_A = k_A B \quad (4.40)$$

And sends it to Bob. Bob computes his public key:

$$P_B = k_B B \quad (4.41)$$

and sends it to Alice. Now Alice and Bob can send messages using these public points in such a way that no one can discover the data without solving the discrete elliptic curve logarithm problem.

Alice wants to send a message to Bob. she first embeds the message information onto the curve, E , using the previous embedding method.

Let's message point P_m .

Alice chooses a random bit pattern, r , and computes two points:

$$P_r = rB \quad (4.42)$$

and

$$P_h = P_m + rP_B \quad (4.43)$$

Alice then sends both points P_h and P_r to Bob. To decrypt the message point, Bob computes:

$$P_s = k_B P_r \quad (4.44)$$

and subtracts this from P_h to get:

$$P_m = P_h - P_s. \quad (4.45)$$

CHAPTER 5

RESULTS OF TESTING OF ELLIPTIC CURVE

5.1 Introduction

There are several implementations of ECC over $GF(2^m)$ on sensor motes. Malan et al. implemented ECCM which was the first implementation of ECC over $GF(2^m)$ on a 8-bit sensor mote [5]. They used ECC to provide a key distribution mechanism for UC Berkeley's TinySec module. ECCM takes 34 secs for generating a public key and requires 34,342-byte of ROM as code size. In [34], Yan and Shi indicated that the software implementations of ECC over $GF(2^m)$ were still slow on small computing devices such as sensor nodes. They implemented 163-bit ECC using fast modular reduction on a 8-bit ATmega128L processor. Their implementation requires 11,592-byte of code size and takes 13.9 secs for a scalar multiplication. Eberle et al. pointed out that field arithmetic over $GF(2^m)$, especially field multiplication is prohibitively slow since general-purpose microprocessors do not support arithmetic in that field [35]. Also, Blaß and Zitterbart implemented ECDH, ECDSA and El-Gamal over $GF(2^{113})$ [11]. Their implementation took 6.88 and 24.17 secs for signature generation and verification, respectively. Their code occupies 75,088-byte of ROM.

Table 5.1 shows comparison between our code and other existing software implementation.

Table 5.1 : comparison between software implementation of ECC

ECC implementing by:	Key size	Basis	Code size
Blaß and Zitterbart	113	Type II ONB	75,088
Yan and Shi	163	Polynomial with fast reduction	11,592
Our code	162, 172, 173 and 174	Type I and Type II ONB	24,995

Compared with Blaß and Zitterbart code the key length is higher and less memory size, which means 33% of main memory, were saved. The execution time is not as critical. consumes less memory and offers faster operation (less execution time).

5.2 Software Used

All the code in this thesis is written in C. The source code is shown in Appendix C.

5.3 Simulation Scenario

The testing of ECC implementation is done with different values of key size, m , (NUMBITS in code). From Appendix B we choose:

- Key size = 162,172 for Type I
- Key size = 173,174 for Type II

And tested with different curves :

- Random curve ($a_2=0$, and a_6 is random value).
- Koblitz curve ($a_2=1$, $a_6=1$).

5.3.1 Diffie-Hellman key exchange

Compute a Diffie-Hellman key exchange:

- Computes sender's public key.

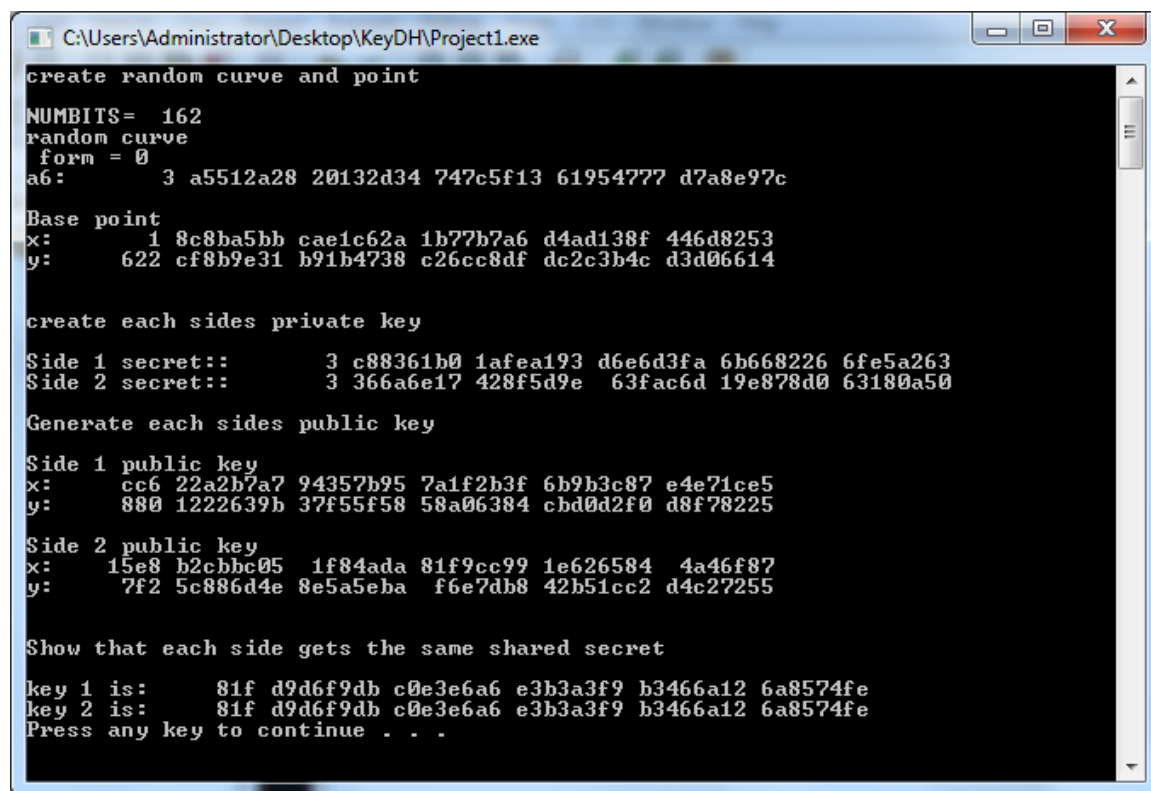
Generate the public key using Base point which sits on public curve, E, and sender's private key.

- Computes shared secret key that is same for sender and receiver.

Enter with public point Base point which sits on public curve E along with senders public key their_public and receivers private key k.

Returns shared_secret as x component of kP.

Figure 5.1 shows the result of computing Diffie-Hellman key exchange when key size 162 bit and Type I ONB.



```
C:\Users\Administrator\Desktop\KeyDH\Project1.exe
create random curve and point
NUMBITS= 162
random curve
form = 0
a6:      3 a5512a28 20132d34 747c5f13 61954777 d7a8e97c
Base point
x:      1 8c8ba5bb cae1c62a 1b77b7a6 d4ad138f 446d8253
y:      622 cf8b9e31 b91b4738 c26cc8df dc2c3b4c d3d06614
create each sides private key
Side 1 secret::      3 c88361b0 1afea193 d6e6d3fa 6b668226 6fe5a263
Side 2 secret::      3 366a6e17 428f5d9e 63fac6d 19e878d0 63180a50
Generate each sides public key
Side 1 public key
x:      cc6 22a2b7a7 94357b95 7a1f2b3f 6b9b3c87 e4e71ce5
y:      880 1222639b 37f55f58 58a06384 cbd0d2f0 d8f78225
Side 2 public key
x:      15e8 b2cbbcb05 1f84ada 81f9cc99 1e626584 4a46f87
y:      7f2 5c886d4e 8e5a5eba f6e7db8 42b51cc2 d4c27255
Show that each side gets the same shared secret
key 1 is:      81f d9d6f9db c0e3e6a6 e3b3a3f9 b3466a12 6a8574fe
key 2 is:      81f d9d6f9db c0e3e6a6 e3b3a3f9 b3466a12 6a8574fe
Press any key to continue . . .
```

Figure 5.1 : Type I ONB, $m=162$ and random (curve, base point)

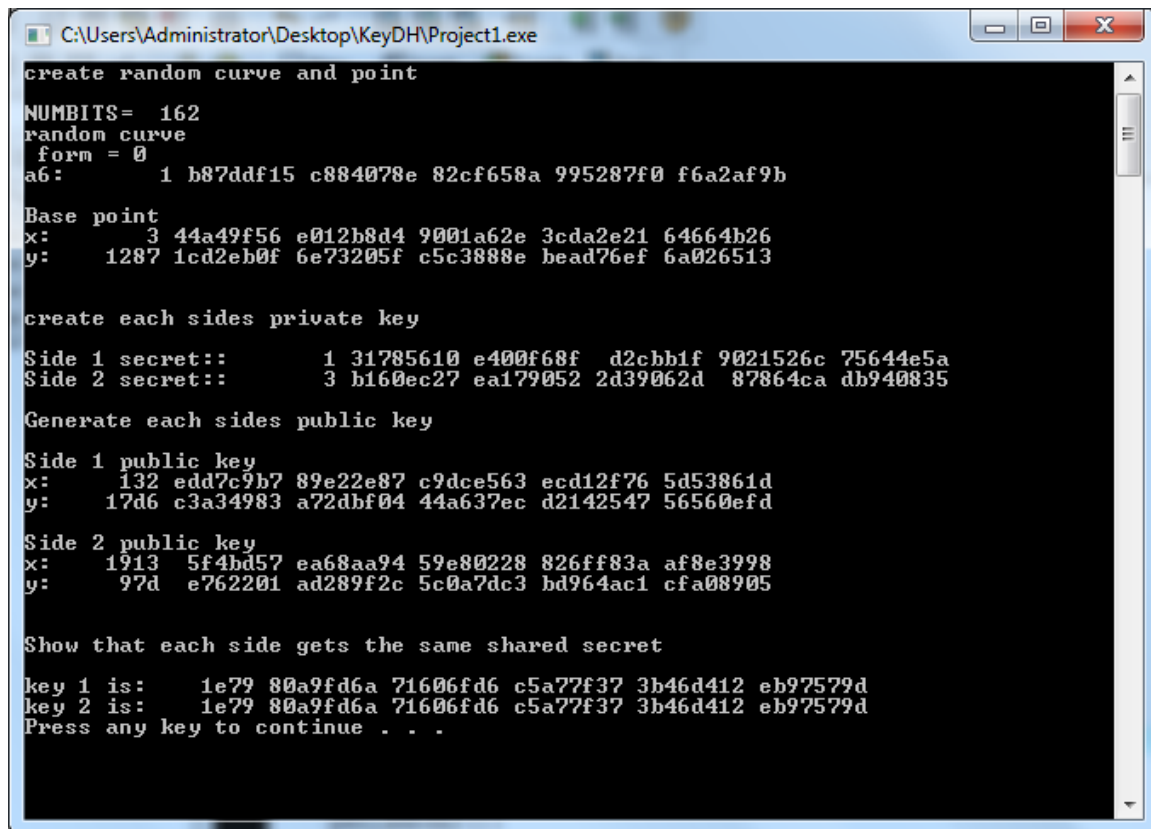
First generate random curve ($a_2 = 0$, a_6 random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.2 shows the result of computing Diffie-Hellman key exchange when key size 162 bit and Type I ONB. Different random curve and Base point are used.



```
create random curve and point
NUMBITS= 162
random curve
form = 0
a6: 1 b87ddf15 c884078e 82cf658a 995287f0 f6a2af9b

Base point
x: 3 44a49f56 e012b8d4 9001a62e 3cda2e21 64664b26
y: 1287 1cd2eb0f 6e73205f c5c3888e bead76ef 6a026513

create each sides private key
Side 1 secret:: 1 31785610 e400f68f d2cbb1f 9021526c 75644e5a
Side 2 secret:: 3 b160ec27 ea179052 2d39062d 87864ca db940835

Generate each sides public key
Side 1 public key
x: 132 edd7c9b7 89e22e87 c9dce563 ecd12f76 5d53861d
y: 17d6 c3a34983 a72dbf04 44a637ec d2142547 56560efd
Side 2 public key
x: 1913 5f4bd57 ea68aa94 59e80228 826ff83a af8e3998
y: 97d e762201 ad289f2c 5c0a7dc3 bd964ac1 cfa08905

Show that each side gets the same shared secret
key 1 is: 1e79 80a9fd6a 71606fd6 c5a77f37 3b46d412 eb97579d
key 2 is: 1e79 80a9fd6a 71606fd6 c5a77f37 3b46d412 eb97579d
Press any key to continue . . .
```

Figure 5.2 : Type I ONB, $m=162$ and another random (curve, base point)

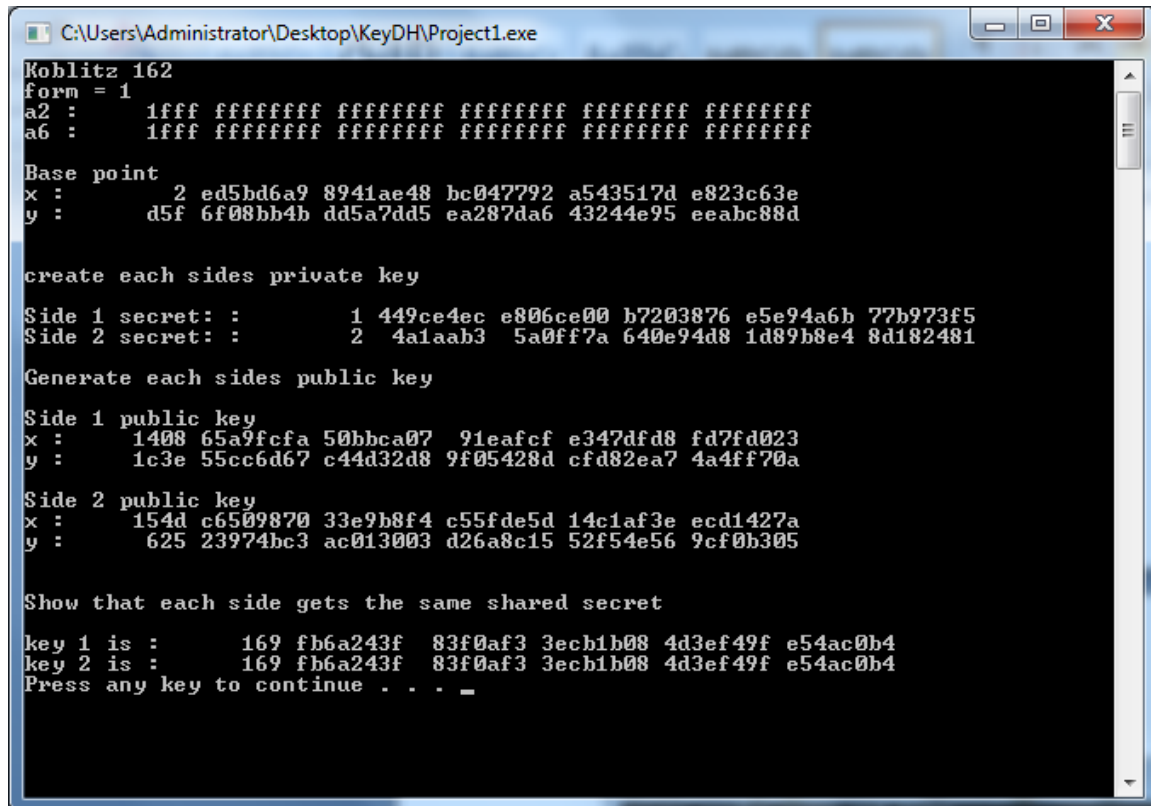
First generate random curve ($a_2 = 0$, a_6 random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.3 shows the result of computing Diffie-Hellman key exchange when key size 162 bit and Type I ONB. Using koblitz curve and random Base point.



```
Koblitz 162
form = 1
a2 :      1fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6 :      1fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x :      2 ed5bd6a9 8941ae48 bc047792 a543517d e823c63e
y :      d5f 6f08bb4b dd5a7dd5 ea287da6 43244e95 eeabc88d

create each sides private key
Side 1 secret: :      1 449ce4ec e806ce00 b7203876 e5e94a6b 77b973f5
Side 2 secret: :      2 4a1aab3 5a0ff7a 640e94d8 1d89b8e4 8d182481

Generate each sides public key
Side 1 public key
x :      1408 65a9fcfa 50bbca07 91eafcf e347dfd8 fd7fd023
y :      1c3e 55cc6d67 c44d32d8 9f05428d cfd82ea7 4a4ff70a
Side 2 public key
x :      154d c6509870 33e9b8f4 c55fde5d 14c1af3e ecd1427a
y :      625 23974bc3 ac013003 d26a8c15 52f54e56 9cf0b305

Show that each side gets the same shared secret
key 1 is :      169 fb6a243f 83f0af3 3ecb1b08 4d3ef49f e54ac0b4
key 2 is :      169 fb6a243f 83f0af3 3ecb1b08 4d3ef49f e54ac0b4
Press any key to continue . . . _
```

Figure 5.3 : Type I ONB, $m=162$ and koblitz curve , base point

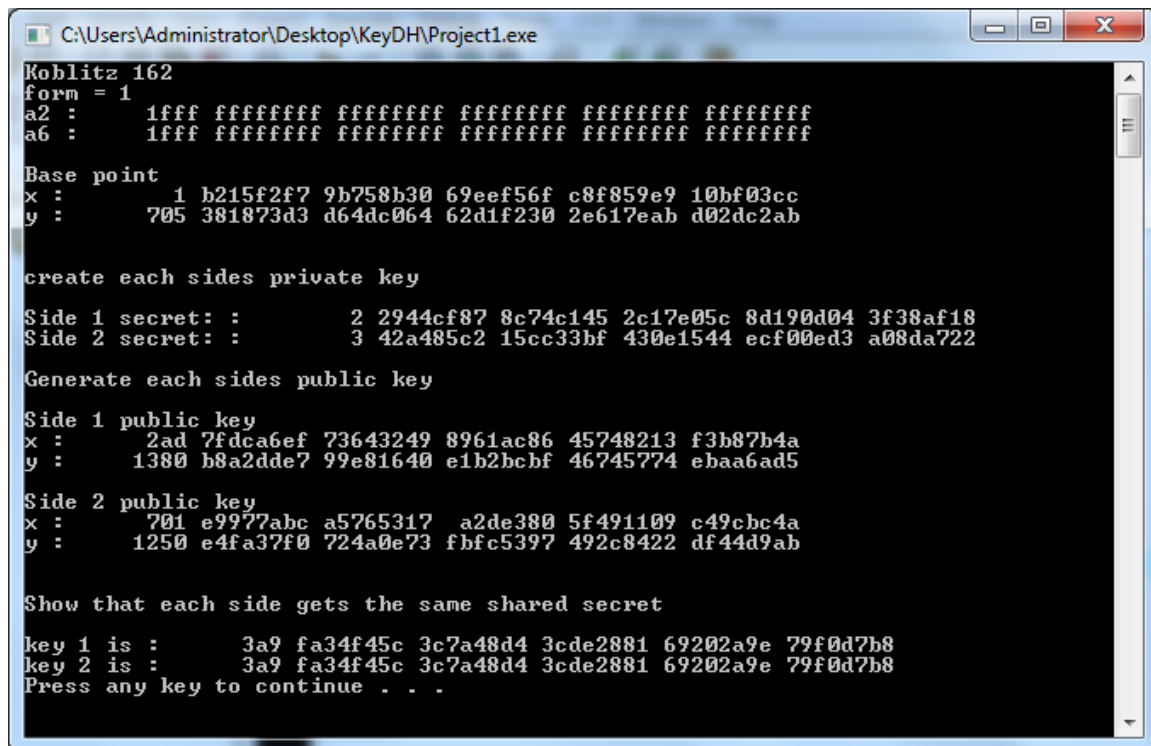
Create koblitz curve ($a_2=1, a_6=1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.4 shows the result of computing Diffie-Hellman key exchange when key size 162 bit and Type I ONB. Using koblitz curve and different Base point.



```
Koblitz 162
form = 1
a2 :      1fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6 :      1fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x :      1 b215f2f7 9b758b30 69eef56f c8f859e9 10bf03cc
y :      705 381873d3 d64dc064 62d1f230 2e617eab d02dc2ab

create each sides private key
Side 1 secret: :      2 2944cf87 8c74c145 2c17e05c 8d190d04 3f38af18
Side 2 secret: :      3 42a485c2 15cc33bf 430e1544 ecf00ed3 a08da722

Generate each sides public key
Side 1 public key
x :      2ad 7fdca6ef 73643249 8961ac86 45748213 f3b87b4a
y :      1380 b8a2dde7 99e81640 e1b2bcbf 46745774 ebaa6ad5
Side 2 public key
x :      701 e9977abc a5765317 a2de380 5f491109 c49cbc4a
y :      1250 e4fa37f0 724a0e73 fbfc5397 492c8422 df44d9ab

Show that each side gets the same shared secret
key 1 is :      3a9 fa34f45c 3c7a48d4 3cde2881 69202a9e 79f0d7b8
key 2 is :      3a9 fa34f45c 3c7a48d4 3cde2881 69202a9e 79f0d7b8
Press any key to continue . . .
```

Figure 5.4 : Type I ONB, $m=162$ and koblitz curve , another base point

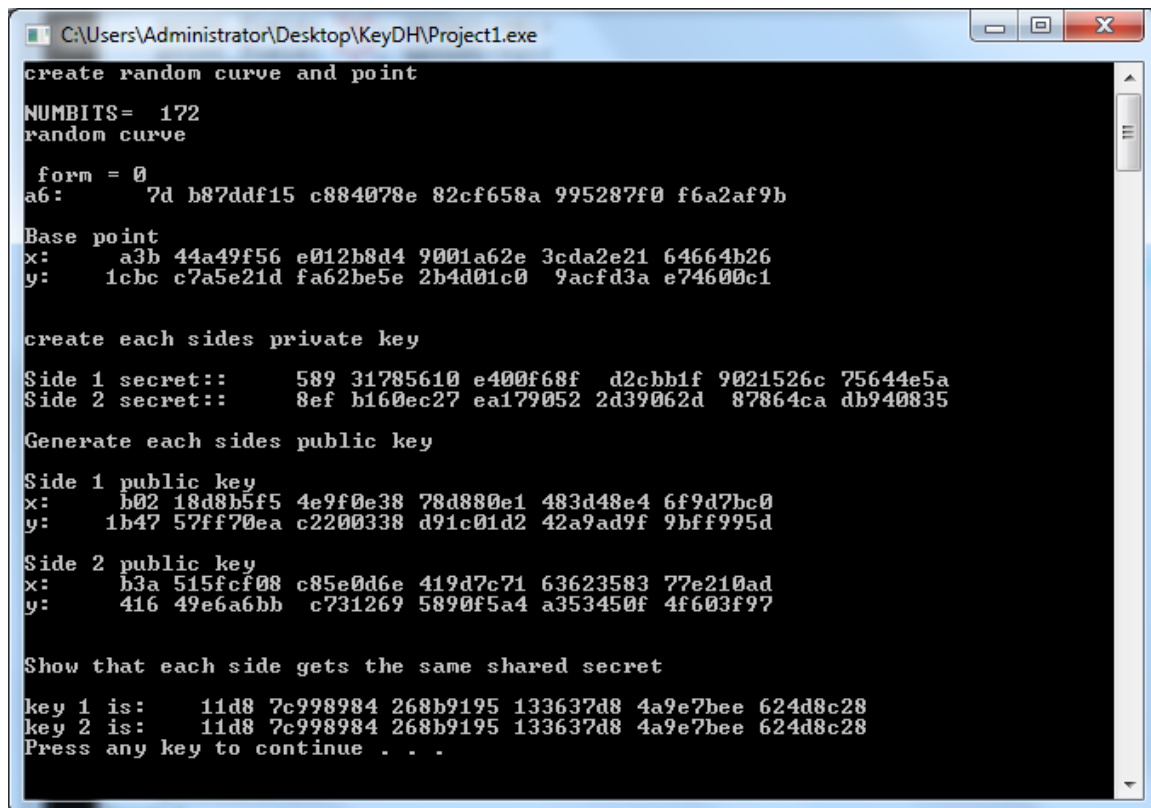
Create koblitz curve ($a_2=1, a_6=1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.5 shows the result of computing Diffie-Hellman key exchange when key size 172 bit and Type I ONB. The curve and Base point that used are random.



```
C:\Users\Administrator\Desktop\KeyDH\Project1.exe
create random curve and point
NUMBITS= 172
random curve
form = 0
a6: 7d b87ddf15 c884078e 82cf658a 995287f0 f6a2af9b
Base point
x: a3b 44a49f56 e012b8d4 9001a62e 3cda2e21 64664b26
y: 1cbc c7a5e21d fa62be5e 2b4d01c0 9acfd3a e74600c1

create each sides private key
Side 1 secret:: 589 31785610 e400f68f d2cbb1f 9021526c 75644e5a
Side 2 secret:: 8ef b160ec27 ea179052 2d39062d 87864ca db940035

Generate each sides public key
Side 1 public key
x: b02 18d8b5f5 4e9f0e38 78d880e1 483d48e4 6f9d7bc0
y: 1b47 57ff70ea c2200338 d91c01d2 42a9ad9f 9bff995d
Side 2 public key
x: b3a 515fcf08 c85e0d6e 419d7c71 63623583 77e210ad
y: 416 49e6a6bb c731269 5890f5a4 a353450f 4f603f97

Show that each side gets the same shared secret
key 1 is: 11d8 7c998984 268b9195 133637d8 4a9e7bee 624d8c28
key 2 is: 11d8 7c998984 268b9195 133637d8 4a9e7bee 624d8c28
Press any key to continue . . .
```

Figure 5.5 : Type I ONB, $m=172$ and random curve, Base point

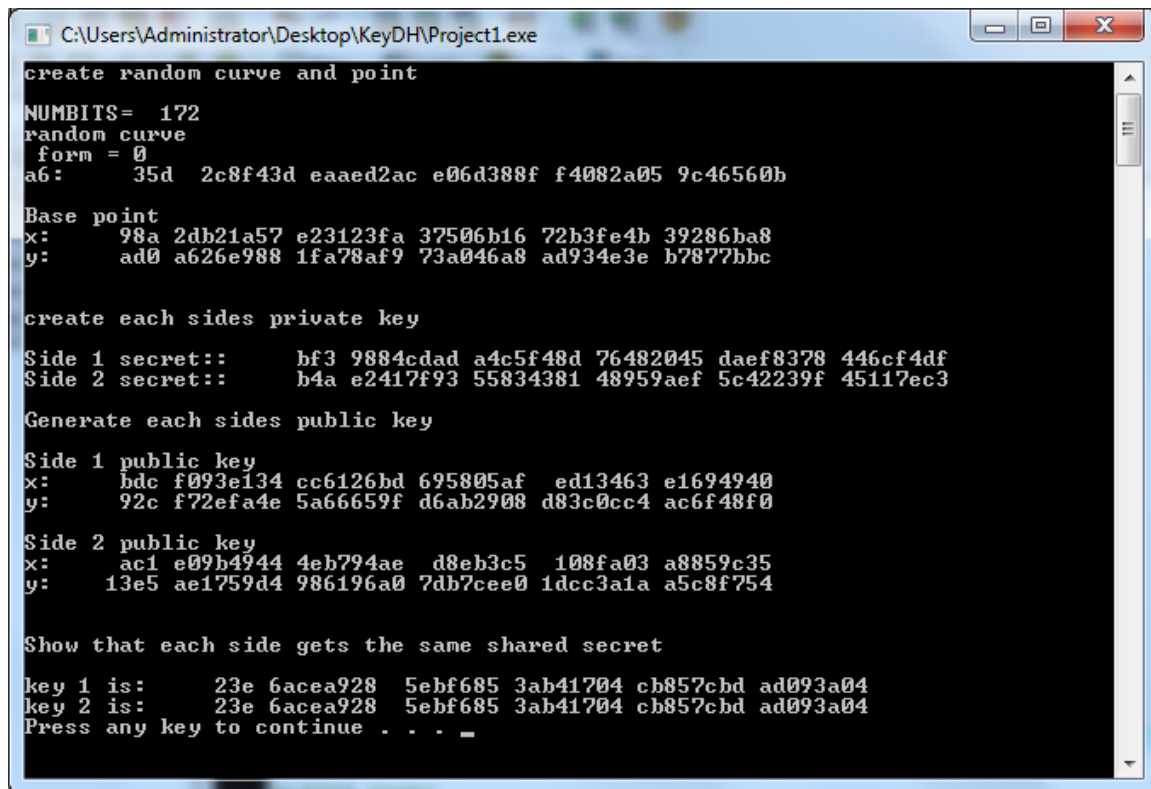
Create random curve ($a_2 = 0$, $a_6 =$ random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.6 shows the result of computing Diffie-Hellman key exchange when key size 172 bit and Type I ONB. Different random curve and Base point are used.



```
create random curve and point
NUMBITS= 172
random curve
form = 0
a6: 35d 2c8f43d eaaed2ac e06d388f f4082a05 9c46560b

Base point
x: 98a 2db21a57 e23123fa 37506b16 72b3fe4b 39286ba8
y: ad0 a626e988 1fa78af9 73a046a8 ad934e3e b7877bbc

create each sides private key
Side 1 secret:: bf3 9884cdad a4c5f48d 76482045 daef8378 446cf4df
Side 2 secret:: b4a e2417f93 55834381 48959aef 5c42239f 45117ec3

Generate each sides public key
Side 1 public key
x: bdc f093e134 cc6126bd 695805af ed13463 e1694940
y: 92c f72efa4e 5a66659f d6ab2908 d83c0cc4 ac6f48f0
Side 2 public key
x: ac1 e09b4944 4eb794ae d8eb3c5 108fa03 a8859c35
y: 13e5 ae1759d4 986196a0 7db7cee0 1dcc3a1a a5c8f754

Show that each side gets the same shared secret
key 1 is: 23e 6acea928 5ebf685 3ab41704 cb857cbd ad093a04
key 2 is: 23e 6acea928 5ebf685 3ab41704 cb857cbd ad093a04
Press any key to continue . . .
```

Figure 5.6 : Type I ONB, $m=172$ and another (random curve, Base point)

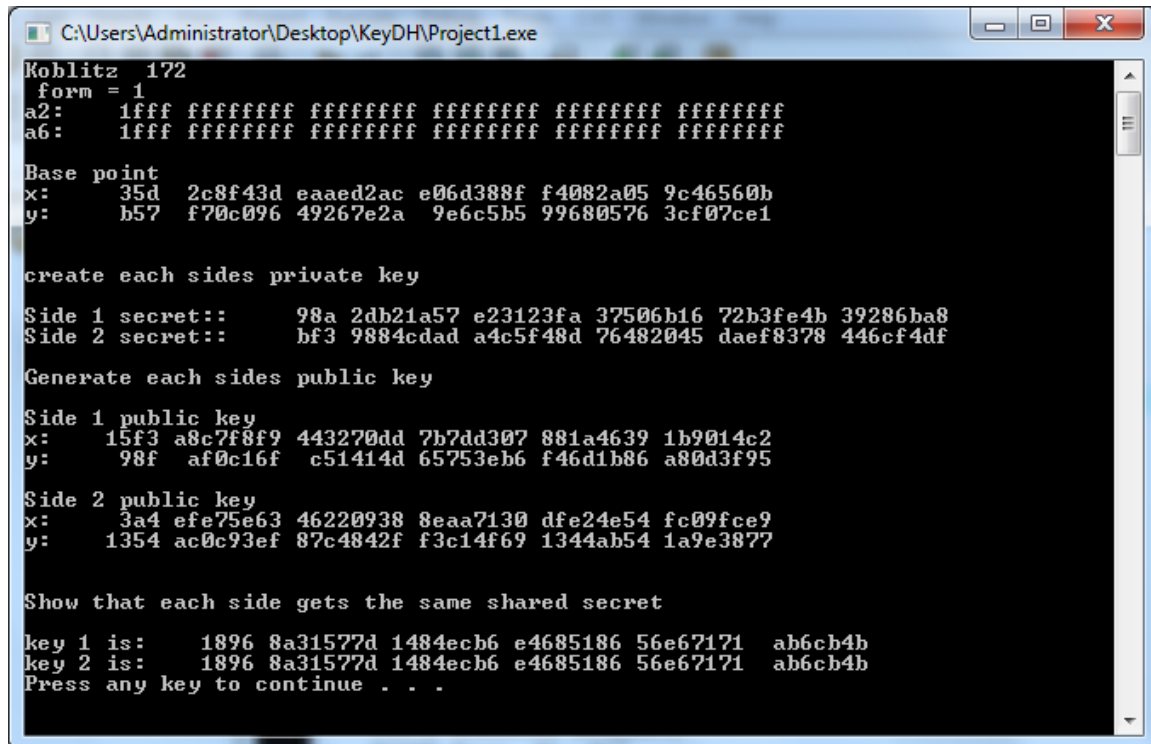
First generate random curve ($a_2 = 0$, a_6 random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.7 shows the result of computing Diffie-Hellman key exchange when key size 172 bit and Type I ONB. The koblitz curve and random Base point are used.



```
Koblitz 172
form = 1
a2: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x: 35d 2c8f43d eaaed2ac e06d388f f4082a05 9c46560b
y: b57 f70c096 49267e2a 9e6c5b5 99680576 3cf07ce1

create each sides private key
Side 1 secret:: 98a 2db21a57 e23123fa 37506b16 72b3fe4b 39286ba8
Side 2 secret:: bf3 9884cdad a4c5f48d 76482045 daef8378 446cf4df

Generate each sides public key
Side 1 public key
x: 15f3 a8c7f8f9 443270dd 7b7dd307 881a4639 1b9014c2
y: 98f af0c16f c51414d 65753eb6 f46d1b86 a80d3f95
Side 2 public key
x: 3a4 efe75e63 46220938 8eaa7130 dfe24e54 fc09fce9
y: 1354 ac0c93ef 87c4842f f3c14f69 1344ab54 1a9e3877

Show that each side gets the same shared secret
key 1 is: 1896 8a31577d 1484ecb6 e4685186 56e67171 ab6cb4b
key 2 is: 1896 8a31577d 1484ecb6 e4685186 56e67171 ab6cb4b
Press any key to continue . . .
```

Figure 5.7 : Type I ONB, $m=172$ and koblitz curve , base point

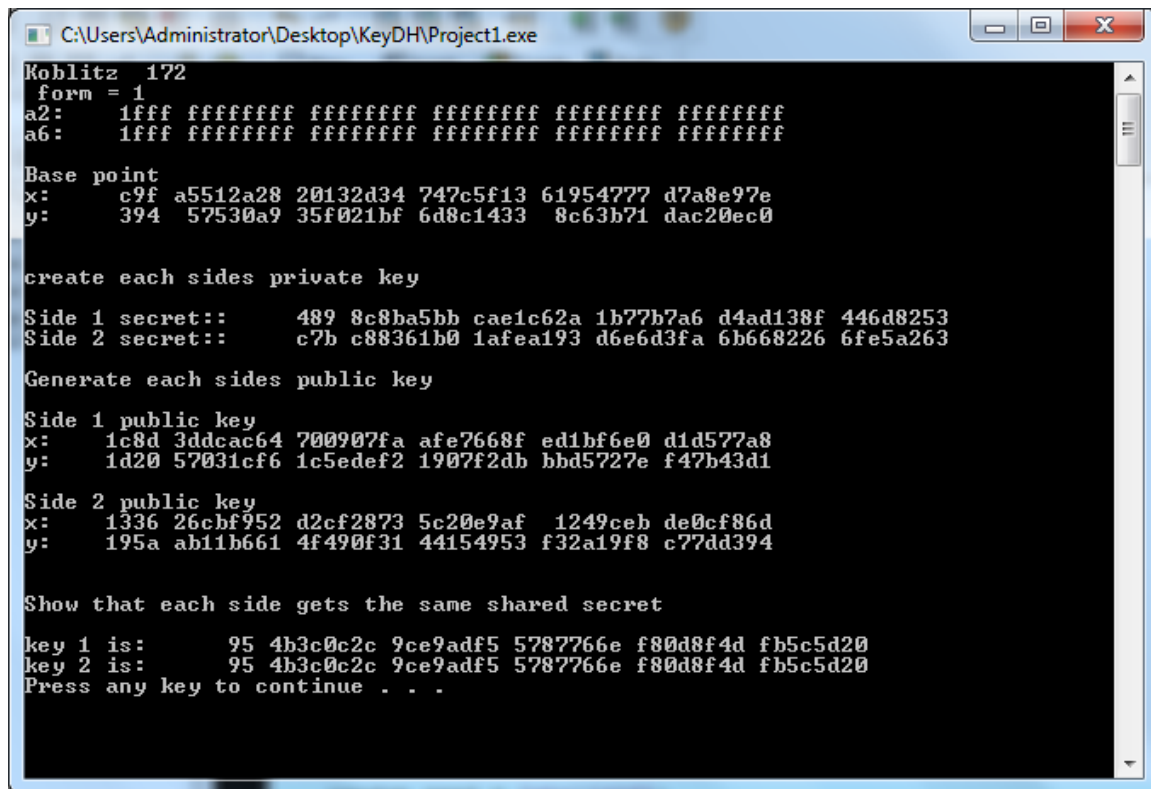
Create koblitz curve ($a_2=1, a_6=1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.8 shows the result of computing Diffie-Hellman key exchange when key size 172 bit and Type I ONB. The koblitz curve is used with different random Base point.



```
Koblitz 172
form = 1
a2: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x: c9f a5512a28 20132d34 747c5f13 61954777 d7a8e97e
y: 394 57530a9 35f021bf 6d8c1433 8c63b71 dac20ec0

create each sides private key
Side 1 secret:: 489 8c8ba5bb cae1c62a 1b77b7a6 d4ad138f 446d8253
Side 2 secret:: c7b c88361b0 1afea193 d6e6d3fa 6b668226 6fe5a263

Generate each sides public key
Side 1 public key
x: 1c8d 3ddcac64 700907fa afe7668f ed1bf6e0 d1d577a8
y: 1d20 57031cf6 1c5edef2 1907f2db bbd5727e f47b43d1
Side 2 public key
x: 1336 26cbf952 d2cf2873 5c20e9af 1249ceb de0cf86d
y: 195a ab11b661 4f490f31 44154953 f32a19f8 c77dd394

Show that each side gets the same shared secret
key 1 is: 95 4b3c0c2c 9ce9adf5 5787766e f80d8f4d fb5c5d20
key 2 is: 95 4b3c0c2c 9ce9adf5 5787766e f80d8f4d fb5c5d20
Press any key to continue . . .
```

Figure 5.8 : Type I ONB, $m=172$ and koblitz curve , another base point

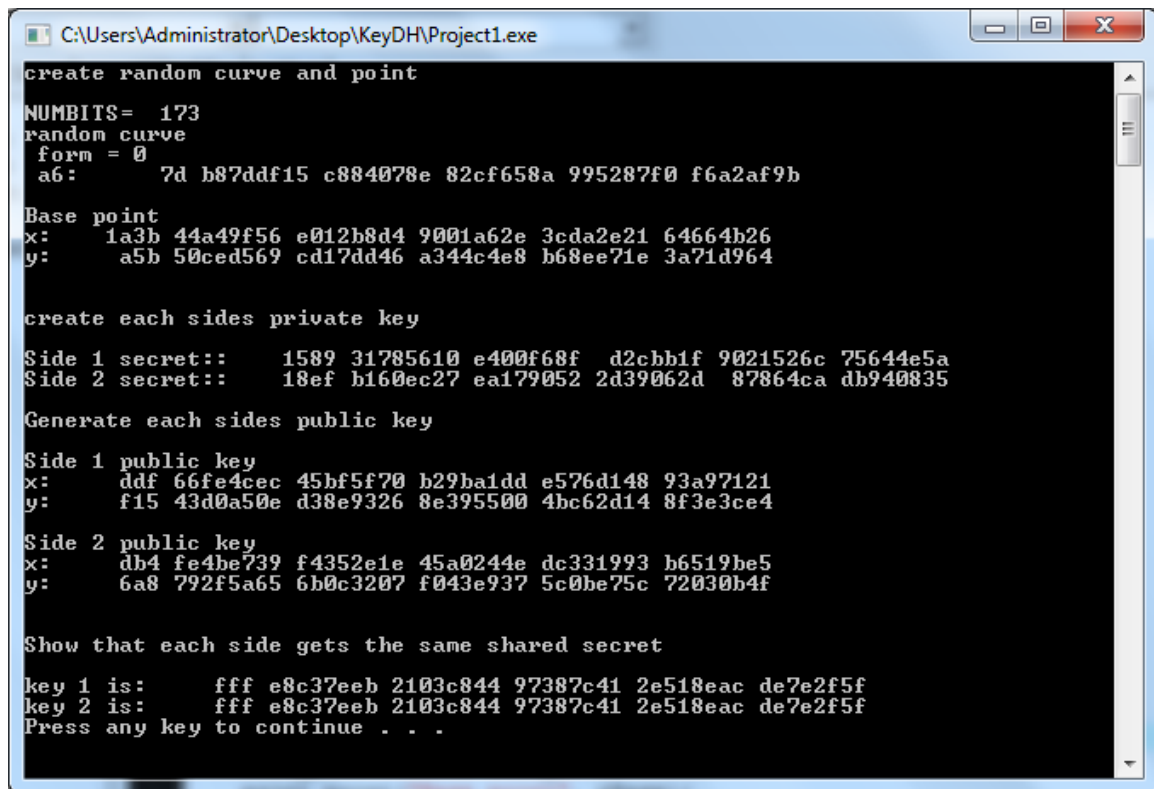
Create koblitz curve ($a_2=1, a_6=1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.9 shows the result of computing Diffie-Hellman key exchange when key size 173 bit and Type II ONB. The curve and Base point that used are random.



```
create random curve and point
NUMBITS= 173
random curve
form = 0
a6: 7d b87ddf15 c884078e 82cf658a 995287f0 f6a2af9b

Base point
x: 1a3b 44a49f56 e012b8d4 9001a62e 3cda2e21 64664b26
y: a5b 50ced569 cd17dd46 a344c4e8 b68ee71e 3a71d964

create each sides private key
Side 1 secret:: 1589 31785610 e400f68f d2cbb1f 9021526c 75644e5a
Side 2 secret:: 18ef b160ec27 ea179052 2d39062d 87864ca db940835

Generate each sides public key
Side 1 public key
x: ddf 66fe4cec 45bf5f70 b29ba1dd e576d148 93a97121
y: f15 43d0a50e d38e9326 8e395500 4bc62d14 8f3e3ce4
Side 2 public key
x: db4 fe4be739 f4352e1e 45a0244e dc331993 b6519be5
y: 6a8 792f5a65 6b0c3207 f043e937 5c0be75c 72030b4f

Show that each side gets the same shared secret
key 1 is: fff e8c37eeb 2103c844 97387c41 2e518eac de7e2f5f
key 2 is: fff e8c37eeb 2103c844 97387c41 2e518eac de7e2f5f
Press any key to continue . . .
```

Figure 5.9: Type II ONB, $m=173$ and random (curve, base point)

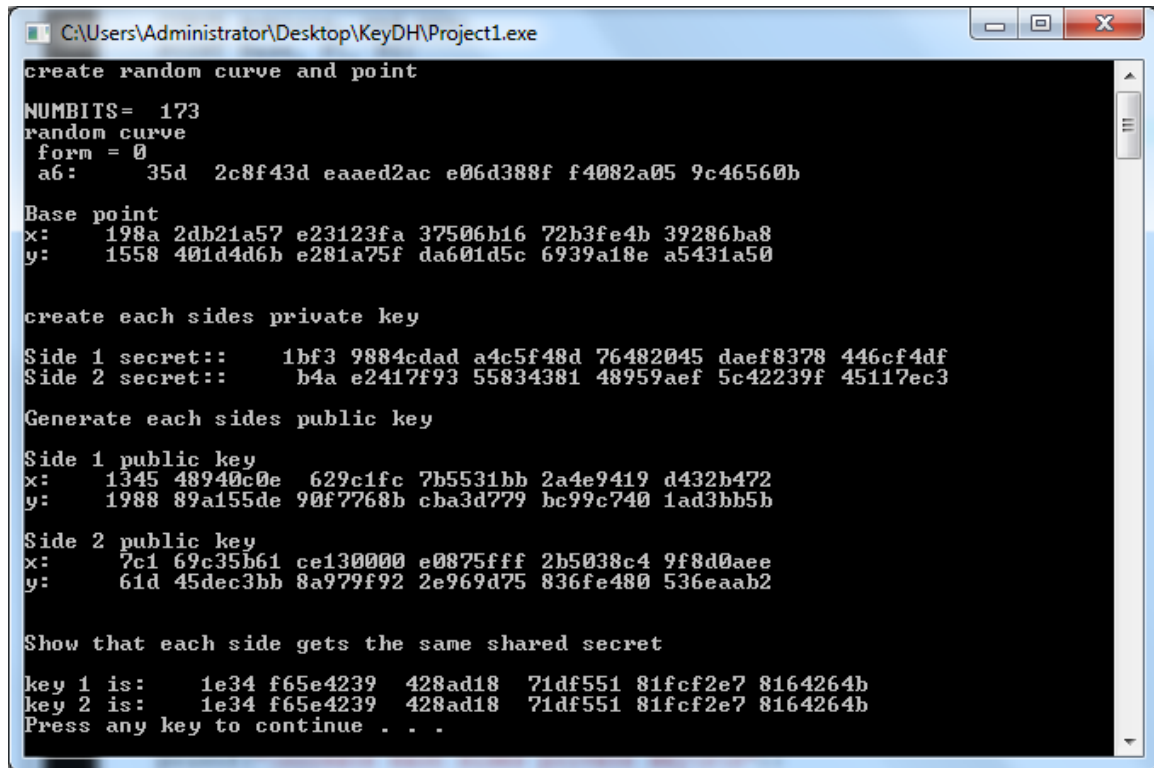
First generate random curve ($a_2 = 0$, a_6 random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.10 shows the result of computing Diffie-Hellman key exchange when key size 173 bit and Type II ONB. Different random curve and Base point are used.



```
C:\Users\Administrator\Desktop\KeyDH\Project1.exe
create random curve and point
NUMBITS= 173
random curve
form = 0
a6: 35d 2c8f43d eaaed2ac e06d388f f4082a05 9c46560b

Base point
x: 198a 2db21a57 e23123fa 37506b16 72b3fe4b 39286ba8
y: 1558 401d4d6b e281a75f da601d5c 6939a18e a5431a50

create each sides private key
Side 1 secret:: 1bf3 9884cdad a4c5f48d 76482045 daef8378 446cf4df
Side 2 secret:: b4a e2417f93 55834381 48959aef 5c42239f 45117ec3

Generate each sides public key
Side 1 public key
x: 1345 48940c0e 629c1fc 7b5531bb 2a4e9419 d432b472
y: 1988 89a155de 90f7768b cba3d779 bc99c740 1ad3bb5b
Side 2 public key
x: 7c1 69c35b61 ce130000 e0875fff 2b5038c4 9f8d0aee
y: 61d 45dec3bb 8a979f92 2e969d75 836fe480 536eaab2

Show that each side gets the same shared secret
key 1 is: 1e34 f65e4239 428ad18 71df551 81fcf2e7 8164264b
key 2 is: 1e34 f65e4239 428ad18 71df551 81fcf2e7 8164264b
Press any key to continue . . .
```

Figure 5.10 : Type II ONB, $m=173$ and another random (curve, base point)

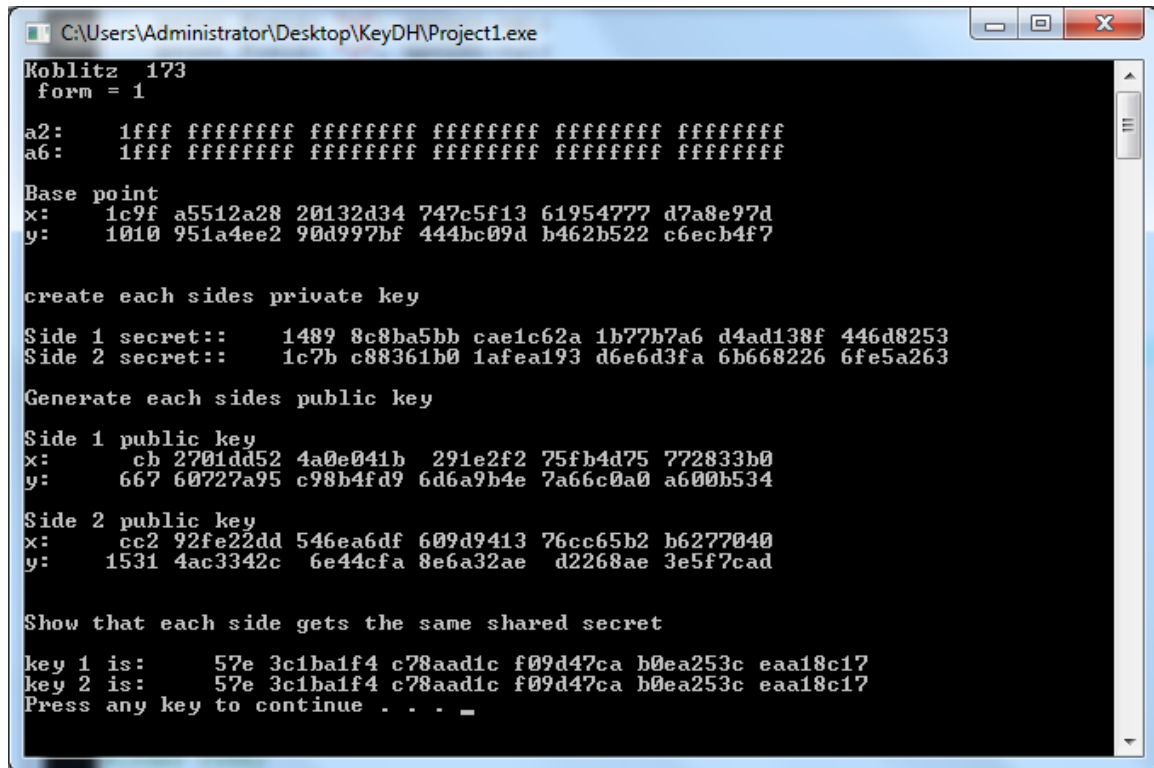
First generate random curve ($a_2 = 0$, a_6 random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.11 shows the result of computing Diffie-Hellman key exchange when key size 173 bit and Type II ONB. The koblitz curve and random Base point are used.



```
C:\Users\Administrator\Desktop\KeyDH\Project1.exe
Koblitz 173
form = 1

a2: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x: 1c9f a5512a28 20132d34 747c5f13 61954777 d7a8e97d
y: 1010 951a4ee2 90d997bf 444bc09d b462b522 c6ecb4f7

create each sides private key
Side 1 secret:: 1489 8c8ba5bb cae1c62a 1b77b7a6 d4ad138f 446d8253
Side 2 secret:: 1c7b c88361b0 1afea193 d6e6d3fa 6b668226 6fe5a263

Generate each sides public key
Side 1 public key
x: cb 2701dd52 4a0e041b 291e2f2 75fb4d75 772833b0
y: 667 60727a95 c98b4fd9 6d6a9b4e 7a66c0a0 a600b534
Side 2 public key
x: cc2 92fe22dd 546ea6df 609d9413 76cc65b2 b6277040
y: 1531 4ac3342c 6e44cfa 8e6a32ae d2268ae 3e5f7cad

Show that each side gets the same shared secret
key 1 is: 57e 3c1ba1f4 c78aad1c f09d47ca b0ea253c eaa18c17
key 2 is: 57e 3c1ba1f4 c78aad1c f09d47ca b0ea253c eaa18c17
Press any key to continue . . . _
```

Figure 5.11 : Type II ONB, $m=173$ and koblitz curve , base point

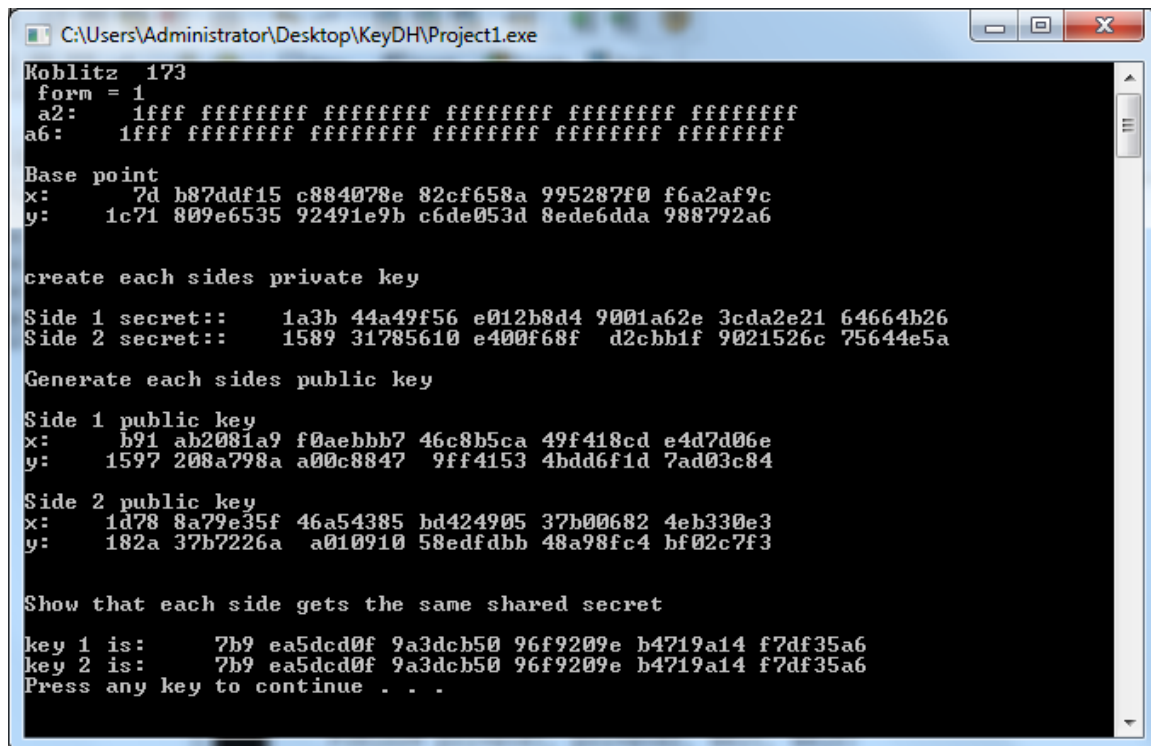
Create koblitz curve ($a_2 = 1$, $a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.12 shows the result of computing Diffie-Hellman key exchange when key size 173 bit and Type II ONB. using koblitz curve and different random Base point.



```
Koblitz 173
form = 1
a2: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x: 7d b87ddf15 c884078e 82cf658a 995287f0 f6a2af9c
y: 1c71 809e6535 92491e9b c6de053d 8ede6dda 988792a6

create each sides private key
Side 1 secret:: 1a3b 44a49f56 e012b8d4 9001a62e 3cda2e21 64664b26
Side 2 secret:: 1589 31785610 e400f68f d2cbb1f 9021526c 75644e5a

Generate each sides public key
Side 1 public key
x: b91 ab2081a9 f0aebbb7 46c8b5ca 49f418cd e4d7d06e
y: 1597 208a798a a00c8847 9ff4153 4bdd6f1d 7ad03c84
Side 2 public key
x: 1d78 8a79e35f 46a54385 bd424905 37b00682 4eb330e3
y: 182a 37b7226a a010910 58edfdbb 48a98fc4 bf02c7f3

Show that each side gets the same shared secret
key 1 is: 7b9 ea5dcd0f 9a3dcb50 96f9209e b4719a14 f7df35a6
key 2 is: 7b9 ea5dcd0f 9a3dcb50 96f9209e b4719a14 f7df35a6
Press any key to continue . . .
```

Figure 5.12 : Type II ONB, $m=173$ and koblitz curve , another base point

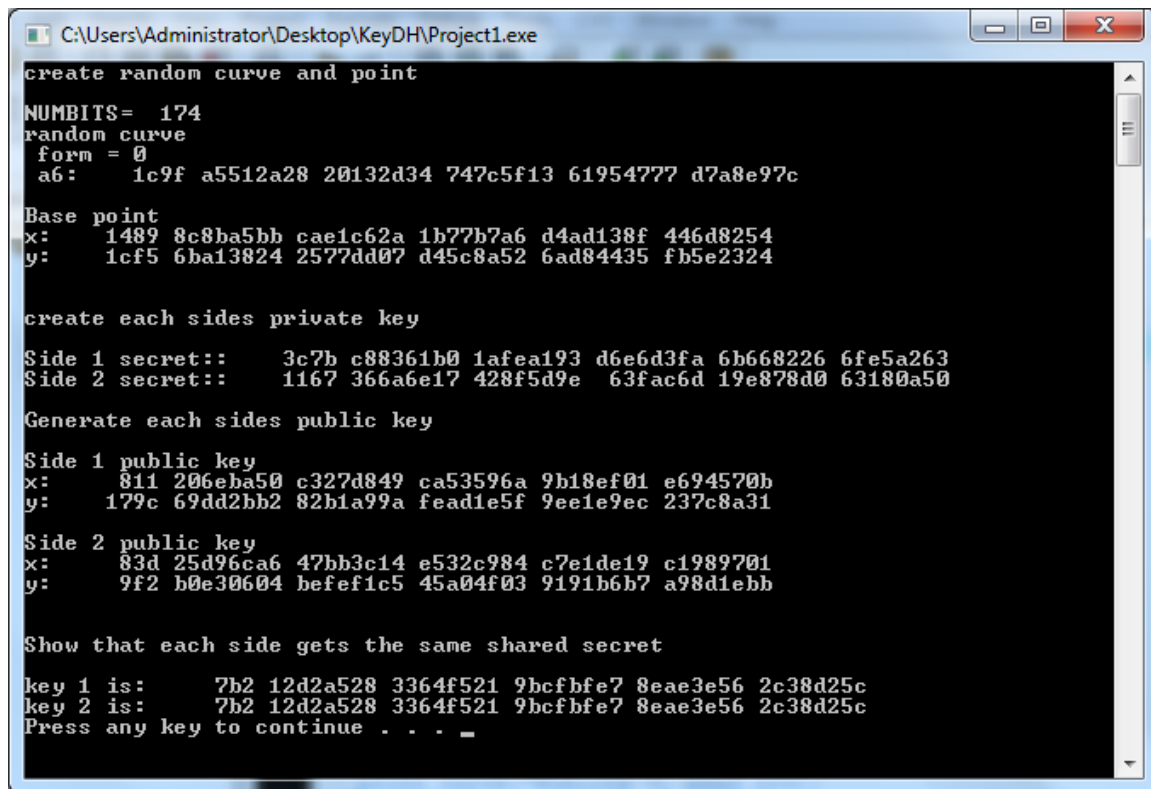
Create koblitz curve ($a_2 = 1, a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.13 shows the result of computing Diffie-Hellman key exchange when key size 174 bit and Type II ONB. The curve and Base point that used are random.



```
create random curve and point
NUMBITS= 174
random curve
form = 0
a6: 1c9f a5512a28 20132d34 747c5f13 61954777 d7a8e97c

Base point
x: 1489 8c8ba5bb cae1c62a 1b77b7a6 d4ad138f 446d8254
y: 1cf5 6ba13824 2577dd07 d45c8a52 6ad84435 fb5e2324

create each sides private key
Side 1 secret:: 3c7b c88361b0 1afea193 d6e6d3fa 6b668226 6fe5a263
Side 2 secret:: 1167 366a6e17 428f5d9e 63fac6d 19e878d0 63180a50

Generate each sides public key
Side 1 public key
x: 811 206eba50 c327d849 ca53596a 9b18ef01 e694570b
y: 179c 69dd2bb2 82b1a99a fead1e5f 9ee1e9ec 237c8a31
Side 2 public key
x: 83d 25d96ca6 47bb3c14 e532c984 c7e1de19 c1989701
y: 9f2 b0e30604 befef1c5 45a04f03 9191b6b7 a98d1ebb

Show that each side gets the same shared secret
key 1 is: 7b2 12d2a528 3364f521 9bcfbfe7 8eae3e56 2c38d25c
key 2 is: 7b2 12d2a528 3364f521 9bcfbfe7 8eae3e56 2c38d25c
Press any key to continue . . .
```

Figure 5.13 : Type II ONB, $m=174$ and random (curve, base point)

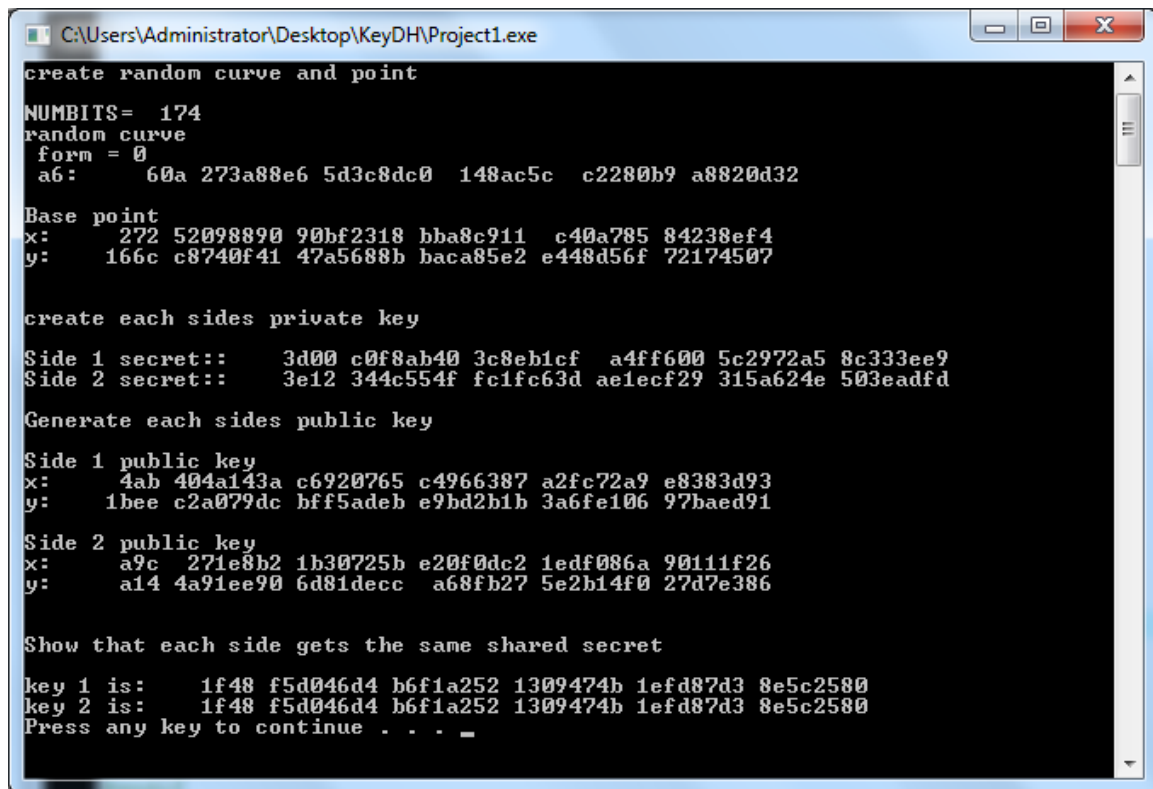
First generate random curve ($a_2 = 0$, a_6 random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.14 shows the result of computing Diffie-Hellman key exchange when key size 174 bit and Type II ONB. Different random curve and Base point are used.



```
C:\Users\Administrator\Desktop\KeyDH\Project1.exe
create random curve and point
NUMBITS= 174
random curve
form = 0
a6:      60a 273a88e6 5d3c8dc0 148ac5c c2280b9 a8820d32

Base point
x:      272 52098890 90bf2318 bba8c911 c40a785 84238ef4
y:      166c c8740f41 47a5688b baca85e2 e448d56f 72174507

create each sides private key
Side 1 secret::      3d00 c0f8ab40 3c8eb1cf a4ff600 5c2972a5 8c333ee9
Side 2 secret::      3e12 344c554f fc1fc63d ae1ecf29 315a624e 503eadfd

Generate each sides public key
Side 1 public key
x:      4ab 404a143a c6920765 c4966387 a2fc72a9 e8383d93
y:      1bee c2a079dc bff5adeb e9bd2b1b 3a6fe106 97baed91
Side 2 public key
x:      a9c 271e8b2 1b30725b e20f0dc2 1edf086a 90111f26
y:      a14 4a91ee90 6d81decc a68fb27 5e2b14f0 27d7e386

Show that each side gets the same shared secret
key 1 is:      1f48 f5d046d4 b6f1a252 1309474b 1efd87d3 8e5c2580
key 2 is:      1f48 f5d046d4 b6f1a252 1309474b 1efd87d3 8e5c2580
Press any key to continue . . .
```

Figure 5.14 : Type II ONB, $m=174$ and another random (curve, base point)

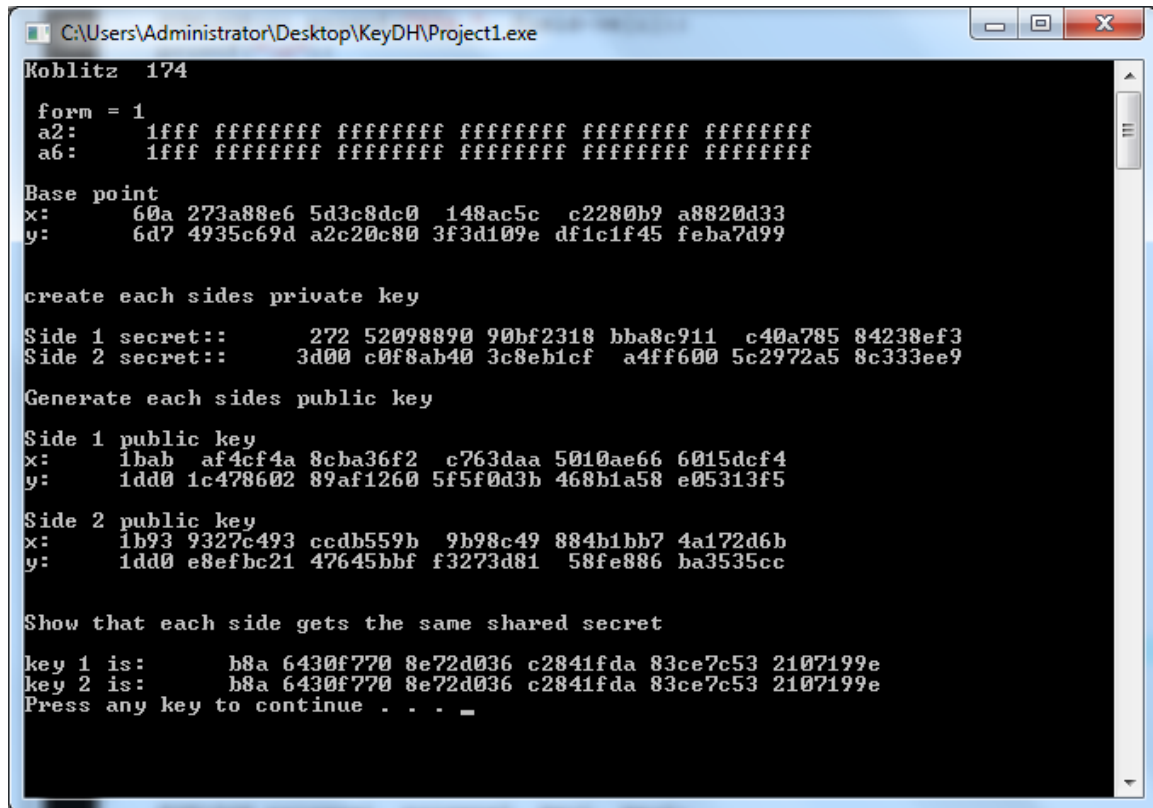
First generate random curve ($a_2 = 0$, a_6 random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.15 shows the result of computing Diffie-Hellman key exchange when key size 174 bit and Type II ONB. The koblitz curve and random Base point are used.



```
Koblitz 174
form = 1
a2: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x: 60a 273a88e6 5d3c8dc0 148ac5c c2280b9 a8820d33
y: 6d7 4935c69d a2c20c80 3f3d109e df1c1f45 feba7d99

create each sides private key
Side 1 secret:: 272 52098890 90bf2318 bba8c911 c40a785 84238ef3
Side 2 secret:: 3d00 c0f8ab40 3c8eb1cf a4ff600 5c2972a5 8c333ee9

Generate each sides public key
Side 1 public key
x: 1bab af4cf4a 8cba36f2 c763daa 5010ae66 6015dcf4
y: 1dd0 1c478602 89af1260 5f5f0d3b 468b1a58 e05313f5
Side 2 public key
x: 1b93 9327c493 cddb559b 9b98c49 884b1bb7 4a172d6b
y: 1dd0 e8efbc21 47645bbf f3273d81 58fe886 ba3535cc

Show that each side gets the same shared secret
key 1 is: b8a 6430f770 8e72d036 c2841fda 83ce7c53 2107199e
key 2 is: b8a 6430f770 8e72d036 c2841fda 83ce7c53 2107199e
Press any key to continue . . . _
```

Figure 5.15 : Type II ONB, $m=174$ and koblitz curve , base point

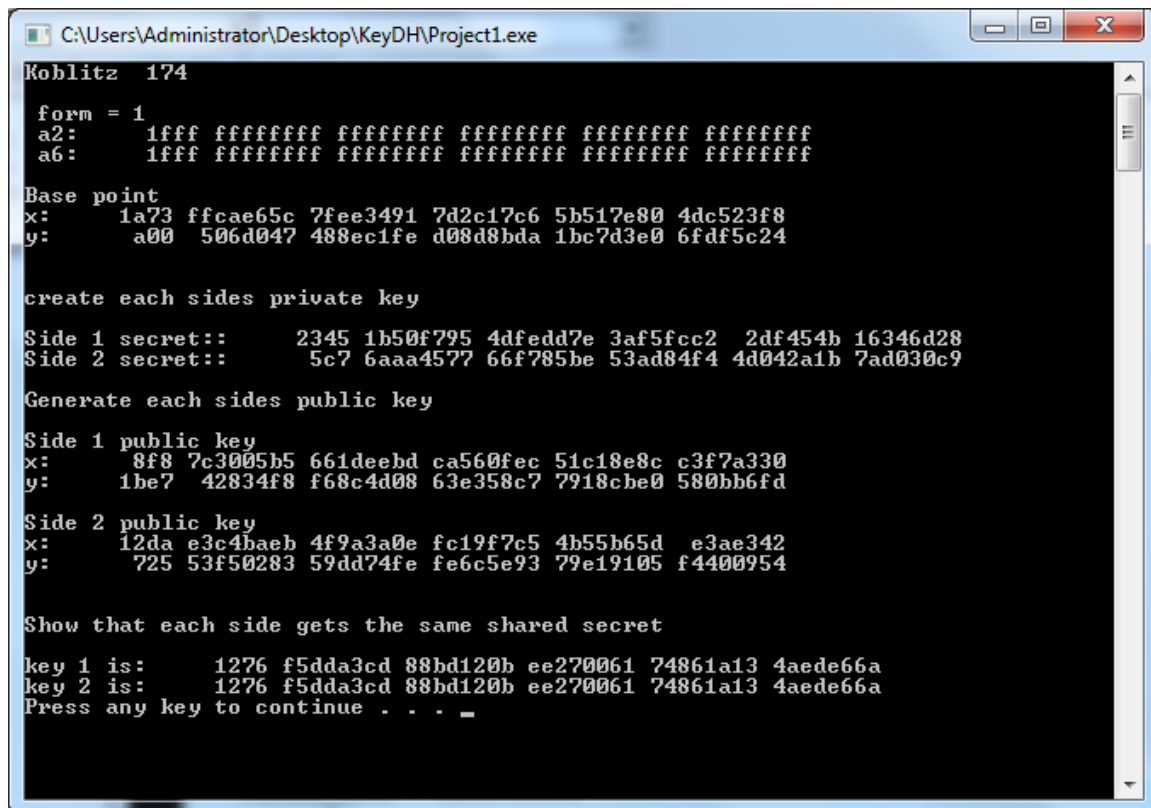
Create koblitz curve ($a_2 = 1$, $a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

Figure 5.16 shows the result of computing Diffie-Hellman key exchange when key size 174 bit and Type II ONB. The koblitz curve and different random Base point are used.



```
Koblitz 174
form = 1
a2: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6: 1fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x: 1a73 ffcae65c 7fee3491 7d2c17c6 5b517e80 4dc523f8
y: a00 506d047 488ec1fe d08d8bda 1bc7d3e0 6fdf5c24

create each sides private key
Side 1 secret:: 2345 1b50f795 4dfedd7e 3af5fcc2 2df454b 16346d28
Side 2 secret:: 5c7 6aaa4577 66f785be 53ad84f4 4d042a1b 7ad030c9

Generate each sides public key
Side 1 public key
x: 8f8 7c3005b5 661deebd ca560fec 51c18e8c c3f7a330
y: 1be7 42834f8 f68c4d08 63e358c7 7918cbe0 580bb6fd
Side 2 public key
x: 12da e3c4baeb 4f9a3a0e fc19f7c5 4b55b65d e3ae342
y: 725 53f50283 59dd74fe fe6c5e93 79e19105 f4400954

Show that each side gets the same shared secret
key 1 is: 1276 f5dda3cd 88bd120b ee270061 74861a13 4aede66a
key 2 is: 1276 f5dda3cd 88bd120b ee270061 74861a13 4aede66a
Press any key to continue . . . _
```

Figure 5.16 : Type II ONB, $m=174$ and koblitz curve , another base point

Create koblitz curve ($a_2 = 1, a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) by creating a random x value for a point and forcing it onto the curve.

Second compute a Diffie-Hellman key exchange :

- Each side create it's private key (**side 1,2 secret**) as x component of elliptic multiply.
- Each side create it's public key
- Compute shared secret key that is **same** for sender (**key1**) and receiver (**key2**).

5.3.2 ElGamal Protocol

Send data to another person using ElGamal protocol.

Send Hidden_data and Random_point to other side.

embed raw data onto the chosen curve.

Assume raw data is contained in least significant ELEMENTs of the field variable

Create the hiding value using the other person's public key

Recieve data from another person using ElGamal protocol.

Get Hidden_data, Random_point and output raw_data.

Compute hidden point using my private key and the random point.

Also, the following Figures show the sent and received data are same with different parameters:

- Type I and Type II ONB
- Key size, m , = 162,172,173 and 174.
- Public and Koblitz curves.

Figure 5.17 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 174 bit, Type II ONB, random curve and random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
create Base curve and point
NUMBITS= 174
Public curve
form = 0
a6 :      1c2d bdf5adc7 f2566ae6 7615e2d0 57247a3b 345b5acd

Base point
x :      1c4b 3ef7262 62606ab4 cd47032f 5ca8836d bb7a2e2
y :      2715 ad97b9e2 4b8333b2 4cf16272 74311b8c ddeab8d5

create side 2's private key
Side 2 secret: :      2e4b 8a3aedcb 70b617c7 38fa81b4 3e24e63f 9e36383b

Generate side 2's public key
Side 2 public key
x :      2973 b19b9b4f 858eca50 5fc155fd dbd671c5 1050283e
y :      1328 a1568892 d26c3d39 d941ca0d 46ec4390 df082789

Create message data

Hide data on curve and send from side 1 to side 2
Hidden data
x :      c34 c8cf3091 3c92cdab 2bf74bac fb4b1359 70427a2b
y :      33ee 6442809f 8a1bc0ed 4020ecc0 24166f30 8c8a2096

Random point
x :      28a3 bac9070b 6d037bd9 9e300b0f ddf9f418 76af067e
y :      228a e53ca5ab a6663598 623c925f bb5350a3 7f5bba72

Recover transmitted message
sent data      :      35a0 80514715 8eea9025 a9007e79 e86c4f27 7dcdc1a8
received data  :      35a0 80514715 8eea9025 a9007e79 e86c4f27 7dcdc1a8
Press any key to continue . . . _

```

Figure 5.17 : Type II ONB, $m=174$ and public curve , base point

Create public curve ($a_2 = 0$, $a_6 =$ random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

Figure 5.18 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 174 bit, Type II ONB, different random curve and random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
create Base curve and point
NUMBITS= 174
Public curve
form = 0
a6 :      daa 2688833d 3db0d3ff ab10d0a6 645fd377 1aa71c98

Base point
x :      20ef 98f59e9a 43f03d72 e15eb812 622f3c40 ae5b4f8d
y :      2ba 1b89e9a9 fee9ec91 9e278a62 ae6436be eb17d2a4

create side 2's private key
Side 2 secret: :      19a a6191ffb 779e4eee b755a31e 9da53fa5 74c754c2

Generate side 2's public key
Side 2 public key
x :      2903 ada5705c 7a10cce1 e4e9b4fd 9e1bf015 ca64b049
y :      2838 4ce596df d72ee79c a66bda46 762a8658 b7ce10bd

Create message data

Hide data on curve and send from side 1 to side 2
Hidden data
x :      fad 371d7ede 68bc1407 7bb5cb2a e2874f49 71b172ca
y :      1c71 3ef873a4 140f8775 be278e49 cb1f68e4 1081d855

Random point
x :      2e80 f2005df2 51662e9f cba3520 2fa761c5 9b6b42ed
y :      2ca2 5a149277 a281cdc3 533120ec a675421 27a25600

Recover transmitted message
sent data      :      477 c0ab7e1a dd75300a d5cd44e5 8a62d193 41ee19aa
received data  :      477 c0ab7e1a dd75300a d5cd44e5 8a62d193 41ee19aa
Press any key to continue . . . _

```

Figure 5.18 : Type II ONB, $m=174$ and different public curve , base point

Create public curve ($a_2 = 0$, $a_6 =$ random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

Figure 5.19 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 174 bit, Type II ONB, koblitz curve and random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
Koblitz 174
form = 1
a2 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x : 2021 dcf209af c10aba20 25a94158 9425f829 a98a0a1f
y : 26f0 29ad7171 87871d5d 554bce7e 7bd06fba 784bb7d2

create side 2's private key
Side 2 secret: : d57 ed9fd78f 52040cfb 8c79dee5 d2c6a0fa f4e7864e

Generate side 2's public key
Side 2 public key
x : 242d 1e310195 738df544 f9c55a4 db7d1b68 76357037
y : 2377 ff304a87 e2723ed 1b8cfc8b 63130b47 9d2ae8a6

Create message data

Hide data on curve and send from side 1 to side 2
Hidden data
x : bbd dd9a9bcb 931961e9 6d82e292 e2e9aae4 4429c011
y : 3991 dbbf9462 e8cbf7e7 42e0bb37 3e47d44a 39b8d987

Random point
x : 3685 b2973740 dba18529 1bfe3748 4a7ae0ce 3ccf3d0
y : 34d0 ae1cdd68 7c73e51 e916fd9d 47e7f7dc 3fdbd260

Recover transmitted message
sent data : 41b a9804eb3 23ba6263 4574031b 1348a6d3 f0affafb
received data : 41b a9804eb3 23ba6263 4574031b 1348a6d3 f0affafb
Press any key to continue . . .

```

Figure 5.19 : Type II ONB, $m=174$ and Koblitz curve , base point

Create Koblitz curve ($a_2 = 1, a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

Figure 5.20 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 174 bit, Type II ONB, koblitz curve and different random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
Koblitz 174
form = 1
a2 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x : 2d12 a977695c 8db62689 ba59afc4 2f259729 db11603d
y : 3853 1120e07 cff9ee4d 96aa11a5 8ac58019 9911ca3b

create side 2's private key
Side 2 secret: : 3914 73da303e d84cc38f f90b5d1c 25d8e46a 8e9499c3

Generate side 2's public key
Side 2 public key
x : 2d95 42c9f62d 51a0e9df 871c0b5f 644eb883 ab323ab9
y : 155f 6c0f5957 1090591a d9b5cb2d 6b10965c cc4544e4

Create message data

Hide data on curve and send from side 1 to side 2
Hidden data
x : 2689 381b8e86 696b7ac0 f249be34 10f6053 d5b1b1ab
y : 38e1 6840c078 11f7dae5 ab53b7a0 86ca19f4 a514f096

Random point
x : 9a6 d1e1ec4a 9fd267f7 b23ca64f 2fc03e7e 19dfc40f
y : 334a 314a360d 2bb1751f 46f5c1f9 956562f6 6f07db3b

Recover transmitted message
sent data : 1f01 c25687d8 5fe42165 d92f5d49 6ad823d3 7b415f94
received data : 1f01 c25687d8 5fe42165 d92f5d49 6ad823d3 7b415f94
Press any key to continue . . . _

```

Figure 5.20 Type II ONB, $m=174$ and public curve , different base point

Create Koblitz curve ($a_2 = 1, a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

Figure 5.21 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 173 bit, Type II ONB, koblitz curve and random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
Koblitz 173
form = 1
a2 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x : 5f6 42a9b9fa 3c1cfa0d bf6b9c9c 5b53b81f ec39b359
y : 166 4132f9a1 b45c620d db4005a2 8fa189dc 55f4b427

create side 2's private key
Side 2 secret: : 1c87 6438891 c703f234 3f10bf4b 7c69a948 36384b76

Generate side 2's public key
Side 2 public key
x : 255c 775f06d 29d998fa f10fbb97 d97c36f8 696dafce
y : d52 1092fca6 e7a27c6a c116ba75 b8cab409 4a4a72c2

Create message data

Hide data on curve and send from side 1 to side 2
Hidden data
x : 2ae0 399c1133 f669637d 1d980521 3b57b91e c909776f
y : 6dc cc76a010 8bb288de 6fab0751 5dd5a22 92443a3d

Random point
x : 1fea 8809bd69 d5c29327 e6c10695 7d2e84be e24e383a
y : 24b7 295a3af0 209c31c 67324b6c 2447212f d0d89e33

Recover transmitted message
sent data : ff7 8997c410 d03e9360 ae30dd1b 6abc5775 4addf9c9
received data : ff7 8997c410 d03e9360 ae30dd1b 6abc5775 4addf9c9
Press any key to continue . . .

```

Figure 5.21 : Type II ONB, $m=173$ and Koblitz curve , base point

Create public curve ($a_2 = 1, a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

Figure 5.22 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 173 bit, Type II ONB, koblitz curve and different random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
Koblitz 173
form = 1
a2 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x : 3b3 f9cb4b9f ead4dca6 fe195f1a 4c57fdf7 3df9c4b6
y : 2f8c b63b1ae5 3540c8c1 638fe6b8 87c4350a 9e7cd204

create side 2's private key
Side 2 secret: : 1849 7375d518 7f18bae4 915ebccc 99118043 91b94ae3

Generate side 2's public key
Side 2 public key
x : 380c 159ba3be 1f7ae663 85675599 a466bb27 3324e2c4
y : 24fe b0645e5c 339bac1a 7a2cdab2 a7769366 acf0c61c

Create message data

Hide data on curve and send from side 1 to side 2
Hidden data
x : 1bf 8d545961 8b8d91a1 bd1e1f67 ceae9221 87dee90d
y : 2d7b 66c6c21e bd4cccc9 40484b90 2fda6196 d593b311

Random point
x : 3758 1352ecc9 7f1fff0f 2edb0221 ebced633 2a9bd65b
y : 1218 b35c1023 d9077c1 477ce41b b6b101af d119bfb0

Recover transmitted message
sent data : 1bd8 4c6f8c0d 2961be7e b2c5427b e092ba18 2160aef5
received data : 1bd8 4c6f8c0d 2961be7e b2c5427b e092ba18 2160aef5
Press any key to continue . . .

```

Figure 5.22 : Type II ONB, $m=173$ and Koblitz curve , different base point

Create public curve ($a_2 = 1, a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

Figure 5.23 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 172 bit, Type I ONB, koblitz curve and random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
Koblitz 172
form = 1
a2 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x : c2d bdf5adc7 f2566ae6 7615e2d0 57247a3b 345b5ad2
y : 1b85 c7bda40d 12eab0af 1a3cef6 babfd9f6 a1f91cec

create side 2's private key
Side 2 secret: : c4b 3ef7262 62606ab4 cd47032f 5ca8836d bb7a2e2

Generate side 2's public key
Side 2 public key
x : f71 385d3b92 91a8406 f6f2593f 922efa6d 862b2180
y : 12c2 3e4c049b dec68f d545fa6f 48108fe 96ede337

Create message data

Hide data on curve and send from side 1 to side 2
Hidden data
x : 312 ca6abf6d 4f7607e0 bf137d19 ac92c0df e3c11470
y : 29fb 42d2ed61 13fa5406 26603965 3344bcf6 f051e407

Random point
x : 374a 839ec4b4 8bdc9f81 f1419304 67ad6ff0 56a9c16b
y : 4fd e7e385fe 391788f4 5e5a0919 995a0cbc 72c2fa79

Recover transmitted message
sent data : e4b 8a3aedcb 70b617c7 38fa81b4 3e24e63f 9e36383b
received data : e4b 8a3aedcb 70b617c7 38fa81b4 3e24e63f 9e36383b
Press any key to continue . . . _

```

Figure 5.23 : Type I ONB, $m=172$ and Koblitz curve , base point

Create Koblitz curve ($a_2 = 1, a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

Figure 5.24 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 172 bit, Type I ONB, koblitz curve and different random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
Koblitz 172
form = 1
a2 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff
Base point
x : 944 3dfbb222 466500d3 e668b035 eee50a91 472dcbf9
y : 34e2 49cb8606 ff8e8f3c 29de5249 30676121 545bcc99

create side 2's private key
Side 2 secret: : fb4 a52f5860 d1576cd6 4aa844a8 59c0eb52 48c1391d
Generate side 2's public key
Side 2 public key
x : 1d98 bf1a321b 56b7394f 659f4de0 d726b32 136f2ada
y : 1add 2c4fa23f 295f6acc e09dacf5 abd91984 401d622d

Create message data
Hide data on curve and send from side 1 to side 2
Hidden data
x : 1b44 7ef067e0 fcc2a98 8abb3033 7dc6f8d5 8a7ce6c4
y : d40 a26a3dba f88eba03 e50f356b 1208b8 9d759ce4
Random point
x : 3158 e887c46d 48847b7d 201b5b1c f584c8ca 85386e57
y : d5a 7b5ac386 7f333ce2 b54ae49e 8dc76241 1f3bf863

Recover transmitted message
sent data : 659 5927f92e 9f18e39b 1931e7f7 bf193341 64852ab6
received data : 659 5927f92e 9f18e39b 1931e7f7 bf193341 64852ab6
Press any key to continue . . .

```

Figure 5.24 : Type I ONB, $m=172$ and Koblitz curve , different base point

Create Koblitz curve ($a_2 = 1, a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

Figure 5.25 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 162 bit, Type I ONB, random curve and random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
create Base curve and point
NUMBITS= 162
Public curve
form = 0
a6 :      1 bdf5adc7 f2566ae6 7615e2d0 57247a3b 345b5acd

Base point
x :      3 3ef7262 62606ab4 cd47032f 5ca8836d bb7a2e2
y :      1c6b d5db2888 46e9a9a8 2a311094 def2f768 5149fab1

create side 2's private key
Side 2 secret: :      3 8a3aedcb 70b617c7 38fa81b4 3e24e63f 9e36383b

Generate side 2's public key
Side 2 public key
x :      133d 76e2ada7 5c7dd6e e78f4f1b ae84c19b cebebeba
y :      3fec 5e846076 151eac80 69bc4288 812e8218 d08e8e06

Create message data

Hide data on curve and send from side 1 to side 2
Hidden data
x :      14c7 144a046d d40852da fa235fdc 2979c3a ddc0a3a8
y :      1c75 dda26328 32cfced3 a33de2be 9e9bcfa6 1bee775e

Random point
x :      371f 7e1cc2c7 6df286e6 b88bb064 cb0a30da 157c58a3
y :      2acf e4b4a5a5 71216cd0 eaf8a6ef cb430b80 2de4625

Recover transmitted message
sent data      :      0 80514715 8eea9025 a9007e79 e86c4f27 7dcdc1a8
received data  :      0 80514715 8eea9025 a9007e79 e86c4f27 7dcdc1a8
Press any key to continue . . . _

```

Figure 5.25 : Type I ONB, $m=162$ and public curve , base point

Create public curve ($a_2=0$, a_6 = random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

Figure 5.26 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 162 bit, Type I ONB, different random curve and different random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
create Base curve and point
NUMBITS= 162
Public curve
form = 0
a6 :      2 a5774780 adccb864 2cfe67b9 df0cd476 a9178807

Base point
x :      1 54929439 66d1602 58e1f037 d8c33b88 f0cfe937
y :      15d5 94c00ba0 a597af74 55dcdd64 11f0f8f9 7874e7b3

create side 2's private key
Side 2 secret: :      3 602e65a6 fab46e6d cc81949a 802784f ab31c0b2

Generate side 2's public key
Side 2 public key
x :      20c4 346ba81a 216254fe 9f013ef2 2fdfe1ab 60a33fc8
y :      3812 c3540669 cc55b559 a4111882 1639525f 1a3b6a9b

Create message data

Hide data on curve and send from side 1 to side 2
Hidden data
x :      389b be1c8936 debea747 3af1b41c 8444c9f d6717b28
y :      7ad 54bbfe24 146fff6e 5a5641a2 2d852d73 7da9a25d

Random point
x :      ff6 dde23a45 cf5bf768 1a20a9c5 a1e238c3 998878cc
y :      baa 62f1e5a9 653c4c49 485b7011 7e01b0aa 4941c49d

Recover transmitted message
sent data      :      3 cef145e1 f138495f 50df5b5f 8b5fc71a 630265d0
received data  :      3 cef145e1 f138495f 50df5b5f 8b5fc71a 630265d0
Press any key to continue . . .

```

Figure 5.26 : Type I ONB, $m=162$ and different public curve , base point

Create public curve ($a_2 = 0$, $a_6 =$ random bit pattern) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

Figure 5.27 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 162 bit, Type I ONB, koblitz curve and random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
Koblitz 162
form = 1
a2 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x : 3 f9cb4b9f ead4dca6 fe195f1a 4c57fdf7 3df9c4b5
y : 7f1 fd444b28 3f822a0f e514cc6a 711f90c6 4ce36e73

create side 2's private key
Side 2 secret: : 1 7375d518 7f18bae4 915ebccc 99118043 91b94ae3

Generate side 2's public key
Side 2 public key
x : 1a2c 60926499 a5566afc 94453260 6f56c5c0 180c30a
y : e20 bbf21175 ba02003f e7e0240f 939b905b 85a4e79f

Create message data

Hide data on curve and send from side 1 to side 2
Hidden data
x : ec1 2a8d352f b36b1a6b 38814550 79059c48 6e7d4bb
y : 1aa 14272050 811e3e75 2302a2a5 9e4c6e13 2d50829f

Random point
x : 2335 14c9e6f0 7cb2ff63 bec6e64b ff574ad2 f200d2d0
y : 3206 43d17631 12c0cdcd 5013f979 a669ef5e 1d3e44f0

Recover transmitted message
sent data : 0 4c6f8c0d 2961be7e b2c5427b e092ba18 2160aef5
received data : 0 4c6f8c0d 2961be7e b2c5427b e092ba18 2160aef5
Press any key to continue . . .

```

Figure 5.27 : Type I ONB, $m=162$ and Koblitz curve , base point

Create koblitz curve ($a_2 = 1, a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

Figure 5.28 shows the sent data and received data are the same using ElGamal protocol with parameters (key size 162 bit, Type I ONB, koblitz curve and different random Base point).

```

C:\Users\Administrator\Desktop\SRData\Project1.exe
Koblitz 162
form = 1
a2 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff
a6 : 3fff ffffffff ffffffff ffffffff ffffffff ffffffff

Base point
x : 0 90ac38bd ee1979db c6fc9994 8fd87fa6 6c8e3512
y : 4b7 af08aef4 f4c595a8 f078cf9b 37266f4 ca469e61

create side 2's private key
Side 2 secret: : 3 4ac77f81 ae71bee 9b9edd 59df045f 78699538

Generate side 2's public key
Side 2 public key
x : 3df7 7252a4c8 f8d21a58 481316b2 315252a0 ffe13aec
y : 1706 5fbeaad5 e66bfe38 e15a8bab ad539b24 61af04b3

Create message data
Hidden data on curve and send from side 1 to side 2
Hidden data
x : 2f5f 37770a02 a9357fc0 3d9c6b39 bd030c00 b8b22d44
y : 3504 38e5781d f215ee87 e4659485 57b650af 62a0f34

Random point
x : 1efe c1828d2f d4b51008 d5002aee a434d27b a1aeb5e3
y : 763 e8b5d5e3 bba3fc15 8d1d4c58 cc24a8ba 57cc8b2b

Recover transmitted message
sent data : 3 2b368ee7 43bcc89a b29c7765 aeac7eb5 894afb69
received data : 3 2b368ee7 43bcc89a b29c7765 aeac7eb5 894afb69
Press any key to continue . . . _

```

Figure 5.28 : Type I ONB, $m=162$ and koblitz curve , different base point

Create koblitz curve ($a_2 = 1$, $a_6 = 1$) for equation 3.9.

Then generate a random point on a given curve (**Base point**) .

Create side2 public (send to another side) and private keys.

Side 1 create a message **Hidden data** , random point and send to another side.

Side 2 recover the data . show that both **sent data** and **received data** are the same.

5.4 Comments on the results

- When large key size is used, the computation time is increased.
- In some cases the data sent are different from the received data, because the raw data didn't fit on the curve and the embedding routine was instructed to increment offset 0.
- The calculation of Lambda matrix takes 16,500 byte, The precomputing of it may speed the code.
- The word size used in code is 32 bit.
- The code can run on WINS NG 2.0 Node or iPAQ Node.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

A fast testing of elliptic curves is presented for microcontroller based sensor network that use a optimal normal basis representation.

Two approaches are presented Type I and Type II normal basis.

Testing results show less memory size even the key length is higher. Compared with Bläß and Zitterbart code and memory size the execution time is not as critical. consumes less memory and offers faster operation (less execution time).

The key for efficiency are memory optimizations like moving data from expensive RAM to ROM as well as a precomputation of *base points* for faster execution.

The code runs on an Intel Pentium core dual processor, the results shows that the algorithm is faster and can use small memory size.

6.2 Future work

The following can be done:

- Improving the efficiency of inversion routine by using another algorithm will improve overall throughput.
- Digital Signature Standards (DSS) algorithm can implemented easily because most functions needed are existed.
- Remove some variable that not changed out of loops.

REFERENCES

- [1] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In M. Joye and J. J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems — CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 119–132. Springer-Verlag, 2004.
- [2] D. Brown. SEC 1: Elliptic Curve Cryptography. Certicom Research, February 2005. Working Draft Version 1.5.
- [3] M. Scott and P. Szczechowiak. Optimizing Multiprecision Multiplication for Public Key Cryptography. *Cryptology ePrint Archive*, Report 2007/299, 2007. <http://eprint.iacr.org/>.
- [4] L. Uhsadel, A. Poschmann, and C. Paar. Enabling Full-Size Public-Key Algorithms on 8-Bit Sensor Nodes. In F. Stajano, C. Meadows, S. Capkun, and T. Moore, editors, *Security and Privacy in Ad-hoc and Sensor Networks*, volume 4572 of *Lecture Notes in Computer Science*, pages 73–86. Springer-Verlag, 2007.
- [5] D.J. Malan, M. Welsh, and M.D. Smith. A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography. In *Sensor and Ad Hoc Communications and Networks*, 2004, pages 71–80, 2004.
- [6] Z.J. Shi and H. Yan. Software Implementations of Elliptic Curve Cryptography. *International Journal of Network Security*, 7(2):157–166, 2008.
- [7] E.-O. Blaß and M. Zitterbart. Efficient Implementation of Elliptic Curve Cryptography for Wireless Sensor Networks. Technical Report TM-2005-1, Insitute of Telematics, University of Karlsruhe, 3 2005.
- [8] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab. Nanoecc: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. In R. Verdone, editor, *Wireless Sensor Networks — EWSN 2008*, volume 4913 of *Lecture Notes in Computer Science*, pages 305–320. Springer-Verlag, 2008.
- [9] S.C. Seo, D. G. Hand, and S. Hong. TinyECCK: Efficient Elliptic Curve Cryptography Implementation over $GF(2^m)$ on 8-bit MICAz Mote. *Cryptology ePrint Archive*, Report 2008/XXX, 2008. <http://eprint.iacr.org/>.
- [10] R. Lidl and H. Niederreiter, “Introduction to finite fields and their applications”, 1997.
- [11] E. O. Blaß and M. Zitterbart, “*Towards Acceptable Public-Key Encryption in Sensor Networks*,”, 2005.

- [12] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “*Wireless sensor networks: a survey*,” , IEEE Communications Magazine, 2002.
- [13] Feng Zhao, Leonidas Guibas, “*Wireless Sensor Networks*”, Morgan Kaufmann publishers, 2007.
- [14] David W. Carman, Peter S. Kruus, and Brian J. Matt, “*Constraints and Approaches for Distributed Sensor Network Security*”, 2010.
- [15] Perrig, J. Stankovic, and D. Wagner, “*Security in wireless sensor networks*”, 2004.
- [16] F. Amin, A. H. Jahangir, and H. Rasifard, “*Analysis of Public-Key Cryptography for Wireless Sensor Networks Security*”, 2008.
- [17] A. Perrig, R. Szewczyk, V. Wen, D. Culler, J. D. Tygar, “*SPINS: Security Protocols for Sensor Networks*”, in Wireless Networks Journal, 2002.
- [18] C. Karlof, N. Sastry, and D. Wagner, “*Tinysec: a link layer security architecture for wireless sensor networks*”. In Proceedings of the 2nd international conference on Embedded networked sensor systems, pages 162–175. ACM, 2004.
- [19] M. Luk, G. Mezzour, A. Perrig, and V. Gligor. Minisec, “*a secure sensor network communication architecture*”, In Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on, pages 479–488. IEEE, 2007.
- [20] IEEE. Ieee 802.15.4 , “*wireless medium access control (mac) and physical layer (phy) specifications for low-rate wireless personal area networks (lr-wpans). Technical report*”, Park Avenue, New York, USA: IEEE, 2011.
- [21] William Stallings, “*Cryptography and Network Security principles and practices*”.
- [22] N. Koblitz, “*Elliptic curve cryptosystems*”, Mathematics ofComputation, Vol. 48, 1987.
- [23] V.S. Miller, “*Use of Elliptic Curves in Cryptography*”, Advances in Cryptology CRYPTO 85, 1986.
- [24] Alfred J. Menezes, “*Elliptic curve public key cryptosystems*”, 2002.
- [25] Michael Rosing, “*Implementing Elliptic curve cryptography*”, 1999.

- [26] Darrel R. Hankerson, Scott A. Vanstone, Alfred J. Menezes, “*Guide to elliptic curve cryptography*” 2004.
- [27] R. C. Mullin, I. M. Onyszchuk, S. A. Vanstone, and R. M. Wilson, “ *Optimal Normal Basis in $GF(2^n)$* ,” in *Discrete Applied Math*, vol. 22 (Amsterdam: Elsevier science Publisher/North-Holland, 1988), 149-161.
- [28] N. Koblitz, “*CM-Curves with Good Cryptographic Properties*”. 11th Annual International Cryptology Conference on Advances in Cryptology, 1991.
- [29] J. A. Solinas, “ *An Improved Algorithm for Arithmetic on a Family of Elliptic Curves*”. 17th Annual International Cryptology Conference on Advances in Cryptology, 1997.
- [30] N. Koblitz, “ *Introduction to Elliptic Curves and Modular Forms*”, 1993.
- [31] Diffie, W., Hellman, M. E.: *New Directions in Cryptography*. IEEE Transactions on Information Theory.
- [32] El Gamal, T., “*A public key cryptosystem and a signature scheme based on discrete logarithms*”. Proceedings of CRYPTO 84 on Advances in cryptology, 1985.
- [33] H. Yan and Z. Shi, “*Studying software implementations of elliptic curve cryptography*” Third International Conference on Information Technology, 2006.
- [34] H. Eberle, A. Wander, N. Gura, S. Chang-Shantz, and V. Gupta, “*Architectural Extensions for Elliptic Curve Cryptography over $GF(2^m)$ on 8-bit Microprocessors*,” 16th International Conference on Application-Specific Systems, Architecture and Processors, 2005.
- [35] Ian F. Blake, Gadiel Seroussi, Nigel Paul Smart, “*Advances in elliptic curve cryptography*” 2005.
- [36] Henri Cohen, Gerhard Frey, Roberto Avanzi, “*Handbook of elliptic and hyperelliptic curve cryptography*” 2006.
- [37] D. Beauregard, “*Efficient Algorithms for Implementing Elliptic Curve Public Key Schemes*”(Master’s thesis, Worcester Polytechnic Institute, Worcester).

- [38] Biagioni E., and Bridges K., “*The Application of Remote Sensor Technology to assist the recovery of rare and endangered species*,” Special issue on Distributed Sensor Networks for the International Journal of High Performance Computing Applications, Vol. 16, N. 3, August 2002.
- [39] John Paul Walters, Zhengqiang Liang, Weisong Shi, and Vipin Chaudhary, “*Wireless Sensor Network Security: A Survey*”, 2006.
- [40] Tanveer Zia and Albert Zomaya, “*A Security Framework for Wireless Sensor Networks*”, IEEE Sensors Applications Symposium Houston, 2006.
- [41] H. Chan, A. Perrig, and D. Song, “*Random Key Predistribution Schemes for Sensor Networks*”.
- [42] J. Undercoffer, S. Avancha, A. Joshi, and J. Pinkston, “*Security for Sensor Networks*” ,2002.
- [43] Jeffery Undercoffer, Sasikanth Avancha, Anupam Joshi and John Pinkston, “*Security for Sensor Networks*”, 2004.
- [44] Suat Ozdemir, and Prashant Nair,”*Wireless Sensor Network Security*”, 2003.
- [45] David Boyle, Thomas Newe, “*Securing Wireless Sensor Networks: Security Architectures*”,2008.
- [46] Kakali Chatterjee, Asok De, and Daya Gupta, “*Software Implementation of Curve based Cryptography for Constrained Devices*”. International Journal of Computer Applications, 24(5):18–23, 2011.
- [47] F Amin, A H Jahangir, and H Rasifard, “*Analysis of Public-Key Cryptography for Wireless Sensor Networks Security*”. World Academy of Science, Engineering and Technology, 31(July):530–535, 2008.

APPENDICES

APPENDIX A

COMPARISON OF THREE SENSOR PLATFORMS

	WINS NG 2.0 Node	iPAQ with 802.11 and A/D Cards in Sleeve	Berkeley MICA Mote
Size (cm^3)	5300	600	40
Weight (g) (including battery)	5400	350	70
Battery capacity (kJ)	300	35	15
Sensors	Off-board	Microphone & light sensors integrated, others off-board	Integrated on PCB: Acceleration, temperature, light, sound
Memory	32 MB RAM 32 MB flash	64 MB RAM 32 MB flash	4 KB RAM 128 KB flash
CPU	Hitachi SH4	StrongARM or XScale	ATmega 103L
Operating system	linux	WinCE or Linux	TinyOS
Processing capability	400 MIPS/ 1.4GFLOPS	240 MIPS	4 MIPS
Radio range	100 m	100 m	30 m

APPENDIX B

OPTIMAL NORMAL BASIS FOR F_{2^m} .

m (field size), T (Type)

m	T	m	T	m	T	m	T	m	T
82	I,II	148	I,II	226	I,II	306	II	393	II
83	II	155	II	230	II	309	II	398	II
86	II	158	II	231	II	316	I,II	410	II
89	II	162	I,II	233	II	323	II	411	II
90	II	172	I,II	239	II	326	II	413	II
95	II	173	II	243	II	329	II	414	II
98	II	174	II	245	II	330	II	418	I,II
99	II	178	I,II	251	II	338	II	419	II
100	I,II	179	II	254	II	346	I,II	420	I,II
105	II	180	I,II	261	II	348	I,II	426	II
106	I,II	183	II	268	I,II	350	II	429	I,II
113	II	186	II	270	II	354	II	431	II
119	II	189	II	273	II	359	II	438	II
130	I,II	191	II	278	II	371	II	441	II
131	II	194	II	281	II	372	I,II	442	I,II
134	II	196	I,II	292	I,II	375	II	443	II
135	II	209	II	293	II	378	I,II	453	II
138	I,II	210	I,II	299	II	386	II	460	I,II
146	II	221	II	303	II	388	I,II	466	I,II

APPENDIX C

PROGRAM CODE IN C LANGUAGE

```
/****** field2n.h *****/

#define WORDSIZE      (sizeof(int)*8)

#define NUMBITS        174

#define TYPE2

#ifdef TYPE2

#define field_prime    ((NUMBITS<<1)+1)

#else

#define field_prime (NUMBITS+1)

#endif

#define NUMWORD        (NUMBITS/WORDSIZE)

#define UPRSHIFT      (NUMBITS%WORDSIZE)

#define MAXLONG        (NUMWORD+1)

#define MAXBITS        (MAXLONG*WORDSIZE)

#define MAXSHIFT      (WORDSIZE-1)

#define MSB            (1L<<MAXSHIFT)

#define UPRBIT         (1L<<(UPRSHIFT-1))

#define UPRMASK        (~(-1L<<UPRSHIFT))

#define SUMLOOP(i)    for(i=0; i<MAXLONG; i++)

typedef short int INDEX;

typedef unsigned long ELEMENT;

typedef struct {

    ELEMENT      e[MAXLONG];

} FIELD2N;

/****** elliptic.h *****/

typedef struct
```

```

{
    INDEX form;

    FIELD2N a2;

    FIELD2N a6;
} CURVE;

typedef struct
{
    FIELD2N x;

    FIELD2N y;
} POINT;

/***** onb.c *****/

#include "field2n.h"

void rot_left();

void rot_right();

void null();

void copy();

void genlambda();

void genlambda2();

void opt_mul();

void opt_inv();

INDEX log_2();

INDEX Lambda[2][field_prime];

INDEX lg2_m;

void rot_left(a)

FIELD2N *a;

{
    INDEX i;

```

```

ELEMENT bit,temp;

bit = (a->e[0] & UPRBIT) ? 1L : 0L;

for (i=NUMWORD; i>=0; i--) {

    temp = (a->e[i] & MSB) ? 1L : 0L;

    a->e[i] = ( a->e[i] << 1) | bit;

    bit = temp;

}

a->e[0] &= UPRMASK;
}

void rot_right(a)

FIELD2N *a;

{

    INDEX i;

    ELEMENT bit,temp;

    bit = (a->e[NUMWORD] & 1) ? UPRBIT : 0L;

    SUMLOOP(i) {

        temp = ( a->e[i] >> 1) | bit;

        bit = (a->e[i] & 1) ? MSB : 0L;

        a->e[i] = temp;

    }

    a->e[0] &= UPRMASK;

}

void null(a)

FIELD2N *a;

{

    INDEX i;

    SUMLOOP(i) a->e[i] = 0;

}

```

```

void copy (a,b)

FIELD2N *a,*b;

{
    INDEX i;
    SUMLOOP(i) b->e[i] = a->e[i];
}

INDEX log_2( x)

ELEMENT x;

{
    INDEX k, lg2;
    ELEMENT ebit, bitsave, bitmask;

    lg2 = 0;
    bitsave = x;
    k = WORDSIZE/2;
    bitmask = -1L<<k;
    while (k)
    {
        ebit = bitsave & bitmask;
        if (ebit)
        {
            lg2 += k;
            bitsave = ebit;
        }
        k /= 2;
        bitmask ^= (bitmask >> k);
    }
    return( lg2);
}

```

```

}

void genlambda()
{
    INDEX i, logof, n, index;

    INDEX log2[field_prime], twoexp;

    for (i=0; i<field_prime; i++) log2[i] = -1;

    twoexp = 1;

    for (i=0; i<field_prime; i++)
    {
        log2[twoexp] = i;

        twoexp = (twoexp << 1) % field_prime;
    }

    n = (field_prime - 1)/2;

    Lambda[0][0] = n;

    for (i=1; i<field_prime; i++)

        Lambda[0][i] = (Lambda[0][i-1] + 1) % NUMBITS;

    Lambda[1][0] = -1;

    Lambda[1][1] = n;

    Lambda[1][n] = 1;

    for (i=2; i<=n; i++) {

        index = log2[i];

        logof = log2[field_prime - i + 1];

        Lambda[1][index] = logof;

        Lambda[1][logof] = index;

    }

    Lambda[1][log2[n+1]] = log2[n+1];

    lg2_m = log_2((ELEMENT)(NUMBITS - 1));

```

```

}

void genlambda2()
{
    INDEX i, logof[4], n, index, j, k;

    INDEX log2[field_prime], twoexp;

    twoexp = 1;
    for (i=0; i<NUMBITS; i++)
    {
        log2[twoexp] = i;
        twoexp = (twoexp << 1) % field_prime;
    }
    if (twoexp == 1)
    {
        twoexp = 2*NUMBITS;
        for (i=0; i<NUMBITS; i++)
        {
            log2[twoexp] = i;
            twoexp = (twoexp << 1) % field_prime;
        }
    }
    else
    {
        for (i=NUMBITS; i<field_prime-1; i++)
        {
            log2[twoexp] = i;
            twoexp = (twoexp << 1) % field_prime;
        }
    }
}

```

```

Lambda[0][0] = 1;
Lambda[1][0] = -1;
n = (field_prime - 1)/2;
twoexp = 1;
for (i=1; i<n; i++)
{
    twoexp = (twoexp<<1) % field_prime;
    logof[0] = log2[field_prime + 1 - twoexp];
    logof[1] = log2[field_prime - 1 - twoexp];
    logof[2] = log2[twoexp - 1];
    logof[3] = log2[twoexp + 1];
    k = 0;
    j = 0;
    while (k<2)
    {
        if (logof[j] < n)
        {
            Lambda[k][i] = logof[j];
            k++;
        }
        j++;
    }
}

lg2_m = log_2((ELEMENT)(NUMBITS - 1));

void opt_mul(a, b, c)
FIELD2N *a, *b, *c;
{

```

```

INDEX i, j;

INDEX k, zero_index, one_index;

ELEMENT bit, temp;

FIELD2N      amatrix[NUMBITS], copyb;

null(c);

copy(b, &copyb);

copy( a, &amatrix[0]);

for (i = 1; i < NUMBITS; i++)
{
    copy( &amatrix[i-1], &amatrix[i]);
    rot_right( &amatrix[i]);
}

zero_index = Lambda[0][0];

SUMLOOP (i) c->e[i] = copyb.e[i] & amatrix[zero_index].e[i];

for (j = 1; j<NUMBITS; j++)
{
    rot_right( &copyb);
    zero_index = Lambda[0][j];
    one_index = Lambda[1][j];
    SUMLOOP (i) c->e[i] ^= copyb.e[i] &
                        (amatrix[zero_index].e[i] ^ amatrix[one_index].e[i]);
}
}

void opt_inv(a, result)

FIELD2N *a, *result;

{
    FIELD2N      shift, temp;

    INDEX m, s, r, rsft;

```

```

s = lg2_m - 1;
copy( a, result);
m = NUMBITS - 1;
while (s >= 0)
{
    r = m >> s;

    copy( result, &shift);

    for (rsft = 0; rsft < (r>>1); rsft++) rot_left( &shift);

    opt_mul( result, &shift, &temp);

    if ( r&1 )
    {
        rot_left( &temp);

        opt_mul( &temp, a, result);

    }

    else copy( &temp, result);

    s--;

}

rot_left(result);

}

/***** elliptic.c *****/

#include <stdio.h>

#include "field2n.h"

#include "elliptic.h"

extern opt_inv(), rot_left(), rot_right(), null(), opt_mul(), copy();

extern genlambda(), genlambda2();

int opt_quadratic();

void fofx();

void esum ();

```

```

void edbl ();

void esub ();

void copy_point ();

void elptic_mul();

void one(FIELD2N*);

int opt_quadratic(a, b, y)

FIELD2N *a, *b, *y;

{
    INDEX i, el, bits;

    FIELD2N z, k, a2;

    ELEMENT r, t, mask;

    r = 0;

    SUMLOOP(i) r |= a->e[i];

    if (!r)
    {
        copy( b, &y[0]);

        rot_right( &y[0]);

        copy( &y[0], &y[1]);

        return(0);
    }

    opt_inv( a, &a2);

    rot_left(&a2);

    opt_mul( b, &a2, &k);

    rot_right(&k);

    r = 0;

    SUMLOOP(i) r ^= k.e[i];

    mask = -1L;

    for (bits = WORDSIZE/2; bits > 0; bits >>= 1)

```

```

{
    mask >>= bits;

    r = ((r & mask) ^ (r >> bits));
}

if (r)
{
    null(&y[0]);
    null(&y[1]);
    return(1);
}

null(&z);
mask = 1;
for (bits=0; bits < NUMBITS ; bits++)
{
    i = NUMWORD - bits/WORDSIZE;
    el = NUMWORD - (bits + 1)/WORDSIZE;

    r = k.e[i] & mask;
    t = z.e[i] & mask;

    r ^= t;

    if ( el == i )
    {
        r <<= 1;

        z.e[el] |= r;

        mask <<= 1;
    }

    else
    {
        mask = 1;
    }
}

```

```

        if (r) z.e[el] = 1;
    }
}

r = k.e[0] & UPRBIT;
t = z.e[0] & UPRBIT;
if ( r^t )
{
    null(&y[0]);
    null(&y[1]);
    return(2);
}

opt_mul(a, &z, &y[0]);
        null (&y[1]);

SUMLOOP(i) y[1].e[i] = y[0].e[i] ^ a->e[i];

return(0);
}

void fofx(x, curv, f)
FIELD2N *x, *f;
CURVE *curv;
{
    FIELD2N x2,x3;
    INDEX i;
    copy(x, &x2);
    rot_left(&x2);
    opt_mul(x, &x2, &x3);
    if (curv->form) opt_mul(&x2, &curv->a2, f);
        else null(f);

    SUMLOOP(i)

```

```

        f->e[i] ^= (x3.e[i] ^ curv->a6.e[i]);
    }
void esum (p1, p2, p3, curv)
POINT  *p1, *p2, *p3;
CURVE  *curv;
{
    INDEX  i;
    FIELD2N x1, y1, theta, onex, theta2;
    null(&x1);
    null(&y1);
    SUMLOOP(i)
    {
        x1.e[i] = p1->x.e[i] ^ p2->x.e[i];
        y1.e[i] = p1->y.e[i] ^ p2->y.e[i];
    }
    opt_inv( &x1, &onex);
    opt_mul( &onex, &y1, &theta);
    copy( &theta, &theta2);
    rot_left(&theta2);
    if (curv->form)
        SUMLOOP (i)
            p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ p1->x.e[i] ^ p2->x.e[i]
                ^ curv->a2.e[i];
    else
        SUMLOOP (i)
            p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ p1->x.e[i] ^ p2->x.e[i];
    SUMLOOP (i) x1.e[i] = p1->x.e[i] ^ p3->x.e[i];
    opt_mul( &x1, &theta, &theta2);

```

```

SUMLOOP (i) p3->y.e[i] = theta2.e[i] ^ p3->x.e[i] ^ p1->y.e[i];
}

void edbl (p1, p3, curv)

POINT *p1, *p3;

CURVE *curv;

{
    FIELD2N x1, y1, theta, theta2, t1;

    INDEX i;

    opt_inv( &p1->x, &x1);
    opt_mul( &x1, &p1->y, &y1);
    SUMLOOP (i) theta.e[i] = p1->x.e[i] ^ y1.e[i];
    copy( &theta, &theta2);
    rot_left(&theta2);
    if(curv->form)
        SUMLOOP (i) p3->x.e[i] = theta.e[i] ^ theta2.e[i] ^ curv->a2.e[i];
    else
        SUMLOOP (i) p3->x.e[i] = theta.e[i] ^ theta2.e[i];
    one( &y1);
    SUMLOOP (i) y1.e[i] ^= theta.e[i];
    opt_mul( &y1, &p3->x, &t1);
    copy( &p1->x, &x1);
    rot_left( &x1);
    SUMLOOP (i) p3->y.e[i] = x1.e[i] ^ t1.e[i];
}

void esub (p1, p2, p3, curv)

POINT *p1, *p2, *p3;

CURVE *curv;

{

```

```

POINT  negp;

INDEX  i;

copy ( &p2->x, &negp.x);

null (&negp.y);

SUMLOOP(i) negp.y.e[i] = p2->x.e[i] ^ p2->y.e[i];

esum (p1, &negp, p3, curv);
}

void copy_point (p1, p2)

POINT *p1, *p2;

{

    copy (&p1->x, &p2->x);

    copy (&p1->y, &p2->y);

}

void elptic_mul(k, p, r, curv)

FIELD2N      *k;

POINT  *p, *r;

CURVE *curv;

{

    char          blncd[NUMBITS+1];

    INDEX          bit_count, i;

    ELEMENT          notzero;

    FIELD2N          number;

    POINT          temp;

    copy( k, &number);

    notzero = 0;

    SUMLOOP (i) notzero |= number.e[i];

    if (!notzero)

    {

```

```

        null (&r->x);
        null (&r->y);
        return;
    }
    bit_count = 0;
    while (notzero)
    {
        if ( number.e[NUMWORD] & 1 )
        {
            blncd[bit_count] = 2 - (number.e[NUMWORD] & 3);
            if ( blncd[bit_count] < 0 )
            {
                for (i=NUMWORD; i>=0; i--)
                {
                    number.e[i]++;
                    if (number.e[i]) break;
                }
            }
        }
        else
            blncd[bit_count] = 0;
        number.e[NUMWORD] &= ~0 << 1;
        rot_right( &number);
        bit_count++;
        notzero = 0;
        SUMLOOP (i) notzero |= number.e[i];
    }
    bit_count--;

```

```

    copy_point(p,r);
    while (bit_count > 0)
    {
        edbl(r, &temp, curv);

        bit_count--;

        switch (blncd[bit_count])
        {
            case 1: esum (p, &temp, r, curv);
                    break;

            case -1: esub (&temp, p, r, curv);
                    break;

            case 0: copy_point (&temp, r);
        }
    }
}

void one (place)
FIELD2N *place;
{
    INDEX i;

    SUMLOOP(i) place->e[i] = -1L;

    place->e[0] &= UPRMASK;
}

/***** protocols1.c *****/

#include <stdio.h>

#include "field2n.h"

#include "elliptic.h"

extern opt_inv(), rot_left(), rot_right(), null(), opt_mul(), copy();

extern genlambda(), genlambda2();

```

```

int opt_quadratic();

void fofx();

void esum ();

void edbl ();

void esub ();

void copy_point ();

void elptic_mul();

void one(FIELD2N*);

void random_field();

void Mother();

void opt_embed();

void DH_gen_send_key();

void DH_key_share();

void send_elgamal();

void receive_elgamal();

void ECKGP();

void rand_curve ( );

void rand_point();

void print_field();

void print_point();

void print_curve();

void authen_secret();

unsigned long random_seed;

#include <string.h>

static short mother1[10];

static short mother2[10];

static short mStart=1;

#define m16Long 65536L          /* 2^16 */

```

```

#define m16Mask 0xFFFF      /* mask for lower 16 bits */

#define m15Mask 0x7FFF      /* mask for lower 15 bits */

#define m31Mask 0x7FFFFFFF  /* mask for 31 bits */

#define m32Double 4294967295.0 /* 2^32-1 */

/* Mother *****/

|   George Marsaglia's The mother of all random number generators producing uniformly
|   distributed pseudo random 32 bit values with period about 2^250.

*****/

void Mother(unsigned long *pSeed)
{
    unsigned long  number,
                  number1,
                  number2;

    short         n,
                  *p;

    unsigned short sNumber;

    if (mStart) {
        sNumber= *pSeed&m16Mask;
        number= *pSeed&m31Mask;
        p=mother1;
        for (n=18;n--;) {
            number=30903*sNumber+(number>>16);
            *p++=sNumber=number&m16Mask;
            if (n==9)
                p=mother2;
        }
        mother1[0]&=m15Mask;
        mother2[0]&=m15Mask;
    }
}

```

```

        mStart=0;
    }

    memmove(mother1+2,mother1+1,8*sizeof(short));
    memmove(mother2+2,mother2+1,8*sizeof(short));

    number1=mother1[0];
    number2=mother2[0];
    number1+=1941*mother1[2]+1860*mother1[3]+1812*mother1[4]+1776*mother1[5]+
        1492*mother1[6]+1215*mother1[7]+1066*mother1[8]+12013*mother1[9];
    number2+=1111*mother2[2]+2222*mother2[3]+3333*mother2[4]+4444*mother2[5]+
        5555*mother2[6]+6666*mother2[7]+7777*mother2[8]+9272*mother2[9];
    mother1[0]=number1/m16Long;
    mother2[0]=number2/m16Long;
    mother1[1]=m16Mask&number1;
    mother2[1]=m16Mask&number2;
    *pSeed=((long)mother1[1]<<16)+(long)mother2[1];
}

void random_field( value)
FIELD2N *value;
{
    INDEX i;

    SUMLOOP(i)
    {
        Mother( &random_seed);
        value->e[i] = random_seed;
    }

    value->e[0] &= UPRMASK;

```

```

}

void opt_embed( data, curv, incrmt, root, pnt)

FIELD2N      *data;

CURVE *curv;

INDEX  incrmt, root;

POINT *pnt;

{
    FIELD2N      f, y[2];

    INDEX      inc = incrmt;

    INDEX      i;

    if ( (inc < 0) || (inc > NUMWORD) ) inc = 0;

    copy( data, &pnt->x);

    fofx( &pnt->x, curv, &f);

    while (opt_quadratic( &pnt->x, &f, y))
    {
        pnt->x.e[inc]++;

        fofx( &pnt->x, curv, &f);

    }

    copy ( &y[root&1], &pnt->y);
}

void rand_curve ( curv)

CURVE *curv;

{
    curv->form = 0;

    random_field( &curv->a6);

    null( &curv->a2);

}

```

```

void rand_point( point, curve)

POINT *point;

CURVE *curve;

{
    FIELD2N      rf;

    random_field( &rf);

    opt_embed( &rf, curve, NUMWORD, rf.e[NUMWORD]&1, point);
}

void DH_gen_send_key( Base_point, E, my_private, My_public)

POINT *Base_point, *My_public;

CURVE *E;

FIELD2N *my_private;

{
    elptic_mul( my_private, Base_point, My_public, E);
}

void DH_key_share(Base_point, E, their_public, my_private, shared_secret)

POINT *Base_point, *their_public;

CURVE *E;

FIELD2N *my_private, *shared_secret;

{
    POINT  temp;

    elptic_mul( my_private, their_public, &temp, E);

    copy (&temp.x, shared_secret);
}

void send_elgamal(
    Base_point, Base_curve,

```

```

        Their_public, raw_data,
        Hidden_data, Random_point)

FIELD2N *raw_data;

POINT *Base_point, *Their_public, *Hidden_data, *Random_point;

CURVE *Base_curve;

{
    FIELD2N      random_value;

    POINT  hidden_point, raw_point;

    random_field (&random_value);

    elptic_mul (&random_value, Base_point, Random_point, Base_curve);

    opt_embed( raw_data, Base_curve, 0, 0, &raw_point);

    elptic_mul( &random_value, Their_public, &hidden_point, Base_curve);

    esum( &hidden_point, &raw_point, Hidden_data, Base_curve);

}

void receive_elgamal(
    Base_point, Base_curve,
    my_private, Hidden_data, Random_point,
    raw_data)

FIELD2N *my_private, *raw_data;

POINT *Base_point, *Hidden_data, *Random_point;

CURVE *Base_curve;

{
    POINT  hidden_point, raw_point;

    elptic_mul( my_private, Random_point, &hidden_point, Base_curve);

    esub( Hidden_data, &hidden_point, &raw_point, Base_curve);

    copy(&raw_point.x, raw_data);

}

CURVE Base_curve;

```

```

void authen_secret( d, Q, k, R, other_Q, other_R, W)

FIELD2N *d, *k;

POINT *Q, *R, *other_Q, *other_R, *W;

{
    POINT S, T, U;

    elptic_mul(&other_Q->x, other_Q, &S, &Base_curve);

    elptic_mul(&other_R->x, &S, &T, &Base_curve);

    esum(other_R, &T, &U, &Base_curve);

    elptic_mul(d, &U, &S, &Base_curve);

    elptic_mul(&Q->x, &S, &T, &Base_curve);

    elptic_mul(&R->x, &T, &S, &Base_curve);

    elptic_mul(k, &U, &T, &Base_curve);

    esum(&S, &T, W, &Base_curve);

}

void ECKGP( pnt, crv, k, R)

POINT *pnt, *R;

CURVE *crv;

FIELD2N *k;

{
    random_field( k);

    elptic_mul( k, pnt, R, crv);

}

void print_field( string, field)

char *string;

FIELD2N *field;

{
    INDEX i;

    printf("%s: ",string);

```

```

        SUMLOOP(i) printf("%8x ", field->e[i]);

        printf("\n");
    }

void print_point( string, point)

char *string;

POINT *point;

{

    printf("%s\n", string);

    print_field( "x", &point->x);

    print_field( "y", &point->y);

    printf("\n");

}

void print_curve( string, curv)

char *string;

CURVE *curv;

{

    //printf("NUMBITS= %d \n%s\n",NUMBITS, string);

    printf("%s %d\n\n ", string,NUMBITS);

    printf("form = %d\n ", curv->form);

    if (curv->form) print_field( "a2", &curv->a2);

    print_field( " a6", &curv->a6);

    printf("\n");

}

/***** Diffie-Hellman *****/

main()

{

    FIELD2N      private1, private2, key1, key2;

    CURVE koblitz, rnd_crv;

```

```

POINT Base, P1, P2;

INDEX i;

random_seed = 0xba10d1f5;

#ifdef TYPE2

    genlambda2();

#else

    genlambda();

#endif

printf("create random curve and point \n\n");
rand_curve(&rnd_crv);

    print_curve("random curve", &rnd_crv);
    rand_point(&Base, &rnd_crv);
    print_point("Base point", &Base);
    printf("\ncreate each sides private key\n\n");
    random_field(&private1);
    print_field("Side 1 secret:", &private1);
    random_field(&private2);
    print_field("Side 2 secret:", &private2);
    printf("\nGenerate each sides public key\n\n");
    DH_gen_send_key( &Base, &rnd_crv, &private1, &P1);
    print_point("Side 1 public key", &P1);
    DH_gen_send_key( &Base, &rnd_crv, &private2, &P2);
    print_point("Side 2 public key", &P2);
    printf("\nShow that each side gets the same shared secret\n\n");
    DH_key_share( &Base, &rnd_crv, &P2, &private1, &key1);
    print_field("key 1 is", &key1);
    DH_key_share( &Base, &rnd_crv, &P1, &private2, &key2);
    print_field("key 2 is", &key2);

```

```

        system("pause");
    }

/***** ElGamal *****/

main()
{
    FIELD2N      private1, private2, send_data, get_data;

    CURVE Public_Curve;

    POINT Base_Point, Their_public, Hidden_data, Random_point;

    INDEX i;

    random_seed = 0xffe1100;

#ifdef TYPE2
        genlambda2();
#else
        genlambda();
#endif

    printf("create Base curve and point \n\n");

    printf("NUMBITS= %d\n", NUMBITS);

    rand_curve(&Public_Curve);

    print_curve("Public curve", &Public_Curve);

    rand_point(&Base_Point, &Public_Curve);

    print_point("Base point", &Base_Point);

    printf("\ncreate side 2's private key\n\n");

    random_field(&private2);

    print_field("Side 2 secret:", &private2);

    printf("\nGenerate side 2's public key\n\n");

    elptic_mul(&private2, &Base_Point, &Their_public, &Public_Curve);

    print_point("Side 2 public key", &Their_public);

    printf("\nCreate message data\n\n");

```

```

    random_field( &send_data);

    printf("\nHide data on curve and send from side 1 to side 2\n\n");

    send_elgamal( &Base_Point, &Public_Curve, &Their_public,
    &send_data, &Hidden_data, &Random_point);

    print_point("Hidden data", &Hidden_data);
    print_point("Random point", &Random_point);

    printf("\nRecover transmitted message\n\n");

    receive_elgamal( &Base_Point, &Public_Curve, &private2,
    &Hidden_data, &Random_point, &get_data);

    print_field("sent data  ", &send_data);
    print_field("received data", &get_data);

    system("PAUSE");
}

```