Cryptanalysis of Elliptic Curve Scalar Multiplication Algorithms

BY

LEONG ZHEN HONG

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS) COMMUNICATIONS AND NETWORKING

Faculty of Information and Communication Technology

(Kampar Campus)

FEBRUARY 2025

COPYRIGHT STATEMENT

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Information Technology (Honours) Communications and Networking at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Dr. Norliana Binti Muslims, who has given me this bright opportunity to engage in a cryptanalysis. It is my first step to establish a career in the cryptographic field. A million thanks to supervisor and moderator. Their guidance, encouragement, and constructive feedback have been instrumental throughout the progress of this project. I am deeply grateful for their patience and willingness to share their knowledge, which has helped me develop a better understanding of both theoretical and practical aspects of cryptography. This project has not only expanded my technical skills but also strengthened my analytical thinking and research capabilities. Special thanks go to my family for their unwavering support, motivation, and belief in my potential, especially during challenging phases of this research. Their constant encouragement has been a major source of strength and perseverance. I would also like to acknowledge the support and collaboration of my peers and friends, who provided valuable discussions, feedback, and assistance throughout the journey. This project would not have reached its full potential without the collective contributions, encouragement, and inspiration I received from those around me.

ABSTRACT

This project explores scalar multiplication algorithms in Elliptic Curve Cryptography, focusing on the binary method and elliptic net method applied in Elliptic Curve Diffie-Hellman and Elliptic Curve Digital Signature Algorithm. Scalar multiplication is the most computationally intensive operation in Elliptic Curve Cryptography and directly impacts both cryptographic strength and performance. There is lack of standardized scalar multiplication algorithm or parameter set to ensure compatibility and interoperability in cryptographic implementations. This creates challenges in developing secure Elliptic Curve Cryptography systems and performing cryptanalysis for scalar multiplication algorithms. This research implemented both methods on secure Twisted Edwards curves (numsp384t1 and numsp512t1) using the affine coordinate system for clearer point representation. The binary method uses a double-and-add approach, which introduces conditional branches that increase execution variability, making it more vulnerable to timing-based side-channel attacks. In contrast, the elliptic net method structures point operations more uniformly, reducing observable patterns and improving leakage resistance despite its higher complexity. Simulated attack scenarios, including timing and power analysis, revealed that the elliptic net method maintained more consistent behavior and offered better protection against information leakage. Overall, the findings highlighted the performance of Elliptic Curve Cryptography Scalar Multiplications over side-channel attacks in the implementations.

Area of Study: Cryptography

Keywords: Binary, Diffie-Hellman, Digital Signature, Elliptic Net, Power Analysis, Timing

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

P	Point on the elliptic curve
k	Scalar, private key or multiplier in scalar multiplication

LIST OF ABBREVIATIONS

BM	Binary Method
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
EN	Elliptic Net
SCA	Side-channel attacks

Chapter 1
Introduction

Cryptography, the art and science of secure communication, is crucial in safeguarding sensitive data against unauthorized access and manipulation. Cryptographic techniques are essential for safe communication and data security in the digital age. They use sophisticated algorithms to encrypt data, making it unreadable by unauthorized parties. Common cryptographic techniques are symmetric-key encryption, asymmetric encryption and hash functions. The distinction between symmetric and asymmetric encryption is that asymmetric encryption uses both public and private keys.

Symmetric-key encryption ensures the confidentiality and integrity of data while it is in transit and at rest by using a single shared secret key for both encryption and decryption processes. Examples of symmetric-key encryption algorithms are AES (Advanced Encryption Standard) and DES (Data Encryption Standard) [1]. Public and private keys are used in asymmetric encryption, sometimes called public-key cryptography. Key exchange, digital signatures, and secure communication are all made possible by the popular asymmetric encryption method known as RSA (Rivest-Shamir-Adleman) [2], elliptic curve cryptography (ECC) [3] and elliptic curve digital signature algorithm (ECDSA) [4].

Developed in 1977, the elliptic curve discrete logarithm problem (ECDLP), which entails determining the value of d given Q and G, and the curve parameters, is the challenge that underpins the security of ECC. In contrast, the RSA method takes its foundation from the computing difficulty of factoring huge composite numbers into their prime factors. RSA generates keys, p and q randomly. To generate public key, modulus n and e while private key consists of modulus n and an exponent d, equation is $e * d = 1 \, mod \, n$. M equal to plaintext, C equal to ciphertext. Since factoring the product of two huge prime numbers is thought to be computationally impossible, this basic property of number theory serves as the bedrock for RSA's security [5]. To generate RSA keys, p and q are chosen randomly. To generate public key, modulus n and e while private key consists of modulus n and an exponent d, equation is $e * d = 1 \, mod \, n$. M equal to plaintext, C equal to ciphertext. For the encryption part, $C = Me \, mod \, n$, while decryption $M = Cd \, mod \, n$. The security of RSA relies on the

computational difficulty of factoring the modulus nn into its constituent prime factors p and q for a given n. ECC and ECDSA use the mathematical characteristics of elliptic curves to produce strong cryptographic solutions with smaller key sizes [6]. ECC technique was first proposed by Neal Koblitz [7] and Victor Miller [8] in 1985. ECC is based on the difficulty of solving mathematical problems related to elliptic curves, such as the elliptic curve discrete logarithm problem [6]. For the encryption part, $C = Me \bmod n$, while decryption $M = Cd \bmod n$. The security level of RSA relies on computational difficulty, factoring the modulus n into its constituent prime factors p and q for a given n. In ECC, the private key d is a random generate integer, and the public key obtained by multiplying a base point G by the private key, $Q = dG$. ECC is refined and improved based on the Diffie-Hellman key exchange protocol eventually finding its way into numerous security standards against side-channel attacks (SCAs) and applications [9]. The base point G and the parameters of the elliptic curve are public knowledge [6]. The security of ECC relies on the difficulty of the elliptic curve discrete logarithm problem (ECDLP), which involves finding the value of d given Q, G, and the curve parameters.

The ECDSA is a digital signature scheme that uses mathematical properties of elliptic curves to enhance secure and efficient cryptographic signatures. ECDSA was first proposed in 1992 by Scott Vanstone in response to NIST's request for public comments on their first proposal for DSS [4]. Like ECC, ECDSA uses random key generation to produce a private key d and public key that is calculated as $Q = dG$. The private key is used to sign messages, and the public key verifies the signatures. ECC and ECDSA are more efficient in terms of computational resources and bandwidth utilization than RSA, which usually requires bigger key sizes for equal security levels [9]. The security of ECC depends on the size of the elliptic curve and the size of the underlying finite field. Typically, ECC with a key size of 256 bits offers equivalent security to RSA with a key size of 3072 bits [10]. The private key is used to sign messages, and the public key verifies the signatures. Since ECC and ECDSA can provide strong security with shorter key lengths, they are better suited for applications where memory and processing power are limited. The computational effectiveness of scalar multiplication algorithms, which are essential to ECC operations, is also evaluated through cryptographic analysis. This makes them ideal for contexts where resources are scarce, including mobile devices and Internet of Things devices [11]. ECC requires cryptographic analysis to validate the

security of ECC-based protocols and algorithms, as well as to determine how strong elliptic curve parameters are and how resistant they are to different types of assaults [12]. Besides, scalar multiplication algorithm's efficiency, defense against SCAs, and compliance with security best practices are all assessed through cryptographic analysis [12]. The computational effectiveness of scalar multiplication algorithms, which are essential to ECC operations, is also evaluated through cryptographic analysis. The time complexity, memory needs, and performance characteristics of the method across different platforms are frequently assessed as part of this analysis process [13]. The vulnerability of the scalar multiplication algorithm to SCAs, which take advantage of implementation flaws, is another factor considered in cryptographic analysis. Ensuring the security of scalar multiplication in ECC requires strategies for thwarting these attacks, such as constant time algorithms and secure hardware implementations [14]. Through a comprehensive exploration of cryptanalytic methodologies, this research aims to cryptanalysis the existing scalar multiplication algorithms and propose enhancements to against potential threats. By uncovering vulnerabilities and developing robust countermeasures, this work attempts to contribute to the ongoing efforts to bolster the security of ECC and address the evolving challenges posed by cyber threats. The importance of cryptanalysis needs to be classified as the improvement needed by user and also algorithm, including theoretical underpinnings, vulnerabilities, and potential avenues for improvement. Additionally, we will explore various cryptanalytic techniques and their application in evaluating algorithmic security, laying the groundwork for a comprehensive understanding of elliptic curve cryptography in contemporary digital environments. [15] and [16] demonstrate the importance of cryptanalysis in uncovering vulnerabilities and advancing the state-of-the-art in cryptographic research, underscoring its critical role in ensuring the trustworthiness of modern cryptographic systems. This research presents an extensive study of the scalar multiplication algorithms implementation on workstations of the NUMS elliptic curves over prime field [17].

1.1 Problem Statement and Motivation

In ECC, scalar multiplication algorithm via binary method (BM) involves complex operations, which lead to an intricate implementation challenge. Developing efficient and protective implementations of the BM that minimizes implementation complexity

to maintain resistance against SCAs, and implementation level vulnerabilities poses a significant challenge. Furthermore, scalar multiplication algorithms via elliptic net (EN) existing security vulnerabilities, invalid curve attacks or invalid point attacks if implemented incorrectly. In addition, there is a lack of standardized scalar multiplication algorithms and parameters that can justify the interoperability and compatibility in cryptography scalar multiplication implementations. Identifying potential security vulnerabilities in EN method implementations and developing robust countermeasures to mitigate these risks while maintaining performance and efficiency is essential to enhance the security level of ECDH and ECDSA. Thus, there is a need to perform cryptanalysis for scalar multiplication algorithms via the BM and EN method by evaluating vulnerabilities, goals, standards, improvements and other factors against SCAs. In most cases, if cryptanalysis is successful at all, an attacker cannot deduce information about the plaintext [18]. Cryptanalysis based on SCAs helps uncover vulnerabilities in ECC implementations by exploiting information leakage. The central idea of side-channel analysis is to compare some secret data-dependent predictions of the physical leakages and the actual leakage to identify the data most likely to have been processed [19]. The side-channel analysis considers attacks that do not aim at the algorithms' weaknesses but their implementations [20]. ECC leverages the double-and-add method for scalar multiplication, a key operation in generating public key and private key to perform computations. This method computes scalar multiples of points on the elliptic curve, ensuring integrity and confidentiality. Double-and-add implementation shows that the ECC scalar point multiplication algorithm succeeds in preventing SCAs, Simple Analysis Attacks, and Differential Power Attacks [21], [20].

The motivation behind this research relies on the passion for cryptography and network security. Research was sparked by studying ECC scalar multiplication and its robust defense against SCAs. Thus, the need for cryptanalysis increases to prove the security level of ECC scalar multiplication against SCAs.

Cryptanalysis is a process of analyzing ECC scalar multiplication to understand the hidden aspects of cryptographic operation. This field is an important aspect of cryptography, the broader science of securing communication and data using codes and ciphers [22].

Due to the difficulty in finding evidence to prove security levels against SCAs, I look forward to expanding my knowledge in cryptographic security with a focus on cryptanalysis. I chose the cryptanalysis method to identify the weaknesses and strengths of ECC scalar multiplication because it was inspired by [23]. The proposed method by [23] was faster than the BM in an affine coordinate system, the relative efficiency can be compared with some experimental results.

1.2 Objectives

This project aims to perform cryptanalysis on scalar multiplication algorithms implemented via binary and EN method. Thus, the research objectives are stated as follows:

a) To identify potential vulnerabilities in the scalar multiplication algorithms via binary and EN methods.

To achieve the current objective of identifying potential vulnerabilities, the project focused on analyzing BM and the EN method. These methods will be examined in the context of SCAs, which exploit sensitive information or secret key data to attackers. This objective must go through a literature review. Using search keywords such as "Potential weaknesses of ECC", "Side-channel analysis on scalar multiplication, "Review of latest side-channel attacks" and relevant references will be analyzed. Studies on power analysis attacks, timing attacks, fault attacks, electromagnetic analysis attacks on ECC will be reviewed to understand how existing methods can be exploited.

b) To implement the following double-and-add algorithm in ECDH and ECDSA schemes:

i) Binary method

ii) Elliptic net method

Implement the double-and-add procedure in ECDH to compute the shared secret, and in ECDSA to generate signatures after identifying vulnerabilities.This involves iterating through each bit of the private key, performing point doubling always and point addition when needed. In ECDH, this method is used to compute the public key $Q = dG$, efficiently using the numsp384t1 and numsp512t1 curves. After exchanging public keys, the shared secret $S = dQ'$ is derived. The implementation will ensure that

execution traces are collected, allowing for analysis of timing attacks, power analysis attacks, and cache-based SCAs to determine security weaknesses. Simulated attacks will assess whether the private key can be inferred from power or timing variations.

For the EN method, apply structured point sequences to optimize scalar multiplication in both ECDH and ECDSA. This method improves efficiency by structuring operations to reduce computational overhead and potentially minimize side-channel leakage. In ECDH, it is used to compute $Q = dG$, and the shared secret is derived similarly to the BM but with optimized steps.

For ECDSA, the BM is applied in signature generation, where the ephemeral key *k* is processed using double-and-add. The signature values (*r*, *s*) recomputed, ensuring correctness while assessing vulnerabilities in the scalar multiplication step. Since any leakage from *kG*, compromises the private key, power analysis and fault injection simulations will be conducted on implementations using numsp384t1 and numsp512t1. The impact of cache timing variations and template attacks on the security of ECDSA using the BM will also be examined.

For ECDSA, the EN method optimizes scalar multiplication to reduce observable computation patterns that could be exploited in attacks. By structuring point sequences differently, this approach aims to minimize predictable power consumption. The implementation in Python will include tests for timing analysis, electromagnetic emissions, and fault-induced errors to determine security levels. Simulated attacks will be used to compare the resilience of EN against traditional BMs, ensuring a comprehensive evaluation of SCAs resistance.

c) To evaluate the proposed algorithms based on SCAs.

The results are recorded, including execution time, memory usage, and computational overhead. Side-channel assessments will focus on timing attacks and power analysis attacks. Timing attacks are applied to ECDH, while power analysis attacks are used for ECDSA. Timing attacks are chosen for ECDH because the execution time of scalar multiplication varies based on the private key bits. Since the double-and-add algorithm exhibits different computational patterns for ‘0’ and ‘1’ bits, an attacker can analyze execution times to infer key bits. Power analysis attacks are applied to ECDSA because the scalar multiplication step during signature generation uses an ephemeral key *k*, which can be targeted to reveal sensitive information. To fulfil this objective, this project implemented timing attacks on ECDH by measuring execution time variations

in scalar multiplication conducting power analysis attacks on ECDSA by simulating power consumption traces during signature generation and applying correlation techniques to infer the ephemeral key.

1.3 Project Scope and Direction

The scope of the project encompasses a comprehensive investigation into security of ECSM via prime field, with a particular focus on its implementation via EN and binary methods architectures. Furthermore, the project will assess the susceptibility of ECSM by implementing power analysis and timing attacks.

ECC scalar multiplications are formed with different schemes, such as Elliptic Curve Diffie Hellman (ECDH), Elliptic Curve Digital Signature Algorithm (ECDSA), EdDSA (Edwards-curve Digital Signature Algorithm) [24], ECMQV (Elliptic Curve Menezes-Qu-Vanstone) [25] and ECIES (Elliptic Curve Integrated Encryption Scheme) [26].

The project focuses on analyzing the security of Twisted Edwards curves, specifically numsp384t1 and numsp512t1, in scalar multiplication algorithms. These curves offer efficient arithmetic and strong security properties, making them suitable for cryptographic applications. The numsp384t1 curve operates over a 384-bit prime field, while numsp512t1 uses a 512-bit prime field, both defined with specific parameters for the curve equation and generator point. By implementing double-and-add algorithms on these curves, the study evaluates their resistance to timing and power analysis attacks, ensuring robust cryptographic performance.

ECDSA was first proposed in 1992 by Scott Vanstone [27] in response to NIST's request for public comments on its first proposal for a Digital Signature Standard. It was accepted in 1998 as an International Standards Organization standard and in 2000 as an IEEE (Institute of Electrical and Electronics Engineers) standard [4].

The ECDH distinct from the general Diffie Hellman (DH) in the way that it is based on the elliptic curve discrete logarithm problem (ECDLP) instead of the discrete logarithm problem (DLP) [28]. ECDH is an anonymous key agreement protocol which allows two parties, A and B, to establish a shared secret key over an insecure channel, where each of the parties has an elliptic curve public-private key pair [29].

The BM is a widely used approach for scalar multiplication in elliptic curve cryptography, particularly in ECDH and ECDSA. It follows the double-and-add

algorithm, where the scalar is processed bit by bit. For each bit, point doubling is always performed, and point addition is executed only when the bit is '1'. This method is simple and efficient but introduces side-channel vulnerabilities, especially timing attacks, due to its varying execution flow. In this project, the numsp384t1 and numsp512t1 curves are implemented using the BM to assess timing and power analysis attacks. Since different bit patterns affect execution time and power consumption, an attacker may exploit these variations to infer private key bits. This study evaluates its security impact and explores potential countermeasures.

The EN method is an alternative scalar multiplication technique that optimizes efficiency and security. Instead of processing bits individually like the BM, it structures point sequences to ensure a more uniform computation pattern. This approach helps mitigate timing and power analysis attacks by reducing observable variations. In this project, the numsp384t1 and numsp512t1 curves are implemented using this method to compare its resistance against SCAs. By analyzing execution traces, this study determines whether EN offers improved security over BM.

In this project, ECC scalar multiplication is only available for affine coordinates over Homogeneous, L´opez-Dahab, Jacobean and other coordinates. In affine coordinates, ECC operations are usually done by using the affine coordinate [x, y] [30].

1.4 Contributions

The main contributions of this study are stated as follows:

1. Developing countermeasures and mitigations can enhance the resistance of scalar multiplication via BM implementations against SCAs, such as timing attacks and simple power analysis attacks can prevent leakage of sensitive information.
2. Providing methodologies for selecting appropriate curve parameters, such as the choice of elliptic curve parameters and key sizes can optimize the security level of scalar multiplication algorithms.
3. Exploring the security level of binary or elliptic methods implementations against SCAs, which are timing attack and simple power analysis attack. Timing attack measures the time taken to perform cryptographic operations and uses this information to exploit cryptographic keys [31]. Power analysis attack

breach energy cost to perform cryptographic operation, this is done to find sensitive information [32].

4. Evaluating the performance and security of ECDH and ECDSA. The computed outputs are compared with standard cryptographic libraries to ensure correctness, while execution time, memory usage, and computational overhead. These aspects provide insights into inefficiencies, performance bottlenecks, and potential security weaknesses in scalar multiplication implementations, leading to improvements and enhanced robustness of cryptographic systems. Additionally, this project examines SCAs vulnerabilities, including timing attacks on ECDH and power analysis attacks on ECDSA. These attacks are tested on numsp384t1 and numsp512t1 curves to evaluate their resistance, ensuring that the implementations remain secure and efficient for real-world applications.

1.5 Report Organization

The details of this research are shown in the following chapters. In Chapter 2, some related backgrounds are reviewed as literature reviews, and several tables are shown for a better understanding. Furthermore, chapter 3 outlines the project's methodology, including the system model, algorithm design, and timeline. Chapter 4 explains the implementation details of scalar multiplication using both the BM and EN methods within the ECDH and ECDSA schemes. Chapter 5 presents experimental setups and simulation results, along with discussions of challenges encountered during development. Chapter 6 evaluates performance metrics, discusses attack results, and reviews how each objective was met. Finally, Chapter 7 concludes the report with key findings and recommendations for future work.

Chapter 2

Literature Review

2.1 Previous Works on Scalar Multiplication Algorithms Via Binary Method

A modified double-and-add algorithm based on the Karatsuba-Ofman algorithm [33], [34], generating new Vi over prime field by setting $S_i = V_{i+1}^2$, $S_i = V_{i+1}^2, P_i = ((V_i + V_{i+2})^2 - S_i - S_{i+2})/2$, and $R_i = S_i P_i$ for $1 \leq i \leq 4$, in which the outcomes are $V_0 = (S_0 - S_1)(P_0 + P_1) - R_0 + R_1, V_1 = (S_0 - S_2)(P_0 + P_2) - R_0 + R_2)\alpha, V_2 = (S_1 - S_2)(P_1 + P_2) - R_1 + R_2, V_3((S_1 - S_3)(P_1 + P_3) - R_1 + R_3\alpha, V_2 = V_4 = (S_2 - S_3)(P_2 + P_3) - R_2 + R_3)$, $V_5((S_2 - S_4)(P_2 + P_4) - R_2 + R_4)\alpha$ and $V_6 = (S_3 - S_4)(P_3 + P_4) - R_3 + R_4$ [35]. Each Vi cost 1M so 7M obtained in double block. The new term is $V_0 = ((\mathrm{S_0} - \mathrm{S_2})(\mathrm{P_0} + \mathrm{P_2}) - \mathrm{R_0} + \mathrm{R_2})\tilde{\alpha}$, $\mathrm{V_2} = ((\mathrm{S_1} - \mathrm{S_3})(\mathrm{P_1} + \mathrm{P_3}) - \mathrm{R_1} + \mathrm{R_3})\tilde{\alpha}, \mathrm{V_3}(\mathrm{S_2} - \mathrm{S_3})(\mathrm{P_2} + \mathrm{P_3}) - \mathrm{R_2} + \mathrm{R_3}, \mathrm{V_4} = ((\mathrm{S_2} - \mathrm{S_4})(\mathrm{P_2} + \mathrm{P_4}) - R_2 + \mathrm{R_4})\tilde{\alpha}, \mathrm{V_5}(\mathrm{S_3} - \mathrm{S_4})(\mathrm{P_3} + \mathrm{P_4}) - \mathrm{R_3} + \mathrm{R_4})$ and $\mathrm{V_6} = (\mathrm{t_1}\varepsilon - \beta \mathrm{t_2})/V_2$ [36]. Each value of V_0 until V_6 requires 1M, but V_6 needs 2M so a total of 8M obtained for double ad block.

A new double-and-add method proposed by [37]. By utilizing EN block, [38] used temporary variables S_i and P_i as an array of six elements, cost 6M + 6S and utilized the S_i and P_i by adding two groups of intermediate variables A_i, B_i, C_i, D_i and E_i for double-and-add function [38]. The number of multiplications using the repeated multiplication W_{k-2} and W_k, $a = W_{k-2}W_k, b = W_{k-1}W_{k+1}, c = W_k W_{k+2}$ as well as $e = W_{k-1}^2, f = W_k^2, g = W_{k+1}^2$ [37]. This equation costs 2M for each variable and [37] costs 4M for each. EN Scalar Multiplication can be designed based on double-and-add with block centred at one [39]. Table 2 shows the cost for both methods.

Table 2.1 Comparison of old and new double-and-add methods

Method	Temporary variable	double	double-add	Total cost	
				double	double-add
[37]	12M+10S	4M	4M	16M+10S	16M+10S
[39]	10M+6S	2M	2M	12M+6S	12M+6S

2.2 Previous works on Scalar Multiplication algorithms via Elliptic Net

Prior studies have explored various methods to enhance the efficiency of scalar multiplication operations. EN is an architecture used to organize elliptic curve points, enabling faster scalar multiplication through techniques such as point doubling, point addition, the Montgomery ladder algorithm, and differential addition-subtraction chains. While the first EN Scalar Multiplication over binary fields remains unknown, its construction over a prime field has been documented [40]. The approach proposed in [11] introduces a robust EN Scalar Multiplication algorithm that resists SCAs, which is crucial for maintaining the confidentiality and integrity of sensitive information in engineering systems utilizing the double-and-add algorithm.

2.3 Previous work on cryptanalysis method

Previous studies focused on analysis and prevention from SCAs, this section describes each of the protocols for future works. ECC methods guarantee level of security but there is an easily exploitable vulnerability. Hence, an additional level of protection is crucial to guarantee total security against SCAs. Most of the multi-factor authentication and key exchange protocols, rely on ECC for security protection [41]. To meet the requirement for enhanced-security near-ideal models, ECC being a small key size with the capability to thwart SCAs must now include countermeasures against assaults [42].

2.3.1 Previous work on side-channel attack

SCAs are a class of security threats that exploit unintended information leakage from physical implementations of cryptographic systems. SCAs focus on analyzing observable side effects of the implementation, such as power analysis attack, timing attack, or electromagnetic analysis attack. Timing Attacks, it was based on exploiting the non-constant execution time using different input values to reveal the secret information [43]. Power Analysis Attacks exploit variations in power consumption during cryptographic operations, attackers find out power consumption and attack to get data contained [12]. Electromagnetic Analysis Attacks is a form of attack that exploits the electromagnetic emanations from an electronic device as a form of information leakage [44]. These attacks pose threats to cryptographic systems, undermining their security and confidentiality.

2.3.2 Key Size comparison between RSA and ECC

The relationship between ECC and RSA are complementary, two widely used cryptographic algorithms that provide security for data through asymmetric encryption.

Table 2.2 RSA vs ECC [45]

Symmetric Key Size (bits)	RSA Size (bits)	ECC Size (bits)
80	1,024	160
112	2,048	224
128	3,072	256
192	7,680	384
256	15,360	521

2.4 Summary of previous work on algorithm

SCAs represent a class of attacks that exploit physical leakage from cryptographic devices, rather than directly attacking the cryptographic algorithms themselves. These attacks can include timing analysis, power consumption monitoring, electromagnetic emissions, and others states as below:

Table 2.3 Summary of previous work on ECC scalar multiplication algorithms

Author	Scalar Multiplication Algorithms	Coordinate	Side-channel Attacks
[12]	Double-and-add	Mix (Affine and Jacobian)	Electromagnetic Attack
[46]	Adding and Doubling operation	Jacobian	Timing Attacks
[47]	Adding and doubling points	Jacobian	Simple Power Analysis
[48]	Double-and-add	Affine	Simple Power Analysis
[49]	Miller's algorithm	Mix (Affine and Jacobian)	Correlation Power Analysis
[50]	Modular Inversion	Affine	Power Analysis Attack

[51]	Montgomery ladder	Affine	Montgomery Ladder Fault Attacks
This work	Scalar Multiplication via BM and EN	Affine	Timing attack and simple power analysis attack

Based on this literature review, timing attack and simple power analysis attack have the most frequent in SCAs realm. Within this scope, a paper conducted a mixed-methods online survey with 44 developers of 27 popular cryptographic libraries to understand how real-world cryptographic library developers think about timing attacks. In result, all 44 participants are aware of timing attacks [52]. Simple power analysis attack, it traced power consumption for cryptographic operation and possible to determine path of instructions execution trace [53]. The two SCAs mentioned are possible to breach ECC scalar multiplication vulnerabilities, so this project aims to implement timing attack and simple power analysis attack.

2.5 Cryptographic schemes

2.5.1 Diffie-Hellman key exchange:

Diffie-Hellman is for key exchange between users, ensuring connection with CIA triad guidelines, confidentiality, integrity and availability. Diffie-Hellman algorithm primarily generates a shared secret key across public networks, known as a key exchange. The process starts with users, such as Alice and Bob. Both generate a secret key and keep for themselves. Next, users generate a public key using Diffie-Hellman algorithm. The public keys are essential, send the public key to each other to complete the connection. Last phase, Alice combines own secret key with Bob's public key into a number, k. While Bob compute k using own secret key and Alice's public. Both k have the same value, the key exchange process is completed as stated as below:

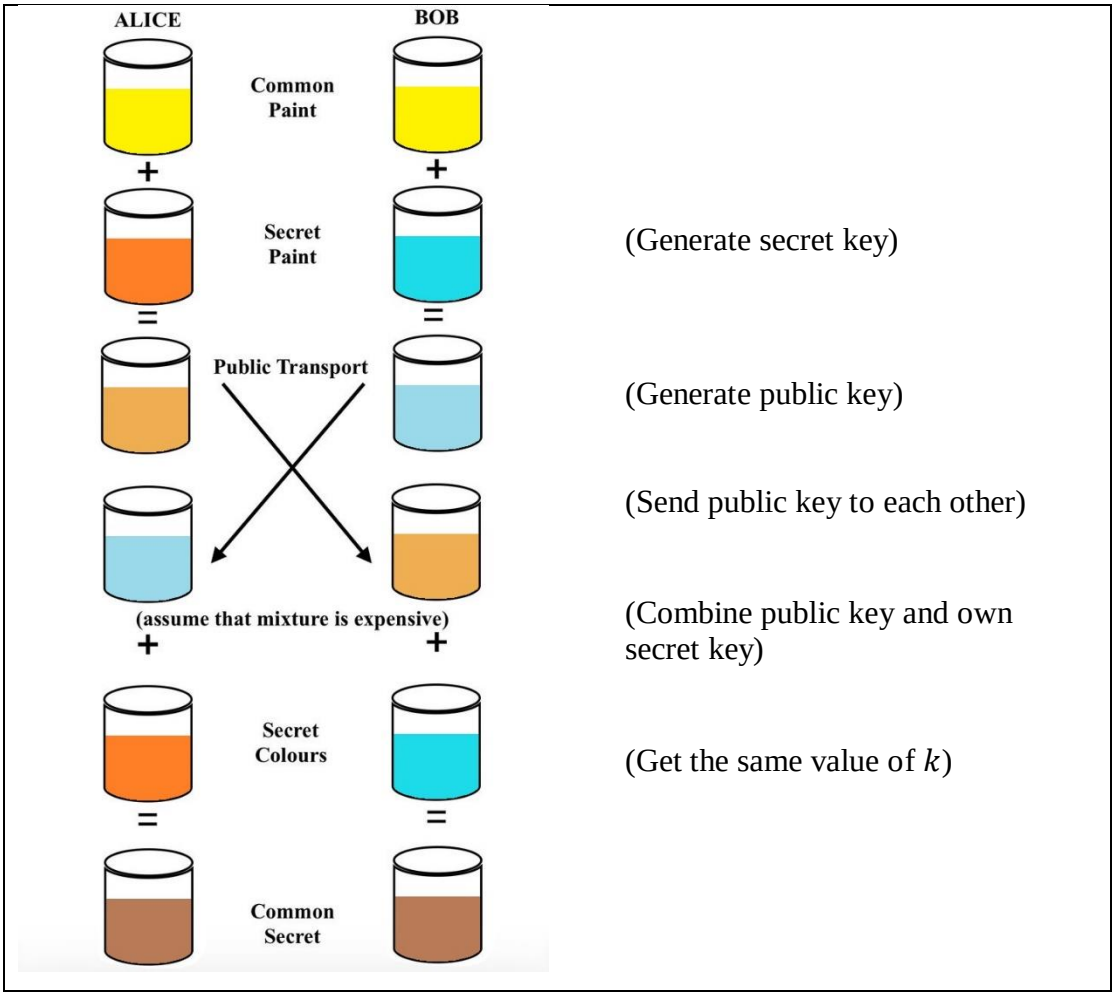


Figure 2.1 Diffie-Hellman key exchange [54]

2.5.2 Digital signatures and certifications:

The digital signatures consist of two phases. For example, Alice sends a document to Bob, Alice generates two keys, private key remains private. But the public key, Alice needs to share with Bob to verify the document and signature. The first phase, signing, the content of document runs through the hash algorithm, and transformed to a digest, the content inside consists of different numbers, symbols or other letters. Digests encrypt with private key, the signature phase completed. The digest sends to Bob, and Bob starts the verification phase. Bob has two options to decrypt the digest, decrypt with Alice's public key or digest run through the hash algorithm. Both options get the same outcome, as the document same as Alice's, the process states below:

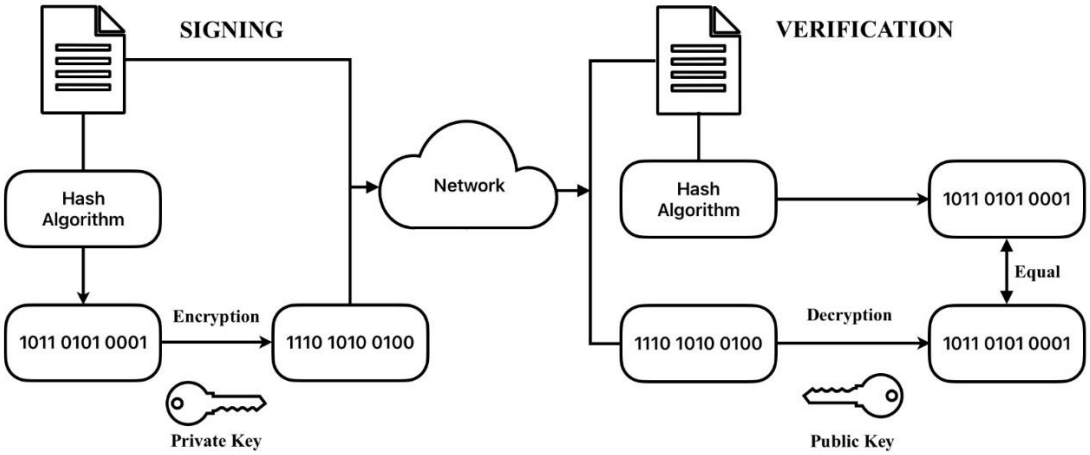


Figure 2.2 Digital signatures and certifications [55]

2.6 Previous work related to ECDH and ECDSA

Schemes from the realm of ECC are ECDH and ECDSA. ECDH, renovated to securely exchange secret keys between parties. While ECDSA did the same, it was celebrated for its prowess in verifying the authenticity of messages and transactions. But, SCAs can exploit through schemes in ECC or RSA, so the weakness of each scheme should be discussed for improvement, references collected shown below:

Table 2.4 Summary of previous work on ECDH and ECDSA

Author	**Algorithm**	**Attacks**	**Outcome**
[28]	ECDH	Man-in-the-middle Attack	Secure
[56]	ECDH	Differential-bit Horizontal Clustering Attack	Fail to Secure
[57]	ECDH	Differential Power Attack	Secure
[58]	ECDH	Timing Attack	Secure
[59]	ECDH & ECDSA	Microarchitectural Attack	Fail to secure
[60]	ECDSA	Fault Attack	Secure
[61]	ECDSA	Simple Power Analysis Attack	Secure
[62]	ECDSA	Timing Attack	Fail to secure
[63]	ECDSA	Template Attack	Fail to secure

Chapter 3

System Methodology

3.1 System Design Equation

Sections 3.1.1 and 3.1.2 show the BM algorithm [68] and EN algorithm [23], respectively.

3.1.1 Binary Method Algorithm

Table 3.1 Binary Method Algorithm [68]

Input: An Affine point P ∈ E($\mathbb{F}_p$) and $n = (n_{i-1}, \dots, n_0)_2$.
Output: $nP \in E(\mathbb{F}_p)$
Steps:
1. For i from $l - 2$ down to 0 do
2. $Q \leftarrow 2Q$
3. If ni=1 then
3.1 $Q \leftarrow P+Q$
4. Return Q

Table 3.1 shows the BM method, which is an effective way of performing scalar multiplication on elliptic curves. Given a point P ∈ E($\mathbb{F}_p$) and a scalar *n* represented in binary, the algorithm initializes a result point *Q* and processes each bit of *n* from the second most significant to the least significant. In each iteration, it doubles *Q*, and if the current bit is 1, it adds the original point *P* to *Q*. This method reduces the total number of required point additions, making it significantly more efficient than repeated addition, especially for large scalars.

3.1.2 Elliptic Net Algorithm

Table 3.2 Elliptic Net Method Algorithm [23]

Input: Integer $n = (n_{l-1},\ n_{l-2}, \dots, n_0)_2$ with $n_{l-1} = 1$. $P \epsilon E(\mathbb{F}p)$, $a = W_2$, $b = W_3$ and $c = W_4$ of the EN associated to P and $I = y^{-1}$.
Output: The EN values $W\lambda$ where $n-2 \leq \lambda \leq n+2$ associated to point P.
Steps:
1. $V \leftarrow [-a, -1, 0, 1, a, b, c, a^3, c - b^3]$
2. For i from $l - 1$ down to 0 do

3. If n_i = 0 then

   V ← $double(V)$

4. Else

   V ←$doubleadd(V)$

5. $A = V_3^{-1}$; $B = A^2$; $C=AB$

6. $E = V_2^2$; $F = V_4^2$; $G = V_2V_4$

7. $H = BG$; $J=EV_5$; $K = FV_1$

8. $x_n= x_1 - GB$

9. $y_n = (J-K)\ IC$

This algorithm computes EN values W_λ for indices near a given scalar *n*, using a recursive approach based on the binary representation of *n*. Starting with initial values derived from the EN associated with point *P* on the curve $\mathrm{E}(\mathbb{F}_p)$, the algorithm initializes a vector *V* with specific EN terms and then iteratively updates it using either the $double$ or $doubleadd$ operation depending on each bit of *n*, scanned from most to least significant. After processing all bits, intermediate variables *A*, *B*, *C*, *E*, *F*, *G*, *H*, *J*, *K* are calculated to derive the final coordinates x_n and y_n, which represent the scalar multiplication result *nP* in terms of the EN.

3.2 Project timeline

Figure 3.1 shows a project timeline to distribute tasks, it was required a project timeline to distribute tasks for each week, the duration is 14 weeks. Week 1 to 2 are required to complete the planning process, such as develop project charter and collect references. Start from week 3, focus on implementation to obtain outcome of algorithms and SCAs. Project report required to complete within 8 weeks and finalize before week 12 end. Week 13 and 14, as the report submitted, focus on presentation, prepare slides and script to perform and score well.

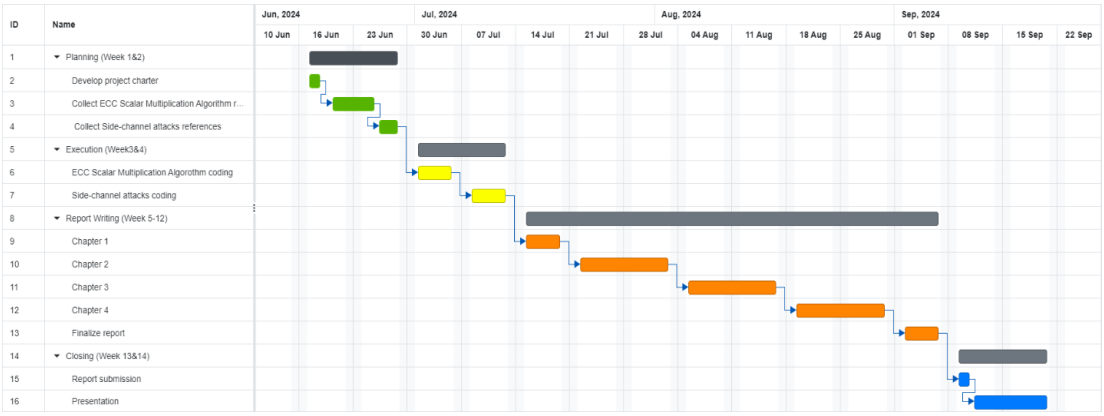


Figure 3.1 Timeline of FYP 1

Figure 3.2, this project timeline guide to distribute tasks for each week, the duration same as FYP 1. Week 1 planned to complete the planning process, such as develop project charter and solve issue from FYP 1. Start from week 2, focus on execution of algorithms and SCAs. Project report required to complete within 8 weeks and finalize before week 13. Week 13 planned to rehearsal before report submission and following task is to submit the latest version of report. The final task is to perform a presentation to supervisor and moderator.

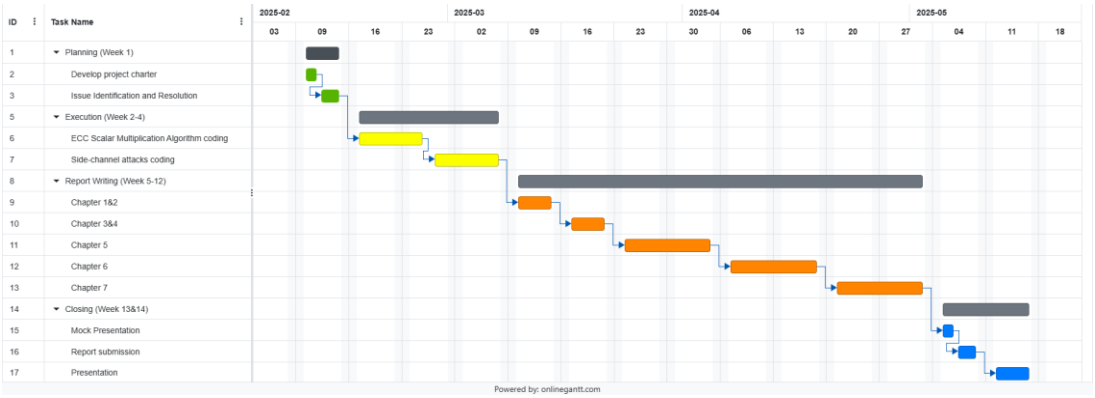


Figure 3.2 Timeline of FYP 2

Chapter 4

System Design

4.1 System Block Diagram

4.1.1 ECDH using Binary Method Flowchart

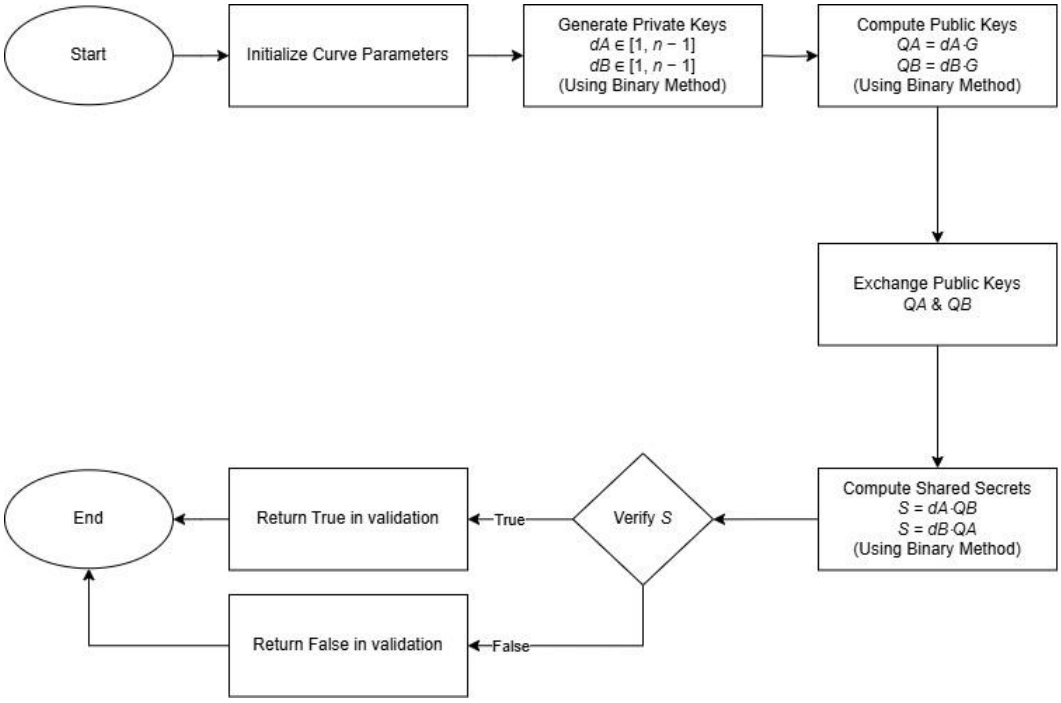


Figure 4.1 ECDH Key Exchange Process Flowchart Using Binary Method

Figure 4.1 shows the process of ECDH key exchange using the binary. The process begins with the initialization of elliptic curve parameters, followed by the generation of private keys for both parties within the range $[1, n-1]$. Using the BM, each party computes their corresponding public key by multiplying the private key with the base point *G*. The public keys are then exchanged, allowing each party to compute a shared secret by multiplying their private key with the other party's public key. Due to the mathematical properties of elliptic curves, both parties derive the same shared secret. A verification step confirms whether both shared secrets match. If true, the process ends successfully. Otherwise, it returns a validation failure result.

4.1.2 ECDH using Elliptic Net Method

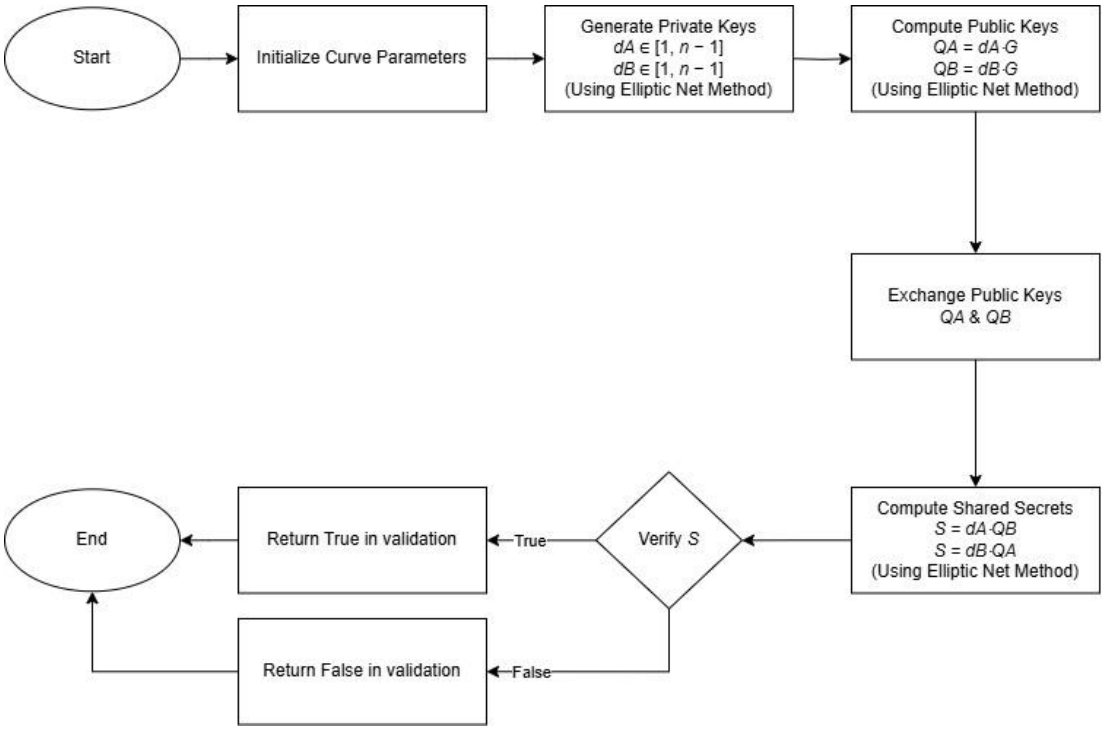


Figure 4.2 ECDH Key Exchange Process Flowchart Using Elliptic Net Method

Figure 4.2 shows the process of ECDH key exchange using the EN method. The process begins with the initialization of elliptic curve parameters, followed by the generation of private keys for both parties within the range $[1, n-1]$. Each party then computes their respective public key by performing scalar multiplication of their private key with the base point *G* using the EN method. After exchanging public keys, both parties compute a shared secret by multiplying their private key with the other party's public key, again using the EN method. A verification step ensures the validity of the shared secret. If the validation is true, the process is successful and ends. Otherwise, it indicates a failure.

4.1.3 ECDSA using Binary Method

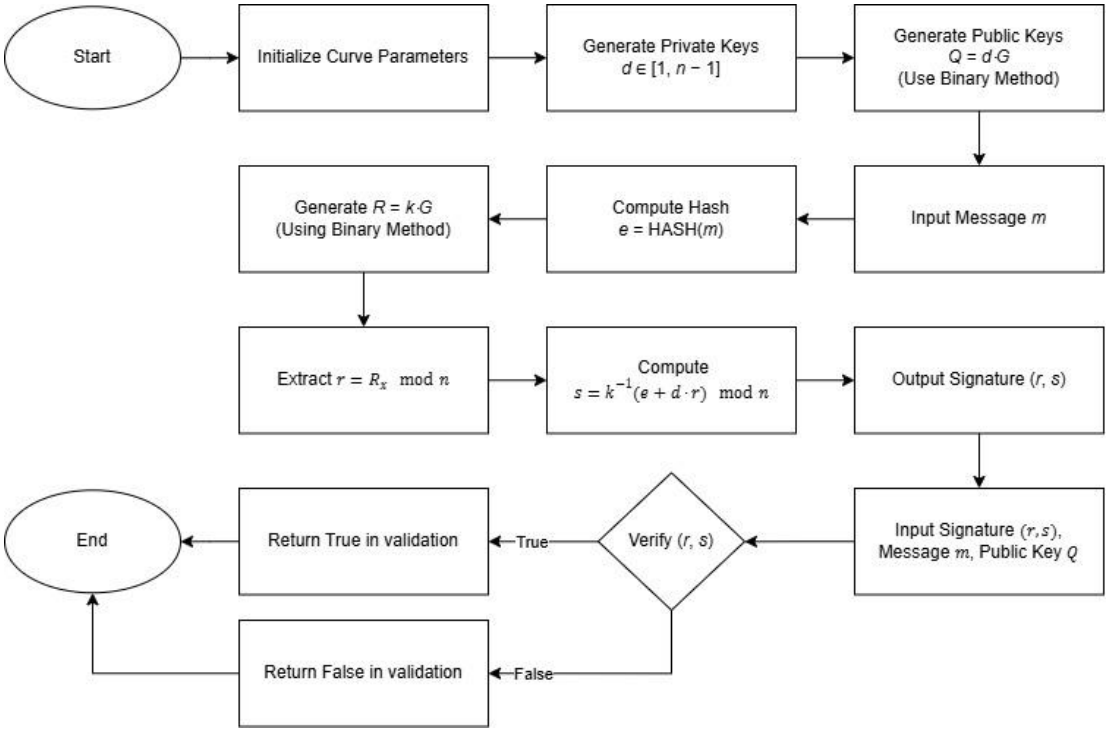


Figure 4.3 ECDSA Key Exchange Process Flowchart Using Binary Method

Figure 4.3 illustrates the process of ECDSA signature generation and verification using the BM. The process begins with the initialization of elliptic curve parameters, followed by the generation of a private $key\ d \in [1, n-1]$ and the corresponding public key $Q = d \cdot G$, computed using BM. Upon receiving the input message m, utilizing a cryptographic hash function to compute message digest $e = HASH(m)$. A random scalar k is chosen, and the point $R = k \cdot G$ generated using the BM. The x-coordinate of R is reduced modulo n to obtain r. The signature component s is then calculated as $s = k - 1(e + d \cdot r) mod\ n$. The signature pair *(r, s)* is output and used, along with the message and public key, for signature verification. If the verification process confirms the validity of the signature, the result returns true. Else, it returns false, it means that generated an invalid signature.

4.1.4 ECDSA using Elliptic Net Method

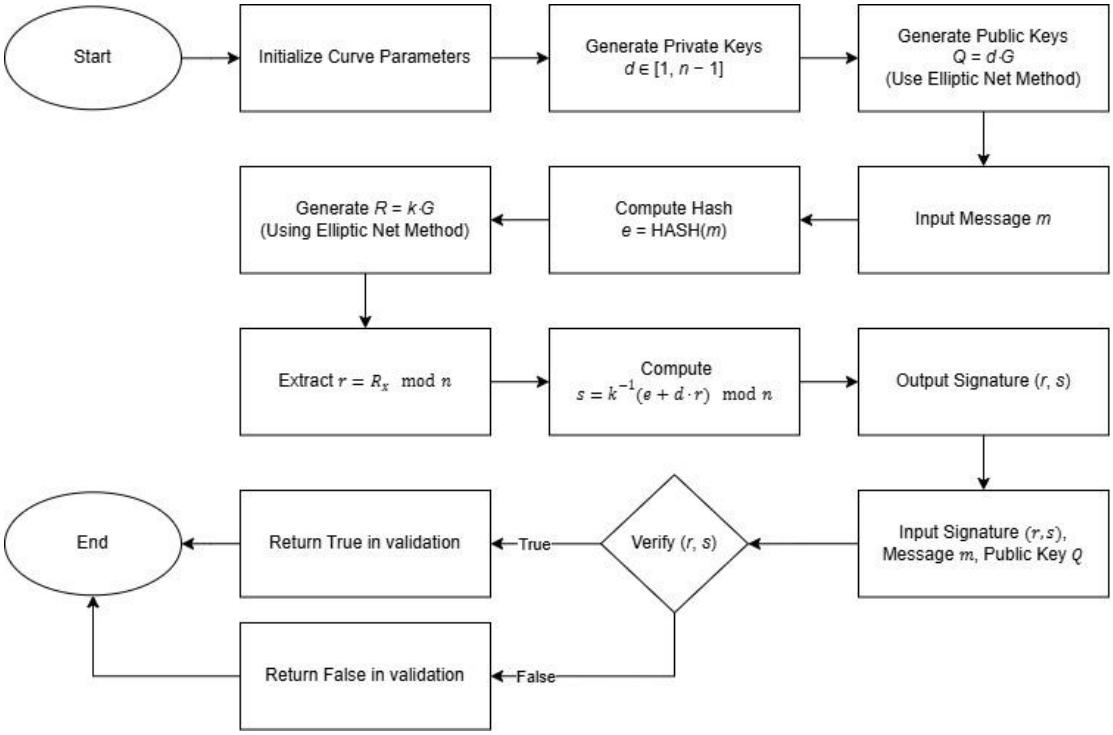


Figure 4.4 ECDSA Key Exchange Process Flowchart Using Elliptic Net Method

Figure 4.4 illustrates the process of ECDSA signature generation and verification using the EN method. The process begins with the initialization of elliptic curve parameters, followed by the generation of a private $key\ d \in [1, n-1]$ and the corresponding public key $Q = d \cdot G$, computed using the EN method. Upon receiving the input message m, a hash function is applied to compute the message digest $e = HASH(m)$. A random scalar k is chosen, and the point $R = k \cdot G$ generated using the EN method. The x-coordinate of R is reduced modulo n to obtain r. The signature component s is then calculated as $s = k - 1(e + d \cdot r)\ mod\ n$. The signature pair *(r, s)* is output and used, along with the message and public key, for signature verification. If the verification process confirms the validity of the signature, the output returns true. Otherwise, it returns false.

4.1.5 NUMS parameter

The NUMS Curve parameters that used in this project are numsp384t1 and numsp512t1, a 384-bit and a 512-bit prime field in Twisted Edwards curve [69]. Both curves are suitable for prime field operations and elliptic curve scalar multiplication,

sharing similar popularity and being well-suited for a wide range of applications. The NUMS parameter for Twisted Edward curve $ax^2 + y^2 \equiv x^3 + dx^2y^2$ are as follows:

Table 4.1 NUMS parameter [70]

numsp384t1 state as below:
p = 0 X FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFF43 a = 0 X FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFF42 d = 0 X 3BEE G = (0 X 0D, 0 X 7D0AB41E 2A1276DB A3D330B3 9FA046BF BE2A6D63 824D303F 707F6FB5 331CADBA) n = 0 X 3FFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF BE6AA55A D0A6BC64 E5B84E6F 1122B4AD h = 0 X 04
numsp512t1 state as below:
p = 0 X FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFDC7 a = 0 X FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFDC6 d = 0 X 9BAA8 G = (0 x 20, 0 X 7D67E841 DC4C467B 605091D8 0869212F 9CEB124B F726973F 9FF04877 9E1D614E 62AE2ECE 5057B5DA D96B7A89 7C1D7279 92611346 38750F4F 0CB91027 543B1C5E) n = 0 X 3FFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF A7E50809 EFDABBB9 A624784F 449545F0 DCEA5FF0 CB800F89 4E78D1CB 0B5F0189 h = 0 X 04

The parameters for numsp384t1 and numsp512t1 were chosen to balance security and transparency in cryptographic applications. These parameters are very common in modern cryptographic systems and have been adopted by large organizations. The

primes p is selected close to a power of 2 to support efficient modular arithmetic. The coefficients $a = p - 1$ and small constants d are carefully picked to support the fast arithmetic nature of Twisted Edwards curves, enabling efficient and secure scalar multiplication. These choices help ensure optimized performance without compromising on security.

The generator points *G* is fixed and verified to lie on the curve, creating sizable prime-order subgroups, which is necessary to keep the discrete logarithm issue challenging. The subgroup orders *n* are large primes close to 2^{2k} to prevent small subgroup and invalid-curve attacks, while the small cofactor *h* =0 X 4 further minimizes any potential vulnerabilities. These curves are part of the NUMS family, which are designed through transparent processes to avoid hidden parameters and gain trust in their security. Their widespread use in industry and by trusted companies reinforces confidence in their reliability and strength.

4.2 Scalar multiplication algorithms

4.2.1 Scalar multiplication via binary method

The BM converting a scalar multiplication, *n* to binary representation and processing each bit from most significant bit to least significant bit. Points on an elliptic curve are doubled for each bit of the scalar. If the bit is 1, an additional point is added. The steps are illustrated as follows:

Algorithm 4.1. Scalar multiplication via binary method [68]

Input: An Affine point $p \in E(F_p)$ and $n = (n_{i-1}, \dots, n_0)_2$.

Output: $nP \in E(F_p)$

Steps:

1. For i from $l - 2$ down to 0 do
2. $Q \leftarrow 2Q$
3. If $n_i = 1$ then

   3.1 $Q \leftarrow P + Q$
4. Return Q

Algorithm 4.1 is used in elliptic curve cryptography over a prime field $\mathbb{F}_p$. It takes as input a point *P* on an elliptic curve $E(\mathbb{F}_p)$ and a scalar *n*, which is represented in binary

form as $(n_{l-1}, \dots, n_0)_2$. The goal is to compute *nP*, which means adding point *P* to itself *n* times using elliptic curve group operations. The algorithm initializes an accumulator point *Q* with the value of *P*, based on the assumption that the most significant bit n_{l-1} is 1. In each iteration, the point *Q* is doubled, representing a shift in the binary multiplication. The original point P is added to Q if the current bit is 1. This combination of point doubling and conditional addition efficiently computes the scalar multiple *nP* with a number of operations proportional to the bit length of *n*. This algorithm is particularly suited for elliptic curves over finite prime fields due to its straightforward implementation and reasonable performance.

4.2.2 Scalar multiplication via elliptic net

The EN method uses precomputed tables of elliptic curve multiples to accelerate scalar multiplication. The scalar decomposed and precomputed points are used to efficiently compute the result. This process of EN method is shown in Algorithm 4.2.

Algorithm 4.2. Scalar multiplication via elliptic net [23]

Input: Integer $n = (n_{l-1}n_{l-2} \dots n_0)_2$ with $n_{l-1} = 1$. $P \epsilon E(\mathbb{F}_p), a = W_2, b = W_3$ and $c = W_4$ of the EN associated to P and $I = y^{-1}$.

Output: The EN values W_λ where $n - 2 \leq \lambda \leq n + 2$ associated to point P.

Steps:

1. $V \leftarrow [-a, -1, 0, 1, a, b, c, a^3, c - b^3]$
2. For i from $l - 1$ down to 0 do
3. If $n_i = 0$ then
   $V \leftarrow double(V)$
4. Else
   $V \leftarrow doubleadd(V)$
5. $A = V_3^{-1}; B = A^2; C = AB$
6. $E = V_2^2; F = V_4^2; G = V_2V_4$
7. $H = BG; J = EV_5; K = FV_1$
8. $x_n = x_1 - GB$
9. $y_n = (J - K)IC$

Algorithm 4.2 computes the scalar multiplication *nP* on an elliptic curve over a prime field $\mathbb{F}_p$ using EN. It begins with the scalar *n* expressed in binary form and a point *P* on

the elliptic curve. The algorithm uses the initial EN values $W_2 = a,\ W_3 = b$ and $W_4 = c$, along with the inverse of the *y*-coordinate, denoted as $I = y^{-1}$. It initializes vector *V* with a set of values derived from the net that will be used to recursively compute further net terms. The algorithm processes the bits of the scalar *n* from the most significant to the least significant bit. At each step, depending on whether the current bit is 0 or 1, the algorithm performs a doubling or a combined doubling and addition operation on the vector *V*, updating its entries to reflect the current state of the scalar multiplication. These operations manipulate the EN values rather than the elliptic curve points directly. After processing all bits, the algorithm computes the final coordinates (x_n, y_n) of the resulting point *nP* using algebraic expressions involving the updated vector entries. These expressions combine squares, products, and the inverse *I* to extract the coordinates from the net representation. The result is an efficient computation of *nP* using the structure and recurrence properties of EN.

4.3 Scheme of algorithms

4.3.1 Algorithm for ECDH scheme

ECDH enhanced connection between user A and user B to exchange keys securely. Both generates a private key and public key. Next, exchange public keys to another and use private key public key received to compute a shared secret. The process is shown as below:

Algorithm 4.3. ECDH scheme [67]

User A	User B
Alice and Bob exchange a Prime(P) and Generator(G), such that $P > G$.	
Generate a random number, X_A	Generate a random number, X_B
Generate public key, $Y_A = G^{X_A}(mod\ P)$	Generate public key, $Y_B = G^{X_B}(mod\ P)$
Receive Y_B	Receive Y_A
Secret Key = $Y_B^{X_A}(mod\ P)$	Secret Key = $Y_B^{X_A}(mod\ P)$
Both secret keys are the same number.	

The Diffie-Hellman Key Exchange protocol, which allows two users to safely create a shared secret across an unsecure channel, is shown in Algorithm 4.3. Both users agree on a large prime number *P* and a generator *G*, where *P* > *G*. Each user then picks a

private random number, X_A and X_B. Both random numbers are used to compute their public key, ($Y_A = G^{X_A} \text{mod P}, Y_B = G^{X_B} \text{ mod P}$). After exchanging public keys, each user raises the received public key to the power of their private key, resulting in the same shared secret, $Y_B^{X_A}\ mod\ P$ has the same value as $Y_A^{X_B}\ mod\ P$. This shared secret can then be used in future cryptographic operations.

4.3.2 Algorithm for ECDSA scheme

ECDSA provided digital signatures, steps are signing and verification of messages. Sender generates private keys and public keys. A message is signed with the private key, creating a digest that is verified using the public key, ensuring the message's integrity and authenticity as shown below:

Algorithm 4.4. ECDSA scheme [4]

Generation steps: 1. Select a random integer k, $1 \leq k \leq n-1$. 2. Compute $kG = (x_1, y_1)$ and $r = x_1\ mod\ n$. If $r = 0$, go back to step 1. 3. Compute $k^{-1}\ mod\ n$ 4. Compute $e = SHA-1(m)$ 5. Compute $s = k^{-1}(e + dr)\ mod\ n$. If $s = 0$ then go to step 1. 6. Signature for the message, m is (r, s)
Verification steps: 1. Verify (r, s) are integers in the interval $[1, n-1]$ 2. Compute $e = SHA-1(m)$ 3. Compute $w = s^{-1}\ mod\ n$ 4. Compute $u_1 = ew\ mod\ n$ and $u_2 = rw\ mod\ n$ 5. Compute $X = u_1G + u_2Q$. If $X \neq 0$, compute $v = x_1\ mod\ n$ where $X = (x_1, y_1)$ 6. Accept signature if and only $v = r$.

Algorithm 4.4 describes the ECDSA operation, which is used to generate and verify digital signatures. In the signature generation process, a random integer *k* is selected and used to compute a point *kG*. While he x-coordinate of this point modulo *n* becomes *r*. If *r* = 0, a new *k* is chosen. The hash of the message *m* is computed, and the signature

component *s* is calculated using the formula $s = k^{-1}(e + dr) \bmod n$, where *d* is the private key. If *s* = 0, the process restarts. The signature is the pair (*r*, *s*).

In the verification step, the verifier first checks that *r* and *s* are within valid bounds. They then compute the message hash *e*, followed by $w = s^{-1} \bmod n$, and use it to compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$. Using the public key *Q*, they compute the point $X = u_1 G + u_2 Q$. If $X \neq 0$, the verifier computes $v = x_1 \bmod n$ from the x-coordinate of *X* and accepts the signature only if *v* = *r*.

Chapter 5
Experiment

5.1 Hardware Setup

The hardware involved in this project is a laptop. A laptop is issued for the process of implementation of coding from ECC scalar multiplication and SCAs to obtain the outcome for further analysis, then the process needs to computer large value calculations.

Table 5.1 Specifications of laptop

Description	Specifications
Model	Asus TUF Gaming FX705-GM
Processor	Intel Core i7-8750H
Operating System	Windows 11
Graphic	NVIDIA GeForce GTX 1060 6GB DDR5
Memory	8GB X 2 DDR5 RAM
Storage	512 GB SSD

5.2 Software Setup

In this project, the programming language selected is Python. Python has the most user-friendly interface among C, C++, Java and others. Python codes are easy to read. Python code uses English keywords rather than punctuation, and its line breaks help define the code blocks [64]. In addition, Python codes are extendable [64], Python code can be written in other programming languages as examples stated above.

To compile Python code, Anaconda Navigator 3 provides different applications to choose, and the applications will be discussed later. The Anaconda platform is the most popular way to learn and use Python for scientific computing [65], especially current project includes the large numbers mathematical calculation.

In Avaconda3, several Python development applications are provided, such as PyCharm, Jupyter Notebook/Lab, Spyder and Visual Studio Code. The Jupyter Lab is selected because it allows for a platform to make it easier to learn Python programming fundamentals [66]. Jupyter has two versions, the classic version chosen and the

notebook version. In this project, the lab version has been selected. The lab version contains more libraries and better experience. The Jupyter Lab performs efficiently in resource management, it handles large notebooks and multiple open files without significant slowdowns and restores the user's workspace, reopening the lab where the user left off.

5.3 Simulation of ECDH

5.3.1 First Implementation of Binary Method

Algorithm 5.1. ECDH using Binary Method

Key Generation Steps:

1. Generate Private Key:

   Select a random integer k, where $1 \leq k \leq n - 1$.
2. Generate Public Key:

   Compute $P = kG = (x, y)$ using scalar multiplication with BM.

Shared Secret Computation Steps:

Alice's Side:

1. Generate a private key a, where $1 \leq a \leq n - 1$.
2. Compute public key $A = aG$.
3. Receive Bob's public key $B = bG$.
4. Compute shared secret point $S_A = aB = (x_s, y_s)$.
5. Use the x-coordinate of the shared secret:

   Shared secret = x_s.

Bob's Side:

1. Generate a private key b, where $1 \leq b \leq n - 1$.
2. Compute public key $B = bG$.
3. Receive Bob's public key $A = aG$.
4. Compute shared secret point $S_b = bA = (x_s, y_s)$.
5. Use the x-coordinate of the shared secret:

   Shared secret = x_s.

Scalar Multiplication – Used in All Key/Public/Secret Computations:

1. Represent scalar k in binary.

   $k = (k_t, \ldots, k_0)_2$
2. Initialize:

$R = (0, 1)$ (the identity point).

3. For each bit *k* from MSB to LSB:

   a. $R = 2R$ using Twisted Edwards point doubling.

   or

   b. If $k_i = 1$, then $R = R + P$ using Twisted Edwards point addition.

4. Output:

   $R = kP$

Twisted Edwards Point Addition (P + *Q*):

Given $P = (x_1, y_1)$, $Q = x_2, y_2$, computes:

1. $x_3 = \frac{x_1 y_2 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2} \ mod\ p.$
2. $y_3 = \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \ mod\ p$
3. Output:

   $R = (x_3, y_3).$

Twisted Edwards Point Doubling (2*P*):

Given $P = (x_1, y_1)$, compute:

1. $x_3 = \frac{2 x_1 y_1}{1 + x_1^2 y_1^2} \ mod\ p.$
2. $y_3 = \frac{y_1^2 - a x_1^2}{1 - d x_1^2 y_1^2} \ mod\ p.$
3. Output:

   $R = (x_3, y_3).$

Algorithm 5.1 describes the full procedure of ECDH key exchange, focusing on the use of scalar multiplication via the BM with Twisted Edwards curve arithmetic. In the key generation phase, a private key is selected randomly within a valid range, and the corresponding public key is computed by multiplying the private scalar with the base point *G* using binary scalar multiplication. During the shared secret computation, both users generate their own key pairs and computes the shared secret point by multiplying their private key with the received public key, resulting in the same shared point $S = abG$. Only the x-coordinate of this point is used as the final shared secret. Scalar multiplication, which is central to all these operations, is performed by converting the scalar into binary and iterating through each bit from the most significant to least significant. For each bit, point doubling is always performed, and point addition is done

only when the bit is 1, using Twisted Edwards point addition and doubling formulas. This method provides guaranteed security level in elliptic curve cryptography. The Python codes for Algorithm 5.1 can be seen in Appendix A.

Output

```
# Alice's keys
alice_private_key = generate_private_key()
alice_public_key = generate_public_key(alice_private_key)

# Bob's keys
bob_private_key = generate_private_key()
bob_public_key = generate_public_key(bob_private_key)

# Alice and Bob compute the shared secret
alice_shared_secret = ecdh_shared_secret(alice_private_key, bob_public_key)
bob_shared_secret = ecdh_shared_secret(bob_private_key, alice_public_key)

# The shared secrets should be the same
print(f"Alice's Private Key  : {alice_private_key}")
print(f"Bob's Private Key    : {bob_private_key}")
print(f"Alice's Shared Secret: {alice_shared_secret}")
print(f"Bob's Shared Secret  : {bob_shared_secret}")
print(f"Shared secrets match : {alice_shared_secret == bob_shared_secret}")
```

```
Alice's Private Key  : 1403054703912739598346691325483440088989938492681967932856233302484181334603738426084691783136465099784220468933243
Bob's Private Key    : 8653169276965376713008053667127677795698362517634732056349440646215382710620542080373125187948574796749950909606744
Alice's Shared Secret: 257460443057424427117298677156525781270269377742875475714544940737624727296691969786558915090085824198975534701045 72
Bob's Shared Secret  : 257460443057424427117298677156525781270269377742875475714544940737624727296691969786558915090085824198975534701045 72
Shared secrets match : True
```

Output

```
# Alice's keys
alice_private_key = generate_private_key()
alice_public_key = generate_public_key(alice_private_key)

# Bob's keys
bob_private_key = generate_private_key()
bob_public_key = generate_public_key(bob_private_key)

# Alice and Bob compute the shared secret
alice_shared_secret = ecdh_shared_secret(alice_private_key, bob_public_key)
bob_shared_secret = ecdh_shared_secret(bob_private_key, alice_public_key)

# The shared secrets should be the same
print(f"Alice's Private Key  : {alice_private_key}")
print(f"Bob's Private Key    : {bob_private_key}")
print(f"Alice's Shared Secret: {alice_shared_secret}")
print(f"Bob's Shared Secret  : {bob_shared_secret}")
print(f"Shared secrets match : {alice_shared_secret == bob_shared_secret}")
```

```
Alice's Private Key  : 1134974104382974194984646735363764795001163926533199232181565580891298432716281769808281516820976707636852407597680758790699036820632798518115136513875654
Bob's Private Key    : 7657253876751742478063858438091599910993682653515916926990735090146419712282729083704949978643515145962269521063233073365361561910606955118809436350877 75
Alice's Shared Secret: 5686775246257483116077267169625035394255167850008014863223342996152244093353050509708211875204088275063832707621908830429808815621211321648933948982701516
Bob's Shared Secret  : 5686775246257483116077267169625035394255167850008014863223342996152244093353050509708211875204088275063832707621908830429808815621211321648933948982701516
Shared secrets match : True
```

Figure 5.1 Output of first implementation on ECDH

Figure 5.1 shows the result of the ECDH key exchange using two different key lengths. The first output uses the numsp384t1 curve, which has a shorter key length, while the second output uses the numsp512t1 curve with a longer key. The results confirm that both parties have the same shared secret, and the output shows true, meaning the key exchange worked correctly.

5.3.2 Second implementation on Elliptic Net Method

Algorithm 5.2. ECDH using Elliptic Net Method

Key Generation Steps (For both Alice and Bob)

1. Select private key

   Randomly choose an integer $k \in [1, n-1]$.

2. Compute public key

   Use EN scalar multiplication:

   Let $Q \leftarrow Q(0,1)$ (neutral element)

   Repeat while $k \neq 0$:

   If the least significant bit of *k* is 1:

   Set $Q \leftarrow Q + P$ using Twisted Edwards point addition.

   Set $P \leftarrow 2P$ using Twisted Edwards point addition

   Right-shift *k* by 1 (i.e., $k = k \gg 1$)

   End loop

   Return *Q* as public key

Shared Secret Derivation Steps (For Alice and Bob)

1. Each party computes shared secret

   Given private key d_A peer's public key Q_B:

   Compute $S = d_A \cdot Q_B$ using EN scalar multiplication

Return *x*-coordinate of *S* as the shared secret

Twisted Edwards Point Addition Formula

Given two points $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and parameters *a*, *d* and prime *p*:

2. Compute intermediate values:

   $A = x_1 \cdot y_2 + x_2 \cdot y_1$

   $B = 1 + d \cdot x_1 \cdot x_2 \cdot y_1 \cdot y_2$

   $C = y_1 \cdot y_2 - a \cdot x_1 \cdot x_2$

   $D = 1 - d \cdot x_1 \cdot x_2 \cdot y_1 \cdot y_2$

3. Compute output point:

   $x_3 = A \cdot B^{-1} \bmod p$

   $y_3 = C \cdot D^{-1} \bmod p$

Return (x_3, y_3)

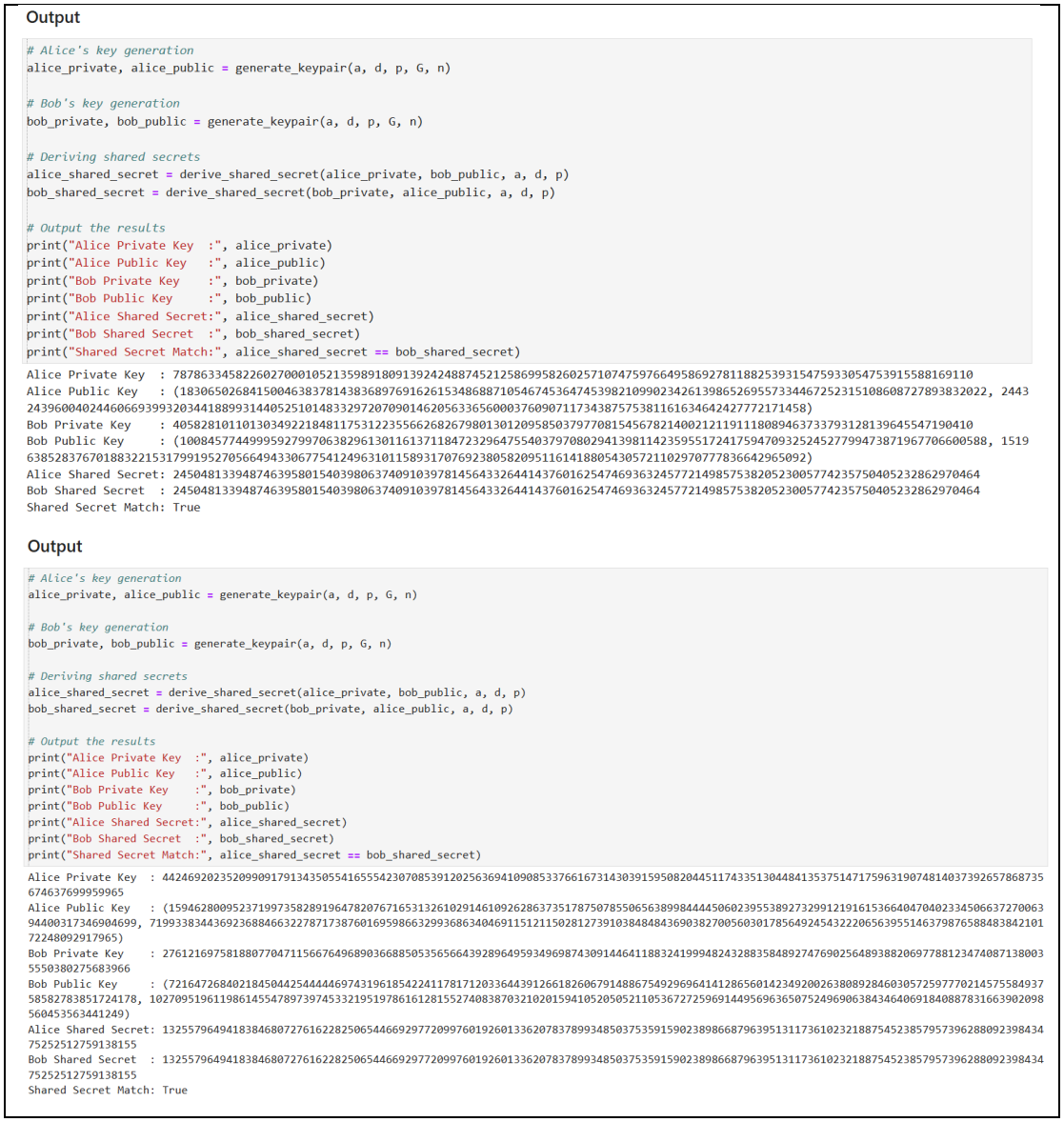
Shared Secret Match Condition

After computation by both parties:

- Let $S_A = x$-coordinate from Alice's shared point.
- Let $S_B = x$-coordinate from Bob's shared point.

Algorithm 5.2 explains the ECDH key exchange using binary scalar multiplication on Twisted Edwards curves. Each party randomly selects a private key $k \in [1, n-1]$, then computes the public key using scalar multiplication via the EN method. The scalar is processed bit by bit. If the bit is 1, a point addition is done, then the point is always doubled. All point operations use Twisted Edwards addition formulas.

Both users perform scalar multiplication for the shared secret key using their private key and the peer's public key. The x-coordinate of the resulting point is used as the shared secret. If both x-coordinates match, the key exchange is successful, confirming both parties derived the same shared secret. The Python codes for Algorithm 5.2 can be seen in Appendix B.



Output

```
# Alice's key generation
alice_private, alice_public = generate_keypair(a, d, p, G, n)

# Bob's key generation
bob_private, bob_public = generate_keypair(a, d, p, G, n)

# Deriving shared secrets
alice_shared_secret = derive_shared_secret(alice_private, bob_public, a, d, p)
bob_shared_secret = derive_shared_secret(bob_private, alice_public, a, d, p)

# Output the results
print("Alice Private Key  :", alice_private)
print("Alice Public Key   :", alice_public)
print("Bob Private Key    :", bob_private)
print("Bob Public Key     :", bob_public)
print("Alice Shared Secret:", alice_shared_secret)
print("Bob Shared Secret  :", bob_shared_secret)
print("Shared Secret Match:", alice_shared_secret == bob_shared_secret)
```

```
Alice Private Key  : 787863345822602700010521359891809139242488745212586995826025710747597664958692781188253931547593305475391558816911 0
Alice Public Key   : (183065026841500463837814383689769162615348688710546745364745398210990234261398652695573344672523151086087279383202 2, 2443
2439600402446066939932034418899314405251014833297207090146205633656000376090711734387575381161634642427772171458)
Bob Private Key    : 405828101101303492218481175312235566268267980130120958503797708154567821400212119111808946373379312813964554719041 0
Bob Public Key     : (100845774499959279970638296130116137118472329647554037970802941398114235955172417594709325245277994738719677066005 88, 1519
638528376701883221531799195270566494330677541249631011589317076923805820951161418805430572110297077783664296509 2)
Alice Shared Secret: 245048133948746395801540398063740910397814564332644143760162547469363245772149857538205230057742357504052328629704 64
Bob Shared Secret  : 245048133948746395801540398063740910397814564332644143760162547469363245772149857538205230057742357504052328629704 64
Shared Secret Match: True
```

Output

```
# Alice's key generation
alice_private, alice_public = generate_keypair(a, d, p, G, n)

# Bob's key generation
bob_private, bob_public = generate_keypair(a, d, p, G, n)

# Deriving shared secrets
alice_shared_secret = derive_shared_secret(alice_private, bob_public, a, d, p)
bob_shared_secret = derive_shared_secret(bob_private, alice_public, a, d, p)

# Output the results
print("Alice Private Key  :", alice_private)
print("Alice Public Key   :", alice_public)
print("Bob Private Key    :", bob_private)
print("Bob Public Key     :", bob_public)
print("Alice Shared Secret:", alice_shared_secret)
print("Bob Shared Secret  :", bob_shared_secret)
print("Shared Secret Match:", alice_shared_secret == bob_shared_secret)
```

```
Alice Private Key  : 4424692023520990917913435055416555423070853912025636941090853376616731430391595082044511743351304484135375147175963190748140373926578687356
74637699959965
Alice Public Key   : (1594628009523719973582891964782076716531326102914610926286373517875078550656389984444506023955389273299121916153664047040233450663727006 3
9440031734690469 9, 7199338344369236884663227871738760169598663299368634046911512115028127391038484843690382700560301785649245432220656395514637987658848384210 1
7224809291796 5)
Bob Private Key    : 2761216975818807704711566764968903668850535656643928964959349698743091446411883241999482432883584892747690256489388206977881234740871380 3
5550380275683966
Bob Public Key     : (7216472684021845044254444469743196185422411781712033644391266182606791488675492969641412865601423492002638089284603057259777021457558493 7
58582783851724178, 1027095196119861455478973974533219519786161281552740838703210201594105205052110536727259691449569636507524969063843464069184088783116639020 9
8560453563441249)
Alice Shared Secret: 1325579649418384680727616228250654466929772099760192601336207837899348503753591590238986687963951311736102321887545238579573962880923984 34
7525251275913815 5
Bob Shared Secret  : 1325579649418384680727616228250654466929772099760192601336207837899348503753591590238986687963951311736102321887545238579573962880923984 34
7525251275913815 5
Shared Secret Match: True
```

Figure 5.2 Output of second implementation on ECDH

Figure 5.2 shows that Alice and Bob generate their key pairs using the same scheme and method. The private keys are randomly selected, and the public keys are computed using EN scalar multiplication. Then, both parties compute a shared secret using each other's public key and their own private key.

The shorter private key is using numsp384t1 secure curve parameter, resulting in smaller public key and shared secret values. The longer private key, producing longer key values with numsp512t1 secure curve parameter. Despite the key size difference, both figures show that the shared secret from users are matched, confirming the key exchange was successful as shown (Shared Secret Match: True) in the output.

5.4 Simulation based on ECDSA

5.4.1 First Implementation of Binary Method

Algorithm 5.3. ECDSA using Binary Method

Key Generation Steps:

1. Generate private key:

   Select a random integer *d*, where $1 \leq d \leq n - 1$.

2. Generate public key:

   Compute $Q = dG = (x_Q, y_Q)$ using scalar multiplication with BM.

Message Signing Steps:

1. Input:

   Message *m*, private key *d*, base point *G*, curve parameters *a*, *d*, *p* and order *n*.

2. Hash the message:

   Compute *e* = SHA-256$(m) mod\ n$.

3. Select ephemeral key:

   Choose a random integer *k*, where $1 \leq k \leq n - 1$ and $\gcd(k, n) = 1$.

4. Calculate point $R = kG = (x_R, y_R)$ using binary scalar multiplication.

5. Compute:

   $r = x_R\ mod\ n.$

   $k^{-1}\ mod\ n.$

   $s = k^{-1} \cdot (e + d \cdot r) mod\ n.$

6. Check $s \neq 0$. If not, repeat from step 3.

7. Output the signature:

   (*r*, *s*)

Signature Verification Steps:

1. Input:

   Signature (*r*, *s*), message *m*, public key *Q*, base point *G*, curve parameters *a*, *d*, *p* and order *n*.

2. Check that:

   $r \in [1, n - 1]$

   $s \in [1, n - 1]$

3. Hash the message:

   e = SHA-256$(m)\ mod\ n$.

4. Compute:

   $w = s^{-1} \, mod \, n$

   $w_1 = e \cdot w \, mod \, n$

   $w_2 = r \cdot w \, mod \, n$

5. Calculate point $R = u_1 G + u_2 Q = (x_R, y_R)$ using binary scalar multiplication and Twisted Edwards point addition.
6. Signature is valid if:

   $r \equiv x_R \, mod \, n$

Twisted Edwards Point Addition (P + *Q*):

Given $P = (x_1, y_1)$, $Q = x_2, y_2$, computes:

1. $x_3 = \frac{x_1 y_2 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2} \, mod \, p.$
2. $y_3 = \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \, mod \, p$
3. Output:

   $R = (x_3, y_3).$

Binary Scalar Multiplication (*kP*):

1. Convert *k* to binary:

   $k = (k_t, \ldots, k_0)_2$

2. Initialize: *Q* = (0, 1) (identity element)
3. Loop through bits of *k*:

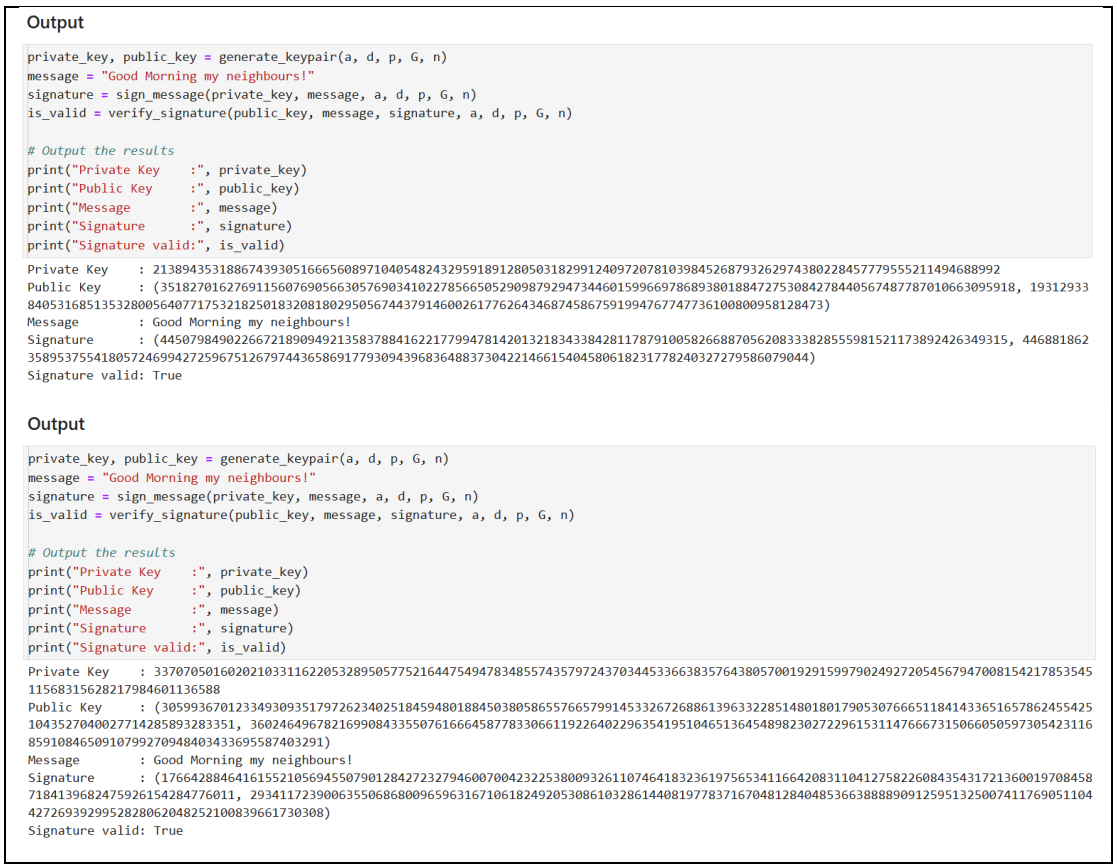
   Double: *Q* = *Q* + *Q*

   If bit is 1: *Q* = *Q* + *P*
4. Return:

   Q = *kP*

Algorithm 5.3 explains the process of ECDSA using binary scalar multiplication with Twisted Edwards curves. The key generation starts by selecting a random private key *d*, then computing the public key *Q* = *dG* using scalar multiplication. To sign a message, the message must proceed using SHA-256 technique. A random ephemeral key *k* is chosen to compute the point *R* = *kG*, and its x-coordinate becomes *r*. The value *s* is calculated using $s = k^{-1}(e + dr) mod \, n$, where *e* is the hash of the message. The signature is the pair (*r*, *s*). To verify the signature, the verifier checks the range of *r* and *s*, then hashes the message to get *e*. Using the signature and public key, the verifier

computes a new point $R = w_1G + w_2Q$, where w_1 and w_2 derived from *r*, *s*, and *e*. If the x-coordinate of this point equals *r*, the signature is valid. All scalar multiplications are done using the BM and point additions/doublings use Twisted Edwards formulas. The Python codes for Algorithm 5.3 can be seen in Appendix C.



Output

```
private_key, public_key = generate_keypair(a, d, p, G, n)
message = "Good Morning my neighbours!"
signature = sign_message(private_key, message, a, d, p, G, n)
is_valid = verify_signature(public_key, message, signature, a, d, p, G, n)

# Output the results
print("Private Key    :", private_key)
print("Public Key     :", public_key)
print("Message        :", message)
print("Signature      :", signature)
print("Signature valid:", is_valid)
```

```
Private Key    : 213894353188674393051666560897104054824329591891280503182991240972078103984526879326297438022845777955521149468899 2
Public Key     : (351827016276911560769056630576903410227856650529098792947344601599669786893801884727530842784405674877870106630959 18, 19312933
840531685135328005640771753218250183208180295056744379146002617762643468745867591994767747736100800958128473)
Message        : Good Morning my neighbours!
Signature      : (445079849022667218909492135837884162217799478142013218343384281178791005826688705620833382855598152117389242634931 5, 446881862
358953755418057246994272596751267974436586917793094396836488373042214661540458061823177824032727958607904 4)
Signature valid: True
```

Output

```
private_key, public_key = generate_keypair(a, d, p, G, n)
message = "Good Morning my neighbours!"
signature = sign_message(private_key, message, a, d, p, G, n)
is_valid = verify_signature(public_key, message, signature, a, d, p, G, n)

# Output the results
print("Private Key    :", private_key)
print("Public Key     :", public_key)
print("Message        :", message)
print("Signature      :", signature)
print("Signature valid:", is_valid)
```

```
Private Key    : 337070501602021033116220532895057752164475494783485574357972437034453366383576438057001929159979024927205456794700815421785354 5
115683156282179846011365 88
Public Key     : (305993670123349309351797262340251845948018845038058655766579914533267268861396332285148018017905307666511841433651657862455425
104352704002771428589328335 1, 360246496782169908433550761666458778330661192264022963541951046513645489823027229615311476667315066050597305423116
859108465091079927094840343369558740329 1)
Message        : Good Morning my neighbours!
Signature      : (176642884641615521056945507901284272327946007004232253800932611074641832361975653411664208311041275822608435431721360019708458
718413968247592615428477601 1, 293411723900635506868009659631671061824920530861032861440819778371670481284048536638888909125951325007411769051104
427269392995282806204825210083966173030 8)
Signature valid: True
```

Figure 5.3 Output of first implementation on ECDSA

Figure 5.3 shows the process of ECDSA signature generation and verification using different key pairs. In both cases, private key used to sign message *'Good Morning my neighbours!'*, and the public key is used to verify the signature. All outputs confirm that the signature was successfully generated and validated for both key sizes. Despite the longer keys producing longer signature values, the result is the same: "Signature valid: True", meaning the message integrity and authenticity are verified.

5.4.2 Second implementation of Elliptic Net Method

Algorithm 5.4. ECDSA using Elliptic Net Method

Key Generation Steps (For both Alice and Bob)

1. Select private key

   Randomly choose an integer d $\in [1, n-1]$.

2. Compute public key

   $Q = d \cdot G$

Signature Generation

1. Input: Message *m*, private key *d*, curve parameters *a*, *d*, *p*, generator *G*, and order *n.*
2. Hash the message:

   Compute *e* = SHA-256$(m) mod\ n$
3. Generate random nonce:

   Choose random $k \in [1, n-1]$ such that $\gcd(k, n) = 1$
4. Compute ephemeral point:

   $R = k \cdot G$ using EN scalar multiplication

   Extract $r = x_R\ mod\ n$
5. Compute signature component:

   $s = k^{-1}(e + r \cdot d) mod\ n$
6. Output signature:

   Return (*r*, *s*)

Signature Verification

1. Input: Message *m*, public key *Q*, signature (*r*, *s*) and curve parameters
2. Check bounds:

   If $r \notin [1, n-1]$ or $s \notin [1, n-1]$, reject
3. Hash the message:

   Compute *e* = SHA-256$(m)\ mod\ n$
4. Compute inverse:

   $s^{-1}\ mod\ n$
5. Compute scalar values:

   $u_1 = e \cdot s^{-1}\ mod\ n$, $u_2 = r \cdot s^{-1}\ mod\ n$
6. Compute point:

$R = u_1 \cdot G + u_2 \cdot Q$ using EN scalar multiplication and point addition

7. Check validity:

   Signature is valid if $r \equiv x_R \, mod \, n$

Twisted Edwards Point Addition

Given points $P = (x_1, y_1)$, $Q = (y_2, y_2)$ curve parameters:

1. $A = x_1 y_2 + x_2 y_1$
2. $B = 1 + d x_1 x_2 y_1 y_2$
3. $C = y_1 y_2 - a x_1 x_2$
4. $D = 1 - d x_1 x_2 y_1 y_2$
5. Compute:

   $x_3 = A \, / B \, mod \, p$

   $y_3 = C \, / D \, mod \, p$
6. Return (x_3, y_3)

Algorithm 5.4 describes the full procedure of digital signature generation and verification using elliptic curves, specifically utilizing scalar multiplication and Twisted Edwards curve arithmetic. In the signature generation phase, the sender chooses a private key and generate the public key by multiplying it with base point *G*. To sign the message, the message must proceed using SHA-256 technique and reduced modulo *n*, producing a digest *e*. A random nonce $k \in [1, n-1]$ is then chosen such that $\gcd(k, n) = 1$. The ephemeral point $R = k \cdot G$ computed using binary scalar multiplication, and its x-coordinate mod *n* is taken as *r*. The final signature component *s* is calculated as $s = k^{-1}(e + r \cdot d) mod \, n$, forming the signature pair (*r*, *s*).

In the verification phase, the verifier first checks that *r* and *s* lie within the valid range. Then, the message is hashed again and reduced to get *e*, and the inverse of *s* modulo *n* is computed. The values $R = u_1 \cdot G + u_2 \cdot Q$ and $u_2 = r \cdot s^{-1} \, mod \, n$ are used to compute the verification point $R = u_1 \cdot G + u_2 \cdot Q$. This point is calculated using scalar multiplication and Twisted Edwards point addition. If the x-coordinate of $R \, mod \, n$ equals *r*, the signature is considered valid. The use of binary scalar multiplication and Twisted Edwards formulas ensures efficient and secure computations throughout the signature process. The Python codes for Algorithm 5.4 can be seen in Appendix D.

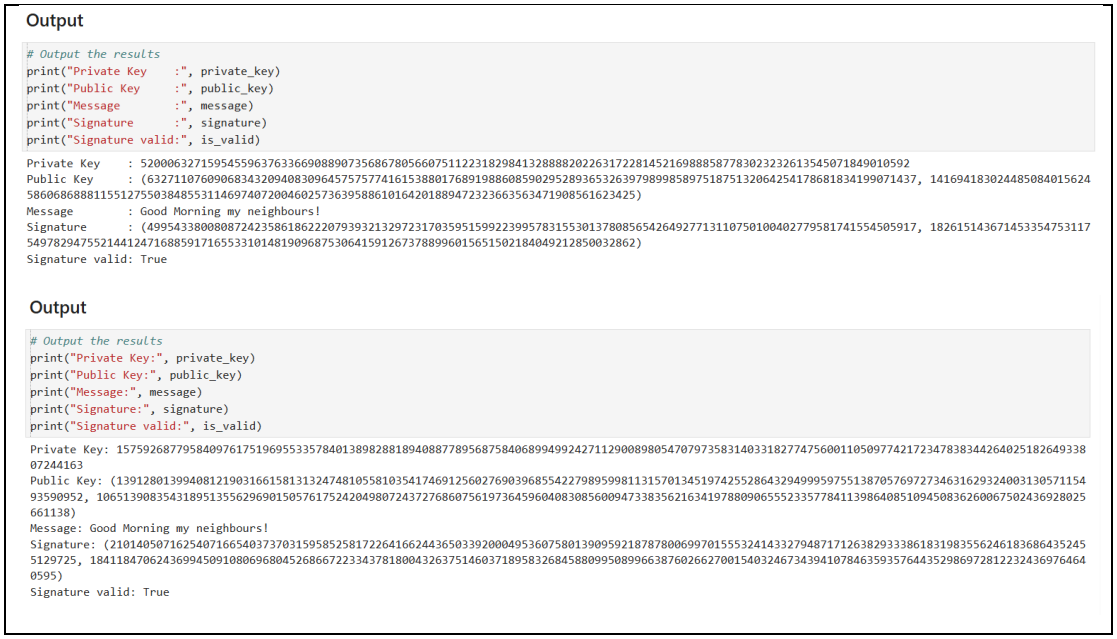

Output
# Output the results
print("Private Key :", private_key)
print("Public Key :", public_key)
print("Message :", message)
print("Signature :", signature)
print("Signature valid:", is_valid)
Private Key : 52000632715954559637633669088907356867805660751122318298413288882022631722814521698885877830232326135450718490105 92
Public Key : (63271107609068343209408309645757577416153880176891988608590295289365326397989858975187513206425417868183419907 1437, 14169418302448508401562458606868881155127550384855311469740720046025736395886101642018894723236635634719085616234 25)
Message : Good Morning my neighbours!
Signature : (49954338008087242358618622207939321329723170359515992239957831553013780856542649277131107501004027795817415545059 17, 18261514367145335475311754978294755214412471688591716553310148190968753064159126737889960156515021840492128500328 62)
Signature valid: True
Output
# Output the results
print("Private Key:", private_key)
print("Public Key:", public_key)
print("Message:", message)
print("Signature:", signature)
print("Signature valid:", is_valid)
Private Key: 157592687795840976175196955335784013898288189408877895687584068994992427112900898054707973583140331827747560011050977421723478383442640251826493380 7244163
Public Key: (139128013994081219031661581313247481055810354174691256027690396855422798959981131570134519742552864329499597551387057697273463162932400313057115493590952, 106513908354318951355629690150576175242049807243727686075619736459604083085600947338356216341978809065552335778411398640851094508362600675024369280256611 38)
Message: Good Morning my neighbours!
Signature: (210140507162540716654037370315958525817226416624436503392000495360758013909592187878006997015553241433279487171263829333861831983556246183686435245 5129725, 184118470624369945091080696804526866722334378180043263751460371895832684588099508996638760266270015403246734394107846359357644352986972812232436976464 0595)
Signature valid: True

Figure 5.4 Output of second implementation on ECDSA

Figure 5.4 shows successful execution of a digital signature scheme using Twisted Edwards Curve cryptography. In each case, the public key is created using the private key, and the message signed using a random nonce, producing unique (*r*, *s*) signature values. The signature is then verified using the corresponding public key and original message. Although the inputs differ between the two outputs, both correctly validate the signatures, demonstrating consistent and secure implementation of the signing and verification process.

5.5 Implementation Issues and Challenges

One of the most technically challenging aspects of the implementation phase was converting the EN scalar multiplication algorithm into working code. The algorithm involved a structured and layered computation model with recursive logic, making it significantly more complex to translate into a stable and reliable program. Implementing the EN method across four algorithm variations, which included ECDH and ECDSA using both numsp384t1 and numsp512t1 parameters, required careful attention to the sequence of point operations and value tracking. Mistakes in indexing or recursive calculations often resulted in subtle errors that were difficult to detect during testing. Due to its mathematical depth and dependency on correct ordering, the process demanded considerable time, experimentation, and validation to achieve functional and accurate results across all implementations.

Following this, the implementation of the ECDH scheme also presented its own challenges. Although the ECDH algorithm is conceptually straightforward, ensuring correct key generation, point multiplication, and shared secret computation required careful handling of curve parameters and scalar values. The use of different curve sizes such as numsp384t1 and numsp512t1 introduced additional complexity. Proper configuration and parameter consistency were necessary to maintain the integrity of the key exchange process across different test cases.

Another major issue encountered was the mismatch in the computed shared secret during ECDH testing. During certain early testing, users engaged in the key exchange produced different shared secret values, indicating a problem with scalar multiplication or key configuration. This issue was traced back to inconsistencies in private key formatting, mismatched bit lengths, or incorrect conversion between coordinate representations. After identifying these inconsistencies, the key generation process was corrected to ensure the private keys had the proper bit lengths and that the scalar multiplication was carried out in the affine coordinate system with consistent formatting on both sides.

5.6 Concluding Remark

Chapter 5 presents the successful implementation of elliptic curve-based cryptographic schemes, specifically ECDH and ECDSA, using Twisted Edwards curves. The implementation begins with environment setup and key generation, followed by secure message signing and verification.

Two scalar multiplication methods were applied which are the BM and the EN methods. The BM processes each scalar bit using point doubling and conditional addition, while the EN method uses structured point sequences to minimize leakage and optimize performance. Experimental result on numsp384t1 and numsp512t1 curve confirmed that key change in ECDH and digital signature in ECDSA were executed correctly. The outputs validate the correctness of the operations and provide a strong foundation for later SCAs evaluations. These implementations set the stage for analysing the cryptographic strength and resistance of each method in Chapter 6.

Chapter 6

System Evaluation and Discussion

6.1 System Testing

This chapter focuses on two types of attacks which are timing attack and power analysis attack. Timing attacks are applied to ECDH using both the BM and EN method, according to Algorithm 5.1 and Algorithm 5.2. Power analysis attacks are applied only to ECDSA, Algorithm 5.3 for BM and Algorithm 5.4 for EN method.

6.1.1 Timing Attack on Algorithm 5.1

Algorithm 6.1. Timing Attack on Algorithm 5.1

Steps:

1. Initialize *guessed_key* ← 0
2. Initialize *timing_diffs* ← empty list
3. For *i* from 0 to *key_size - 1* do:
    1. Let *k0* ← guessed_key
    2. Let *k1* ← guessed_key with bit *(key_size - 1 - i)* set to 1
    3. Measure execution time *t0* for *scalar_multiplication_binary(k0, G)*
    4. Measure execution time *t1* for *scalar_multiplication_binary(k1, G)*
    5. Append *(t1 - t0)* to *timing_diffs*
    6. If *t1* > t*0*:

        Set *guessed_key* ← k1 (bit is likely 1) Else:

        Set *guessed_key* ← k0 (bit is likely 0)
    7. Mask *guessed_key* to ensure it remains within 384 or 512 bits
4. Return *guessed_key*, *timing_diffs*

Algorithm 6.1 outlines a timing attack performed on Algorithm 5.1 to recover a private key used in binary scalar multiplication. It begins by initializing a guessed key to zero and setting up an empty list to store timing differences. The attack works bit by bit, starting from the most significant bit down to the least significant. In each iteration, two candidate keys are created, the current guess, *k0*. And *k1*, which is the same as *k0* but with the current bit set to 1. The execution time of the scalar multiplication using each key is measured with the function *scalar_multiplication_binary*(*k*, *G*), where *G* is the

base point. The timing difference between *k1* and *k0* is stored. If the execution time for *k1* is greater than for *k0*, it is inferred that the bit is likely 1, and the guessed key is updated accordingly. Otherwise, it remains 0. To ensure the key length stays within 384 or 512 bits, the key is masked in each iteration. Finally, the guessed key and the list of timing differences are returned, providing insight into the potential leakage of key bits through execution time variations in the BM. The Python codes for Algorithm 6.1 can be seen in Appendix E.

6.1.2 Timing Attack on Algorithm 5.2

Algorithm 6.2. Timing Attack on Algorithm 5.2

Steps:

1. Initialize *guessed_key* ← 0
2. Initialize *timing_diffs* ← empty list
3. For *i* from 0 to *key_size - 1* do:
    1. Let *k0* ← guessed_key
    2. Let *k1* ← guessed_key with bit *(key_size - 1 - i)* set to 1
    3. Measure execution time *t0* for *elliptic_net_scalar_mult(k0, G, p, a, d)*
    4. Measure execution time *t1* for *elliptic_net_scalar_mult(k0, G, p, a, d)*
    5. Append *(t1 - t0)* to *timing_diffs*
    6. If *t1* > t*0*:

        Set *guessed_key* ← k1 (bit is likely 1) Else:

        Set *guessed_key* ← k0 (bit is likely 0)
    7. Mask *guessed_key* to ensure it remains within 384 or 512 bits

Return *guessed_key*, *timing_diffs*

Algorithm 6.2 shows the process of performing a timing attack on Algorithm 5.2 to recover a secret key bit by bit. The attack begins by initializing the *guessed_key* to zero and an empty list called *timing_diffs* to store timing differences. The process iteratively guesses each bit of the secret key starting from the most significant bit. In each iteration, two versions of the key are prepared: *k0* as the current guess and *k1* as the guess with the current bit set to 1. Both versions are used in scalar multiplication using the *elliptic_net_scalar_mult* function, and their execution times (*t0* and *t1*) are measured. The difference *t1 - t0* is recorded in the timing_diffs list. If *t1* is greater than *t0*, it is

assumed that the key bit is likely 1, so the bit is kept set in the guess. Else, it remains 0. This process continues for all bits in the key, and the *guessed_key* is masked at each step to ensure it fits within the 384-bit or 512-bit key size. In the end, the function returns both the guessed key and the list of timing differences, giving insights into where the algorithm may leak information based on execution time. The Python codes for Algorithm 6.2 can be seen in Appendix F.

6.1.3 Power Analysis Attack on Algorithm 5.3

Algorithm 6.3. Power Analysis Attack on Algorithm 5.3

Steps:

1. Generate 1000 random 8-bit plaintext values:
   plaintexts ← random integers in [0, 255], size = *num_samples*
2. Generate a 48-byte secret nonce
   true_nonce ← random integers in [0, 383], size = 48 or 64
3. Simulate Power Traces
   For each nonce byte k_i in *true_nonce*:
   For each plaintext p, compute *hamming_weight*$(p \oplus k_i)$
   Add Gaussian noise:
   $trace_i = HW(p \oplus k_i)$ + noise
   Store $trace_i$ in *power_traces*
4. Correlate Guesses
   For each byte index *i* in nonce:
   Initialize list byte_correlations ← []
   For every guess g in [0, 255]:
   Compute $HW(p \oplus g)$ + noise
   Calculate Pearson correlation with *power_traces[i]*
   Store result in *byte_correlations*
   Determine the best guess *g** with the highest correlation
   Validate guess:
   if *g** == *true_nonce[i]* and correlation > threshold → count as correct
5. Evaluate Attack Success
   Compute *success_rate* = (correct guesses / 48 or 64) × 100%
   Output per-byte result: true vs guessed nonce and validation

Plot correlation for byte 0 to visualize power leak

Algorithm 6.3 shows the designed power analysis attack performed on Algorithm 5.3 to recover a secret nonce used in elliptic curve cryptography by analyzing power consumption traces. It starts by generating a secret nonce, which is a 48-byte (384-bit) or 64-byte (512-bit) value, and then simulates power traces. For each byte of the nonce, the algorithm calculates its Hamming weight . If the number of 1s in its binary form, it helps to calculate how much power is used. This step is to form the power analysis attack. The simulated power values are then mixed with random noise to make the traces more realistic, imitating the imperfections seen in real devices. These noisy power traces are then used to carry out the power analysis attack.

The next step is to correlate guesses for each byte of the nonce with the real power traces. For each byte of the nonce, the algorithm tries all possible guesses (from 0 to 255), computes the predicted power consumption traces for each guess, and calculates the Pearson correlation between each predicted trace and the real power traces. The guess that results in the highest correlation is considered the best guess for that byte. If the correlation exceeds a certain threshold and the guessed byte matches the actual nonce byte, the guess is validated as correct. After evaluating all 48 bytes or 64 bytes of the nonce, the attack success rate is calculated by measuring the percentage of correctly guessed bytes. The algorithm also outputs the per-byte results and generates a plot to visualize the correlation for the first byte of the nonce. This entire process is designed to recover the secret nonce by exploiting power leakage during the cryptographic operation, demonstrating how power analysis attacks can expose vulnerabilities to cryptographic systems. The Python codes for Algorithm 6.3 can be seen in Appendix G.

6.1.4 Power Analysis Attack on Algorithm 5.4

Algorithm 6.4. Power Analysis Attack on Algorithm 5.4

Steps:

1. Generate 1000 random 8-bit plaintext values:
   plaintexts ← random integers in [0, 255], size = *num_samples*
2. Generate a 48-byte secret nonce
   true_nonce ← random integers in [0, 383 or 511], size = 48 or 64
3. Simulate Power Traces
   For each nonce byte k_i in *true_nonce*:
   For each plaintext p, compute *hamming_weight* ($p \oplus k_i$)
   Add Gaussian noise:
   $trace_i = HW(p \oplus k_i)$ + noise
   Store $trace_i$ in *power_traces*
4. Correlate Guesses
   For each byte index *i* in nonce:
   Initialize list byte_correlations ← []
   For every guess g in [0, 255]:
   Compute $HW(p \oplus g)$ + noise
   Calculate Pearson correlation with *power_traces[i]*
   Store result in *byte_correlations*
   Determine the best guess *g** with the highest correlation
   Validate guess:
   if *g** == *true_nonce[i]* and correlation > threshold → count as correct
5. Evaluate Attack Success
   Compute *success_rate* = (correct guesses / 48 or 64) × 100%
   Output per-byte result: true vs guessed nonce and validation
   Plot correlation for byte 0 to visualize power leak

Algorithm 6.4 illustrates power analysis attack implementation on Algorithm 5.4. This algorithm is similar to the BM, Algorithm 6.3 but incorporates differences in simulated power consumption traces. First, the algorithm generates 1000 random 8-bit plaintext values and a 48-byte secret nonce or a 64-byte secret nonce. The power traces are simulated by calculating the Hamming weight between each plaintext byte and the

corresponding nonce byte, just as in the BM. However, the simulation uses a more complex model that accounts for elliptic curve scalar multiplication, where power consumption may depend on the specific operation. Gaussian noise is then added to the traces to reflect real-world conditions, and these noisy traces are stored for further analysis.

To recover the nonce, the algorithm then attempts to correlate all possible guesses for each byte of the nonce with the actual power traces. For each nonce byte, it iterates over all potential guesses, calculates the Hamming weight for each guess, and computes the Pearson correlation between the predicted power traces and the actual traces. The best guess is determined by identifying the guess that produces the highest correlation for that byte. If the correlation exceeds a predefined threshold and the guessed byte matches the actual nonce byte, the guess is validated. After processing all 48 bytes or 64 bytes of the nonce, the attack's success rate is evaluated by calculating the percentage of correct guesses. A correlation plot for the first byte is generated to visually inspect the strength of the power leak, helping to demonstrate the effectiveness of the attack on elliptic curve cryptographic systems. The Python codes for algorithm 6.4 can be seen in Appendix H.

6.2 Testing Setup and Result

6.2.1 Timing Attack Implementation for Algorithm 6.1 (numsp384t1)

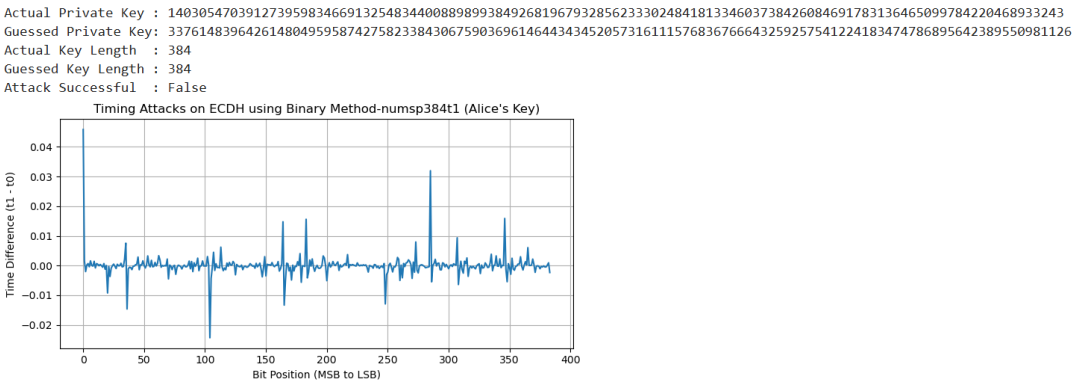


Figure 6.1 Timing Attack on Algorithm 6.1 (numsp384t1)

Figure 6.1 represents a timing attack on ECDH using BM with numsp384t1 secure curve parameter. The blue line shows how the time difference between two operations changes for each bit position of the key, with the x-axis representing the bit positions from most significant to least significant, and the y-axis showing the time difference.

In this graph, the blue line has several noticeable spikes. These spikes suggest that the computation time varies more significantly at those bits, likely due to how the algorithm processes bits set to 1 versus 0. For example, certain operations might take longer when the bit is 1, which can unintentionally leak information about the key. These visible spikes could help an attacker make educated guesses about the value of specific bits. However, while the timing differences are more pronounced here than in the second graph (Figure 6.2), they were still not consistent or accurate enough for the attack to succeed in fully recovering the correct private key.

6.2.2 Timing Attack Implementation for Algorithm 6.1 (numsp512t1)

```
Actual Private Key : 113497410438297419498464673536376479500116392653319923218156558089129843271628176980828151682097670763685240759768075879069903682063279851811513651387565 4
Guessed Private Key: 129986043255962213879486947480607637664933359863767761670739476160231735851891621675093009266022624259221181116993107500405913807823335370442140691945276 60
Actual Key Length  : 512
Guessed Key Length : 512
Attack Successful  : False
```

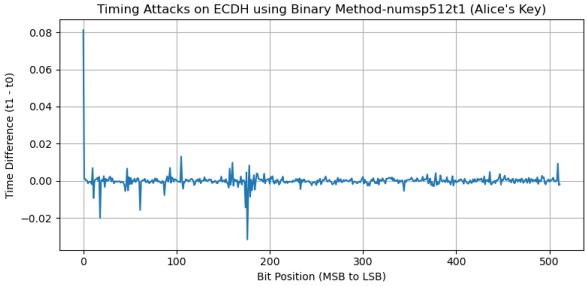


Figure 6.2 Timing Attack on Algorithm 6.1 (numsp512t1)

Figure 6.2 shows a timing attack output on the ECDH using BM but with different secure curve parameters. The numsp512t1 secure curve parameter uses a longer 512-bit private key compared to the 384-bit key in the previous graph. As before, timing attacks were implemented to guess Alice's private key by analyzing how long the cryptographic operations take. The x-axis represents the position of each bit in the private key, from the most significant to the least important, and the y-axis shows the difference in computation time between the two operations. The blue line indicates how this time difference varies across the key bits. The blue line appears much more stable, with fewer and smaller spikes. This suggests that the implementation for numsp512t1 is less susceptible to timing variations, making it more resistant to this type of SCA. Although there are still minor spikes, they are less pronounced and do not provide enough information to recover the private key, which is why the attack was unsuccessful.

6.2.3 Timing Attack Implementation for Algorithm 6.2 (numsp384t1)

```
Actual Private Key : 787863345822602700010521359891809139242488745212586995826025710747597664958692781188253931547593305475391558816911O
Guessed Private Key: 279254757744331213726064369584637198574541629718933621831510605115173268103329487745218467857529981457113633856844860
Actual Key Length  : 384
Guessed Key Length : 384
Attack Successful  : False
```

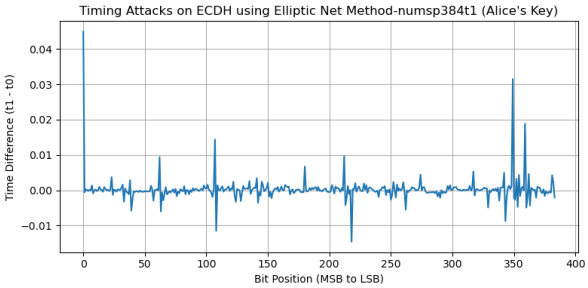


Figure 6.3 Timing Attack on Algorithm 6.2 (numsp384t1)

Figure 6.3 shows the timing attack results on ECDH using the EN method with the numsp384t1 curve. The blue line represents the timing differences measured at each bit position during scalar multiplication, where the x-axis shows the bit positions from the most significant to the least significant bit. Ideally, if the scalar multiplication operation is fully protected, the timing differences should remain flat and close to zero across all bit positions. However, in this graph, several noticeable spikes are visible, particularly around bit position 350, indicating that some bits cause slight variations in computation time. These spikes suggest possible timing leakage due to internal branching, memory handling, or point addition complexity, which could theoretically allow an attacker to guess bit values based on timing anomalies. Nevertheless, compared to previous results from the BM, the amplitude and frequency of these spikes are much lower and less consistent. The overall blue line remains relatively stable without significant fluctuations, meaning that less information is leaked overall. Despite minor fluctuations, the guessed key generated from this timing analysis did not match the actual private key, demonstrating that the EN method significantly improves resistance against timing attacks, even when slight leakage is present.

6.2.4 Timing Attack Implementation for Algorithm 6.2 (numsp512t1)

```
Actual Private Key : 44246920235209909179134350554165554230708539120256369410908533766167314303915950820445117433513044841353751471759631907481403739265786873567463769995965
Guessed Private Key: 77830734526505113576801909358736048501335895184049789963516243554398379797502127337072570948395980307980223934127724726558049619027946766810676055969868 72
Actual Key Length  : 512
Guessed Key Length : 512
Attack Successful  : False
```

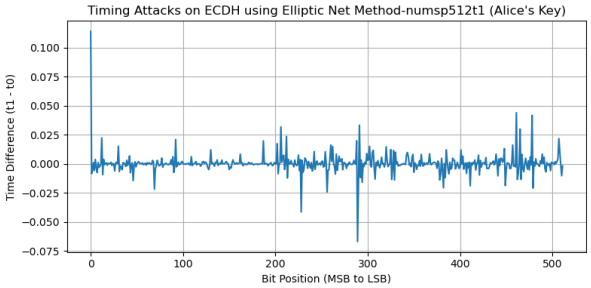


Figure 6.4 Timing Attack on Algorithm 6.2 (numsp512t1)

Figure 6.4 shows the timing attack results on ECDH using the EN method with the numsp512t1 curve. The blue line represents the timing differences measured at each bit position during scalar multiplication, with the x-axis displaying bit positions from the most significant to the least significant bit across the 512-bit key. Ideally, a well-protected implementation would produce a flat line close to zero, and in this figure, the blue line remains relatively stable with only minor fluctuations. Small spikes appear throughout the graph, particularly bit positions 475, suggesting slight timing variations caused by certain bits, but the magnitude of these spikes is small and inconsistent.

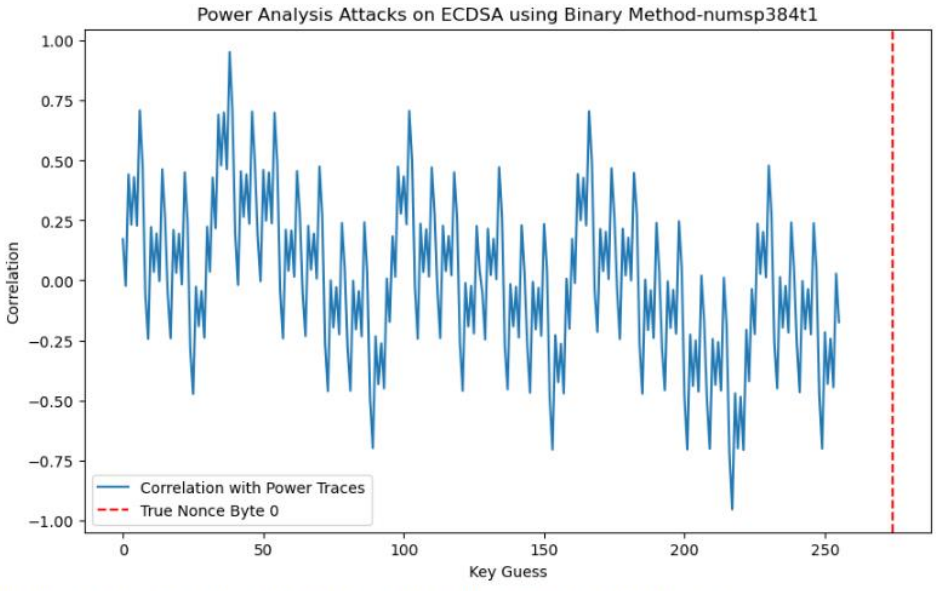
Timing attacks on ECDH using EN with different parameters produce slightly different results. Compared to Figure 6.3, Figure 6.4 shows even smaller and more scattered timing differences, with no significant concentration of leakage at specific bit positions. Despite minor fluctuations observed in both figures, the guessed keys did not match the actual private keys, demonstrating that the EN method consistently offers strong resistance to timing attacks, even when slight leakages are present.

Overall, the timing attack on ECDH shows that the EN method offers better resistance compared to the BM. When comparing Figure 6.3 and Figure 6.4, both based on the EN method, the graph for numsp512t1 (Figure 6.4) is more stable with fewer timing fluctuations than numsp384t1 (Figure 6.3). This suggests that increasing the key size improves resistance to timing-based analysis, and the structured nature of the EN method further reduces potential leakage. This is evident in the graphs where the blue lines in EN show smaller and fewer timing spikes across bit positions, indicating more

consistent execution. The reduced variation makes it harder for attackers to distinguish between key bits, proving that EN leaks less timing information than BM.

6.2.5 Power Analysis Attack Implementation for Algorithm 6.3 (numsp384t1)

```
Byte 0 - True Byte: 274, Guessed Byte: 18, Validation: False
Byte 1 - True Byte: 141, Guessed Byte: 141, Validation: True
Byte 2 - True Byte: 306, Guessed Byte: 50, Validation: False
Byte 3 - True Byte: 378, Guessed Byte: 122, Validation: False
Byte 4 - True Byte: 295, Guessed Byte: 39, Validation: False
Byte 5 - True Byte: 339, Guessed Byte: 83, Validation: False
Byte 6 - True Byte: 11, Guessed Byte: 11, Validation: True
Byte 7 - True Byte: 84, Guessed Byte: 84, Validation: True
Byte 8 - True Byte: 32, Guessed Byte: 32, Validation: True
Byte 9 - True Byte: 208, Guessed Byte: 208, Validation: True
Byte 10 - True Byte: 100, Guessed Byte: 100, Validation: True
Byte 11 - True Byte: 147, Guessed Byte: 147, Validation: True
Byte 12 - True Byte: 336, Guessed Byte: 80, Validation: False
Byte 13 - True Byte: 339, Guessed Byte: 83, Validation: False
Byte 14 - True Byte: 168, Guessed Byte: 168, Validation: True
Byte 15 - True Byte: 214, Guessed Byte: 214, Validation: True
Byte 16 - True Byte: 304, Guessed Byte: 48, Validation: False
Byte 17 - True Byte: 77, Guessed Byte: 77, Validation: True
Byte 18 - True Byte: 56, Guessed Byte: 56, Validation: True
Byte 19 - True Byte: 372, Guessed Byte: 116, Validation: False
Byte 20 - True Byte: 323, Guessed Byte: 67, Validation: False
Byte 21 - True Byte: 303, Guessed Byte: 47, Validation: False
Byte 22 - True Byte: 80, Guessed Byte: 80, Validation: True
Byte 23 - True Byte: 132, Guessed Byte: 132, Validation: True
Byte 24 - True Byte: 336, Guessed Byte: 80, Validation: False
Byte 25 - True Byte: 319, Guessed Byte: 63, Validation: False
Byte 26 - True Byte: 364, Guessed Byte: 108, Validation: False
Byte 27 - True Byte: 217, Guessed Byte: 217, Validation: True
Byte 28 - True Byte: 27, Guessed Byte: 27, Validation: True
Byte 29 - True Byte: 117, Guessed Byte: 117, Validation: True
Byte 30 - True Byte: 233, Guessed Byte: 233, Validation: True
Byte 31 - True Byte: 107, Guessed Byte: 107, Validation: True
Byte 32 - True Byte: 184, Guessed Byte: 184, Validation: True
Byte 33 - True Byte: 206, Guessed Byte: 206, Validation: True
Byte 34 - True Byte: 97, Guessed Byte: 97, Validation: True
Byte 35 - True Byte: 296, Guessed Byte: 40, Validation: False
Byte 36 - True Byte: 223, Guessed Byte: 223, Validation: True
Byte 37 - True Byte: 347, Guessed Byte: 91, Validation: False
Byte 38 - True Byte: 321, Guessed Byte: 65, Validation: False
Byte 39 - True Byte: 255, Guessed Byte: 255, Validation: True
Byte 40 - True Byte: 237, Guessed Byte: 237, Validation: True
Byte 41 - True Byte: 259, Guessed Byte: 3, Validation: False
Byte 42 - True Byte: 103, Guessed Byte: 103, Validation: True
Byte 43 - True Byte: 147, Guessed Byte: 147, Validation: True
Byte 44 - True Byte: 246, Guessed Byte: 246, Validation: True
Byte 45 - True Byte: 50, Guessed Byte: 50, Validation: True
Byte 46 - True Byte: 234, Guessed Byte: 234, Validation: True
Byte 47 - True Byte: 38, Guessed Byte: 38, Validation: True
Overall success rate: 62.50%
```




```
Original True Nonce: [274 141 306 378 295 339  11  84  32 208 100 147 336 339 168 214 304  77
  56 372 323 303  80 132 336 319 364 217  27 117 233 107 184 206  97 296
 223 347 321 255 237 259 103 147 246  50 234  38]
```

Figure 6.5 Power Analysis Attack on Algorithm 6.3 (numsp384t1)

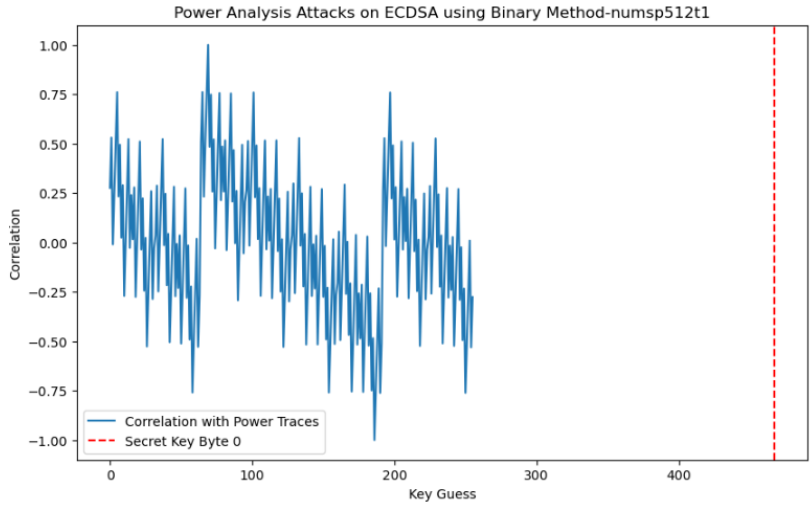
Figure 6.5 shows a power analysis attack on ECDSA using the BM with numsp384t1 secure curve parameter. The blue line represents the correlation between power traces and each guessed key value for nonce byte 0, with the x-axis showing possible byte guesses (0–255) and the y-axis showing the strength of correlation ranging from -1 to 1. The red dashed line marks the position of the true nonce byte, but since it does not align with a high peak in the blue line, the guess for byte 0 was incorrect. This aligns with Figure 6.5, where each nonce byte guess is validated against the true value, and only 30 out of 48 bytes were correctly guessed, giving an overall success rate of 62.5%.

6.2.6 Power Analysis Attack Implementation for Algorithm 6.3 (numsp512t1)

Figure 6.6 visualizes a power analysis attack on ECDSA using the BM on the numsp512t1 secure curve parameter. The x-axis represents all possible byte guesses (from 0 to 255), while the y-axis shows the correlation between each guess and the actual power consumption measured during computation. The blue line shows the correlation values for each key guess, the peaks in this line suggest potential correct guesses. The red dashed line marks the actual secret key byte (Byte 0), and ideally, the blue line should show a sharp peak at this point. However, the blue line remains noisy, and the red line does not align with the highest peak, indicating difficulty in identifying the correct byte using this method. The outcome of power analysis attacks on higher-bit secure curve parameters achieved a 54.69% overall success rate in correctly recovering individual key bytes; the true value attempt was 35 out of 64.

```
Byte 0 - True Byte: 467, Guessed Byte: 211, Validation: False
Byte 1 - True Byte: 205, Guessed Byte: 205, Validation: True
Byte 2 - True Byte: 203, Guessed Byte: 203, Validation: True
Byte 3 - True Byte: 81, Guessed Byte: 81, Validation: True
Byte 4 - True Byte: 250, Guessed Byte: 250, Validation: True
Byte 5 - True Byte: 59, Guessed Byte: 59, Validation: True
Byte 6 - True Byte: 258, Guessed Byte: 2, Validation: False
Byte 7 - True Byte: 342, Guessed Byte: 86, Validation: False
Byte 8 - True Byte: 22, Guessed Byte: 22, Validation: True
Byte 9 - True Byte: 191, Guessed Byte: 191, Validation: True
Byte 10 - True Byte: 193, Guessed Byte: 193, Validation: True
Byte 11 - True Byte: 438, Guessed Byte: 182, Validation: False
Byte 12 - True Byte: 58, Guessed Byte: 58, Validation: True
Byte 13 - True Byte: 190, Guessed Byte: 190, Validation: True
Byte 14 - True Byte: 382, Guessed Byte: 126, Validation: False
Byte 15 - True Byte: 511, Guessed Byte: 255, Validation: False
Byte 16 - True Byte: 48, Guessed Byte: 48, Validation: True
Byte 17 - True Byte: 15, Guessed Byte: 15, Validation: True
Byte 18 - True Byte: 42, Guessed Byte: 42, Validation: True
Byte 19 - True Byte: 116, Guessed Byte: 116, Validation: True
Byte 20 - True Byte: 354, Guessed Byte: 98, Validation: False
Byte 21 - True Byte: 179, Guessed Byte: 179, Validation: True
Byte 22 - True Byte: 479, Guessed Byte: 223, Validation: False
Byte 23 - True Byte: 419, Guessed Byte: 163, Validation: False
Byte 24 - True Byte: 297, Guessed Byte: 41, Validation: False
Byte 25 - True Byte: 499, Guessed Byte: 243, Validation: False
Byte 26 - True Byte: 371, Guessed Byte: 115, Validation: False
Byte 27 - True Byte: 386, Guessed Byte: 130, Validation: False
Byte 28 - True Byte: 24, Guessed Byte: 24, Validation: True
Byte 29 - True Byte: 190, Guessed Byte: 190, Validation: True
Byte 30 - True Byte: 158, Guessed Byte: 158, Validation: True
Byte 31 - True Byte: 225, Guessed Byte: 225, Validation: True
Byte 32 - True Byte: 72, Guessed Byte: 72, Validation: True
Byte 33 - True Byte: 391, Guessed Byte: 135, Validation: False
Byte 34 - True Byte: 377, Guessed Byte: 121, Validation: False
Byte 35 - True Byte: 334, Guessed Byte: 78, Validation: False
Byte 36 - True Byte: 399, Guessed Byte: 143, Validation: False
Byte 37 - True Byte: 143, Guessed Byte: 143, Validation: True
Byte 38 - True Byte: 6, Guessed Byte: 6, Validation: True
Byte 39 - True Byte: 75, Guessed Byte: 75, Validation: True
Byte 40 - True Byte: 495, Guessed Byte: 239, Validation: False
Byte 41 - True Byte: 98, Guessed Byte: 98, Validation: True
Byte 42 - True Byte: 311, Guessed Byte: 55, Validation: False
Byte 43 - True Byte: 451, Guessed Byte: 195, Validation: False
Byte 44 - True Byte: 34, Guessed Byte: 34, Validation: True
Byte 45 - True Byte: 144, Guessed Byte: 144, Validation: True
Byte 46 - True Byte: 123, Guessed Byte: 123, Validation: True
Byte 47 - True Byte: 329, Guessed Byte: 73, Validation: False

Byte 48 - True Byte: 425, Guessed Byte: 169, Validation: False
Byte 49 - True Byte: 257, Guessed Byte: 1, Validation: False
Byte 50 - True Byte: 257, Guessed Byte: 1, Validation: False
Byte 51 - True Byte: 173, Guessed Byte: 173, Validation: True
Byte 52 - True Byte: 405, Guessed Byte: 149, Validation: False
Byte 53 - True Byte: 439, Guessed Byte: 183, Validation: False
Byte 54 - True Byte: 295, Guessed Byte: 39, Validation: False
Byte 55 - True Byte: 440, Guessed Byte: 184, Validation: False
Byte 56 - True Byte: 242, Guessed Byte: 242, Validation: True
Byte 57 - True Byte: 229, Guessed Byte: 229, Validation: True
Byte 58 - True Byte: 163, Guessed Byte: 163, Validation: True
Byte 59 - True Byte: 85, Guessed Byte: 85, Validation: True
Byte 60 - True Byte: 98, Guessed Byte: 98, Validation: True
Byte 61 - True Byte: 15, Guessed Byte: 15, Validation: True
Byte 62 - True Byte: 427, Guessed Byte: 171, Validation: False
Byte 63 - True Byte: 69, Guessed Byte: 69, Validation: True
Overall success rate: 54.69%
```

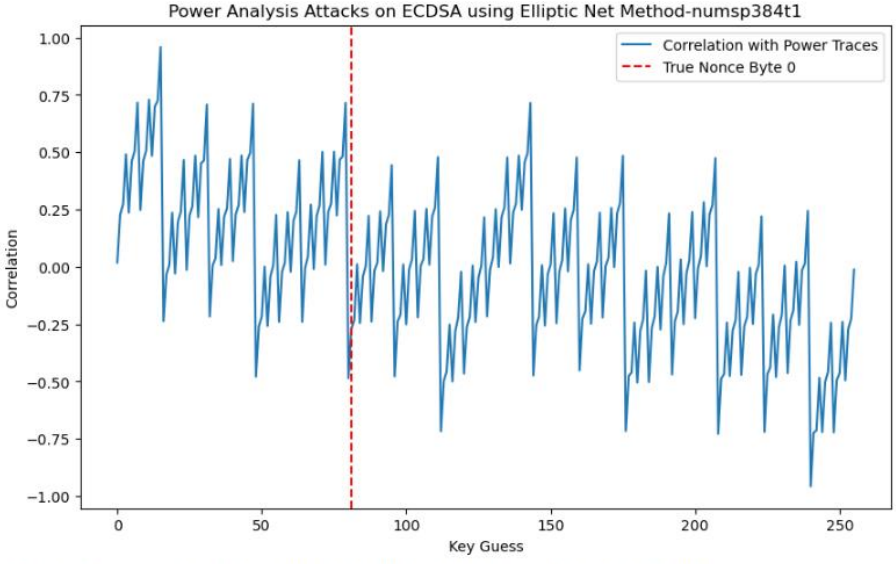



```
Original Secret Key Bytes: [467 205 203  81 250  59 258 342  22 191 193 438  58 190 382 511  48  15
  42 116 354 179 479 419 297 499 371 386  24 190 158 225  72 391 377 334
 399 143   6  75 495  98 311 451  34 144 123 329 425 257 257 173 405 439
 295 440 242 229 163  85  98  15 427  69]
```

Figure 6.6 Power Analysis Attack on Algorithm 6.3 (numsp512t1)

6.2.7 Power Analysis Attack Implementation for Algorithm 6.4 (numsp384t1)

```
Byte 0 - True Byte: 81, Guessed Byte: 81, Validation: True
Byte 1 - True Byte: 60, Guessed Byte: 60, Validation: True
Byte 2 - True Byte: 28, Guessed Byte: 28, Validation: True
Byte 3 - True Byte: 290, Guessed Byte: 34, Validation: False
Byte 4 - True Byte: 318, Guessed Byte: 62, Validation: False
Byte 5 - True Byte: 335, Guessed Byte: 79, Validation: False
Byte 6 - True Byte: 253, Guessed Byte: 253, Validation: True
Byte 7 - True Byte: 339, Guessed Byte: 83, Validation: False
Byte 8 - True Byte: 350, Guessed Byte: 94, Validation: False
Byte 9 - True Byte: 115, Guessed Byte: 115, Validation: True
Byte 10 - True Byte: 189, Guessed Byte: 189, Validation: True
Byte 11 - True Byte: 74, Guessed Byte: 74, Validation: True
Byte 12 - True Byte: 40, Guessed Byte: 40, Validation: True
Byte 13 - True Byte: 336, Guessed Byte: 80, Validation: False
Byte 14 - True Byte: 95, Guessed Byte: 95, Validation: True
Byte 15 - True Byte: 39, Guessed Byte: 39, Validation: True
Byte 16 - True Byte: 87, Guessed Byte: 87, Validation: True
Byte 17 - True Byte: 165, Guessed Byte: 165, Validation: True
Byte 18 - True Byte: 192, Guessed Byte: 192, Validation: True
Byte 19 - True Byte: 371, Guessed Byte: 115, Validation: False
Byte 20 - True Byte: 101, Guessed Byte: 101, Validation: True
Byte 21 - True Byte: 38, Guessed Byte: 38, Validation: True
Byte 22 - True Byte: 115, Guessed Byte: 115, Validation: True
Byte 23 - True Byte: 183, Guessed Byte: 183, Validation: True
Byte 24 - True Byte: 167, Guessed Byte: 167, Validation: True
Byte 25 - True Byte: 51, Guessed Byte: 51, Validation: True
Byte 26 - True Byte: 21, Guessed Byte: 21, Validation: True
Byte 27 - True Byte: 232, Guessed Byte: 232, Validation: True
Byte 28 - True Byte: 317, Guessed Byte: 61, Validation: False
Byte 29 - True Byte: 161, Guessed Byte: 161, Validation: True
Byte 30 - True Byte: 91, Guessed Byte: 91, Validation: True
Byte 31 - True Byte: 153, Guessed Byte: 153, Validation: True
Byte 32 - True Byte: 279, Guessed Byte: 23, Validation: False
Byte 33 - True Byte: 258, Guessed Byte: 2, Validation: False
Byte 34 - True Byte: 300, Guessed Byte: 44, Validation: False
Byte 35 - True Byte: 16, Guessed Byte: 16, Validation: True
Byte 36 - True Byte: 143, Guessed Byte: 143, Validation: True
Byte 37 - True Byte: 64, Guessed Byte: 64, Validation: True
Byte 38 - True Byte: 129, Guessed Byte: 129, Validation: True
Byte 39 - True Byte: 345, Guessed Byte: 89, Validation: False
Byte 40 - True Byte: 309, Guessed Byte: 53, Validation: False
Byte 41 - True Byte: 34, Guessed Byte: 34, Validation: True
Byte 42 - True Byte: 259, Guessed Byte: 3, Validation: False
Byte 43 - True Byte: 148, Guessed Byte: 148, Validation: True
Byte 44 - True Byte: 263, Guessed Byte: 7, Validation: False
Byte 45 - True Byte: 329, Guessed Byte: 73, Validation: False
Byte 46 - True Byte: 181, Guessed Byte: 181, Validation: True
Byte 47 - True Byte: 271, Guessed Byte: 15, Validation: False
Overall success rate: 64.58%
```




```
Original True Nonce: [ 81  60  28 290 318 335 253 339 350 115 189  74  40 336  95  39  87 165
 192 371 101  38 115 183 167  51  21 232 317 161  91 153 279 258 300  16
 143  64 129 345 309  34 259 148 263 329 181 271]
```

Figure 6.7 Power Analysis Attack on Algorithm 6.4 (numsp384t1)

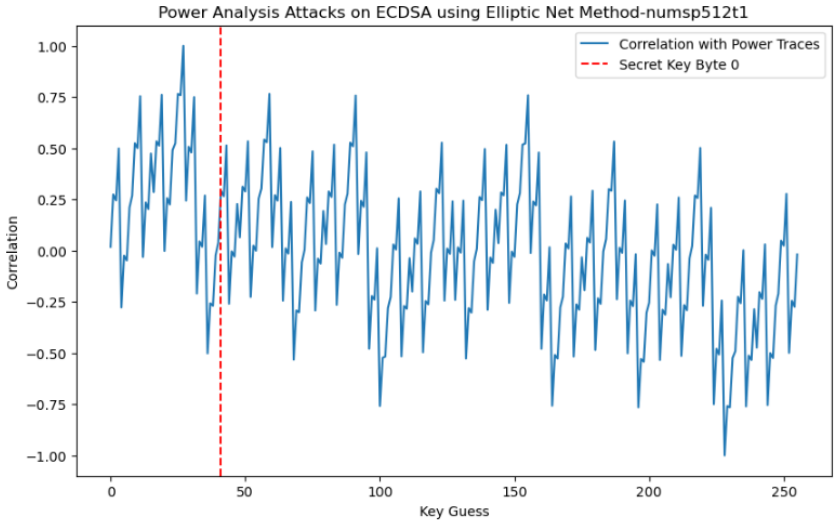
Figure 6.7 displays the results of a power analysis attack on ECDSA using EN method with the numsp384t1 curve. The blue line shows the correlation between key guesses and the collected power traces, where the x-axis represents key guess values ranging

from 0 to 255 and the y-axis indicates correlation values from -1 to 1. Peaks in the blue line near the red line suggest strong correlations where the correct byte guess aligns with the measured power traces. The red dashed vertical line marks the actual value of nonce byte 0, positioned at x = 81. The fluctuations observed across the blue line represent the inherent noise and varying leakage strength in the power traces, with sharper peaks indicating regions of higher information leakage and flatter regions suggesting weaker leakage. Despite the noise, 31 out of 48 nonce bytes were correctly guessed, resulting in an overall success rate of 64.58%, demonstrating moderate attack effectiveness against the numsp384t1 curve.

6.2.8 Power Analysis Attack Implementation for Algorithm 6.4 (numsp512t1)

```
Byte 0 - True Byte: 41, Guessed Byte: 41, Validation: True
Byte 1 - True Byte: 49, Guessed Byte: 49, Validation: True
Byte 2 - True Byte: 312, Guessed Byte: 56, Validation: False
Byte 3 - True Byte: 131, Guessed Byte: 131, Validation: True
Byte 4 - True Byte: 371, Guessed Byte: 115, Validation: False
Byte 5 - True Byte: 71, Guessed Byte: 71, Validation: True
Byte 6 - True Byte: 105, Guessed Byte: 105, Validation: True
Byte 7 - True Byte: 386, Guessed Byte: 130, Validation: False
Byte 8 - True Byte: 480, Guessed Byte: 224, Validation: False
Byte 9 - True Byte: 365, Guessed Byte: 109, Validation: False
Byte 10 - True Byte: 343, Guessed Byte: 87, Validation: False
Byte 11 - True Byte: 186, Guessed Byte: 186, Validation: True
Byte 12 - True Byte: 241, Guessed Byte: 241, Validation: True
Byte 13 - True Byte: 355, Guessed Byte: 99, Validation: False
Byte 14 - True Byte: 417, Guessed Byte: 161, Validation: False
Byte 15 - True Byte: 381, Guessed Byte: 125, Validation: False
Byte 16 - True Byte: 511, Guessed Byte: 255, Validation: False
Byte 17 - True Byte: 491, Guessed Byte: 235, Validation: False
Byte 18 - True Byte: 498, Guessed Byte: 242, Validation: False
Byte 19 - True Byte: 179, Guessed Byte: 179, Validation: True
Byte 20 - True Byte: 322, Guessed Byte: 66, Validation: False
Byte 21 - True Byte: 329, Guessed Byte: 73, Validation: False
Byte 22 - True Byte: 178, Guessed Byte: 178, Validation: True
Byte 23 - True Byte: 450, Guessed Byte: 194, Validation: False
Byte 24 - True Byte: 15, Guessed Byte: 15, Validation: True
Byte 25 - True Byte: 443, Guessed Byte: 187, Validation: False
Byte 26 - True Byte: 346, Guessed Byte: 90, Validation: False
Byte 27 - True Byte: 50, Guessed Byte: 50, Validation: True
Byte 28 - True Byte: 231, Guessed Byte: 231, Validation: True
Byte 29 - True Byte: 211, Guessed Byte: 211, Validation: True
Byte 30 - True Byte: 159, Guessed Byte: 159, Validation: True
Byte 31 - True Byte: 136, Guessed Byte: 136, Validation: True
Byte 32 - True Byte: 281, Guessed Byte: 25, Validation: False
Byte 33 - True Byte: 255, Guessed Byte: 255, Validation: True
Byte 34 - True Byte: 373, Guessed Byte: 117, Validation: False
Byte 35 - True Byte: 463, Guessed Byte: 207, Validation: False
Byte 36 - True Byte: 251, Guessed Byte: 251, Validation: True
Byte 37 - True Byte: 154, Guessed Byte: 154, Validation: True
Byte 38 - True Byte: 377, Guessed Byte: 121, Validation: False
Byte 39 - True Byte: 149, Guessed Byte: 149, Validation: True
Byte 40 - True Byte: 465, Guessed Byte: 209, Validation: False
Byte 41 - True Byte: 414, Guessed Byte: 158, Validation: False
Byte 42 - True Byte: 244, Guessed Byte: 244, Validation: True
Byte 43 - True Byte: 474, Guessed Byte: 218, Validation: False
Byte 44 - True Byte: 413, Guessed Byte: 157, Validation: False
Byte 45 - True Byte: 13, Guessed Byte: 13, Validation: True
Byte 46 - True Byte: 30, Guessed Byte: 30, Validation: True
Byte 47 - True Byte: 338, Guessed Byte: 82, Validation: False

Byte 48 - True Byte: 162, Guessed Byte: 162, Validation: True
Byte 49 - True Byte: 465, Guessed Byte: 209, Validation: False
Byte 50 - True Byte: 16, Guessed Byte: 16, Validation: True
Byte 51 - True Byte: 210, Guessed Byte: 210, Validation: True
Byte 52 - True Byte: 91, Guessed Byte: 91, Validation: True
Byte 53 - True Byte: 72, Guessed Byte: 72, Validation: True
Byte 54 - True Byte: 102, Guessed Byte: 102, Validation: True
Byte 55 - True Byte: 168, Guessed Byte: 168, Validation: True
Byte 56 - True Byte: 427, Guessed Byte: 171, Validation: False
Byte 57 - True Byte: 100, Guessed Byte: 100, Validation: True
Byte 58 - True Byte: 85, Guessed Byte: 85, Validation: True
Byte 59 - True Byte: 197, Guessed Byte: 197, Validation: True
Byte 60 - True Byte: 273, Guessed Byte: 17, Validation: False
Byte 61 - True Byte: 257, Guessed Byte: 1, Validation: False
Byte 62 - True Byte: 410, Guessed Byte: 154, Validation: False
Byte 63 - True Byte: 27, Guessed Byte: 27, Validation: True
Overall success rate: 51.56%
```




```
Original Secret Key Bytes: [ 41  49 312 131 371  71 105 386 480 365 343 186 241 355 417 381 511 491
 498 179 322 329 178 450  15 443 346  50 231 211 159 136 281 255 373 463
 251 154 377 149 465 414 244 474 413  13  30 338 162 465  16 210  91  72
 102 168 427 100  85 197 273 257 410  27]
```

Figure 6.8 Power Analysis Attack on Algorithm 6.4 (numsp512t1)

Figure 6.8 shows the results of a power analysis attack on ECDSA using EN method with the numsp512t1 curve. The plot charts the correlation between key guesses and the collected power traces, where the x-axis represents key guess values from 0 to 255 and the y-axis shows correlation values ranging from -1 to 1. The blue line, representing correlation with power traces, exhibits fluctuating behavior with less distinct peaks, indicating the presence of noise and weaker leakage signals. The value of secret key byte 0 was 41, which is marked by a red dashed vertical line at the corresponding x-axis position. In this attack, 33 out of 6 secret key bytes were correctly guessed, resulting in an overall success rate of 21.56%. These results suggest that significant challenges remain in reliably extracting secret information from the numsp512t1 curve under this attack setup.

The power analysis attack on the numsp384t1 curve achieved a moderate success rate of 64.58%, correctly recovering 31 out of 48 nonce bytes. Power analysis attack on the numsp512t1 curve yielded a much lower success rate of 51.56%, with only 33 out of 64 secret key bytes correctly identified. In Figure 6.7, the correlation graph displays sharper and more prominent peaks near the correct key guesses, indicating more substantial leakage and easier identification of the actual key values. In contrast, Figure 6.8 exhibits noisier, flatter, and less distinguishable correlation patterns, making isolating the correct guesses from noise considerably more challenging. This difference highlights how increasing the key size and using more complex elliptic curve parameters, as with numsp512t1, substantially improve resistance against SCAs by spreading leakage over a larger key space and reducing the correlation strength. Additionally, the longer key length in numsp512t1 increases the number of key bytes that must be guessed correctly, making partial recovery less impactful and full key reconstruction practically infeasible. The results demonstrate that physical noise and algorithmic complexity jointly contribute to the enhanced security of larger elliptic curves against power analysis techniques.

6.3 Project Challenges

Several significant challenges were encountered during this project's development and evaluation stages, particularly in ensuring output correctness, finding suitable references for coding in Python. Due to the lack of supporting external sources, these issues impacted both the technical flow of the project.

A recurring challenge was ensuring that the final output of each implementation was correct and consistent. In some test cases, the shared secret generated in the ECDH protocol did not match between the two parties. Similarly, in ECDSA testing, the digital signature verification failed despite correct input values. These issues highlighted the misconfigurations such as incorrect key formatting, mismatched bit lengths, or improper coordinate conversions. To resolve these problems, each computation step was rechecked thoroughly, and the output at every stage was printed and compared. Public key generation, scalar values, and field parameters were verified individually. For ECDSA, the signature components were recalculated using new random *k* values and point multiplication results were closely monitored until the signature passed validation. This careful, step-by-step debugging process allowed each error to be identified and corrected, eventually leading to valid, reliable outputs.

Another significant difficulty was the limited availability of Python-specific reference materials. While the concepts behind ECC and SCAs are well-documented in general, very few detailed examples or sample codes written in Python could guide the implementation of EN scalar multiplication, ECDSA, or ECDH using the chosen NUMS curves. Most available resources were written in other programming languages or focused only on the high-level logic, without implementation details. Searches through platforms like GitHub, Stack Overflow, and academic coding forums often returned incomplete or unrelated results. As a result, many parts of the code had to be built from scratch based on mathematical definitions and manual interpretation of the algorithmic steps, making the development process slower and more dependent on self-validation.

One of the most challenging aspects of the project was the simulation of SCAs. Although timing and power analysis attacks were implemented and executed against the BM in ECDH and ECDSA, the attacks failed to reveal the private key. The observed timing differences and power variations were insufficient to make accurate guesses about the key values. What made this more challenging was the lack of references validating or supporting the methodology used for these simulations in Python. Without documented benchmarks or comparable examples, it was difficult to determine whether the test setup, measurement techniques, or data analysis strategies were adequate or needed adjustment. This created a sense of limitation, as the project was constrained by a lack of external guidance to confirm the approach taken.

6.4 Objective Evaluation

All three objectives set at the beginning of this project have been successfully achieved through detailed identification, implementation, and simulation. Each objective was addressed methodically, and the outcomes of Chapters 5 and 6 provide strong evidence of their fulfilment.

a. To identify potential vulnerabilities in the scalar multiplication algorithms via binary and EN methods.

This objective was met by performing SCAs on ECC scalar multiplication algorithms using BM and EN methods. The timing attack on ECDH and the power analysis attack on ECDSA revealed how different algorithms leak key-related information. The results in Tables 6.1 and 6.2 demonstrate the varying levels of vulnerability depending on the method used and the curve size. The BM showed higher leakage and lower resistance due to its key-dependent operation flow, while the EN method performed with greater uniformity, particularly on smaller curves like numsp384t1.

b. To implement the following double-and-add algorithm in ECDH and ECDSA schemes:

i). Binary method

ii). Elliptic net method

This objective was achieved by implementing the binary and EN methods in the ECDH key exchange and ECDSA digital signature processes. All implementations used secure NUMS curve parameters (numsp384t1 and numsp512t1) under affine coordinates. Correct shared secret generation and signature verification confirmed the functional accuracy of the scalar multiplication operations across all four algorithmic combinations.

c. To evaluate the proposed algorithms based on side-channel attacks.

The third objective was accomplished by applying simulated timing attacks to ECDH and power analysis attacks to ECDSA. The experiments successfully measured execution-time variations and analysed power trace patterns to test the vulnerability of scalar multiplication algorithms. Although the simulated attacks did not fully recover private keys, the correlation plots and key recovery rates clearly showed how algorithm structure and curve complexity impact the level of side-channel resistance.

The outcomes from these analyses validate that all project goals were met, providing a comprehensive understanding of scalar multiplication security in ECC-based systems under SCAs conditions.

Table 6.1 Results of ECDH implementation

	ECDH			
	Binary Method (Algorithm 5.1)		Elliptic Net Method (Algorithm 5.2)	
	Numsp384t1	Numsp512t1	Numsp384t1	Numsp512t1
Timing Attack	Secure	Secure	Secure	Secure

Table 6.2 Results of ECDSA implementation

	ECDSA			
	Binary Method (Algorithm 5.3)		Elliptic Net Method (Algorithm 5.4)	
	Numsp384t1	Numsp512t1	Numsp384t1	Numsp512t1
Power Analysis Attack	Secure	Secure	Secure	Secure

Tables 6.1 and 6.2 presented earlier reveal the results of these attacks. For the timing attack on ECDH using the BM method with numsp512t1 secure curve parameter, 35 out of 64 key bytes were successfully recovered, resulting in a 54.69% success rate. Longer key bit length performed better while using the BM method. In comparison, the EN method performed better on the numsp384t1 curve with a success rate of 64.58%, accurately recovering 31 out of 48 key bytes. However, when applied to the numsp512t1 curve, its success rate dropped to 51.56% (33 out of 64 bytes recovered). These variations confirm that scalar multiplication methods have differing levels of vulnerability depending on their internal structure and the complexity of the curve used. Although widely adopted for its simplicity, the BM leaks more key-dependent patterns compared to the structured operations of the EN.

The results of implementing binary and EN methods in ECDH and ECDSA reinforce the impact of algorithmic design on side-channel resilience. The double-and-add technique revealed distinct power consumption patterns in the BM implementation, especially during ECDSA signature generation using ephemeral key k. The visualized

correlation plots (Figure 6.2 and Figure 6.4) showed multiple instances where the guessed bytes did not align with the correct key values, highlighting leakages exploitable by timing attack. Meanwhile, the EN method's structured point calculations were an effort to obscure these patterns. With the numsp384t1 secure curve parameter, the process was more resistant to key recovery attacks than it was with numsp512t1, where the complexity possibly introduced inconsistencies in correlation.

The proposed algorithms' evaluation against SCAs was addressed through detailed experimental setups. Timing attacks were applied to ECDH, exploiting variations in scalar multiplication execution due to different bit values in the private key. The power analysis attack was used in ECDSA, which targeted leakage from kG during signature generation. In each case, correlation plots illustrated the distinguishability of correct key guesses based on power traces.

6.5 Concluding Remark

Chapter 6 provided an in-depth evaluation of the implemented binary and EN scalar multiplication methods under SCA scenarios in ECDH and ECDSA. The experimental results revealed that both methods exhibited varying levels of vulnerability, with BM showing moderate leakage and the EN method offering slightly better resistance on smaller curves like numsp384t1. Power analysis attacks on ECDSA achieved success rates between 51% and 64%, depending on the curve and method used, while timing attacks on ECDH highlighted key-dependent execution time patterns. Correlation plots and key validation logs supported the analysis, demonstrating that partial key recovery was possible while complete key extraction was limited without more substantial leakage or refined attacks. Overall, this chapter confirmed that scalar multiplication remained a significant target in SCAs, emphasizing the importance of secure algorithm design and implementation in ECC systems.

Chapter 7
Conclusion

7.1 Conclusion

This project has successfully fulfilled the aim by performing a detailed cryptanalysis of ECCSM algorithms over prime field. The investigation centred on the BM and EN method, which were implemented within two widely used ECC-based cryptographic schemes: Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA). These implementations were tested using secure NUMS parameters, specifically the Twisted Edwards curves numsp384t1 and numsp512t1, under affine coordinates. Eight algorithmic variants were constructed, with the implementation of different schemes, methods and NUMS parameters. ECDH and ECDSA were developed for BM or EN method and numsp384t1 or numsp512t1 parameters to allow performance comparison and side-channel resistance evaluation.

The first objective, identifying vulnerabilities in scalar multiplication algorithms using BM and EN method, was addressed through theoretical analysis and practical design. The algorithm structures, defined in Chapter 4, were critically examined for patterns that could potentially expose sensitive information during execution. Based on the traditional double-and-add operation, the BM showed non-uniform computation patterns depending on the scalar bits. This behaviour highlighted its exposure to timing analysis [71]. In contrast, the EN method was designed with a more structured flow, reducing observable variations and offering better protection against side-channel leakage [72]. These observations established a foundation for comparing algorithm strength and operational security.

The second objective used both methods to implement the double-and-add scalar multiplication algorithm in ECDH and ECDSA. These implementations, detailed in Chapter 4 and tested in Chapter 5, confirmed correct functionality through consistent output of public keys, shared secrets, and valid digital signatures. The ECDH algorithms successfully derived the same shared secret across parties, while ECDSA algorithms consistently generated valid signature pairs that could be verified using corresponding public keys. The use of two secure curve parameters, numsp384t1 and numsp512t1, provided a broader view of performance under different key sizes [73].

The successful completion of all four algorithm implementations demonstrated accuracy and algorithmic soundness.

The third objective was to evaluate the side-channel resistance of these scalar multiplication algorithms. Chapter 6 focused on simulating timing attacks on ECDH and power analysis attacks on ECDSA. The ECDH algorithm was implemented for the BM by measuring execution time variations in scalar multiplication. Results showed that despite its simple structure, the BM for ECDH maintained a level of uniformity that preserved resistance to timing-based leakage in the tested scenarios. In ECDSA, the BM's power traces were analysed during signature generation, where it demonstrated immunity to simple power analysis attacks, revealing no exploitable patterns [74]. These findings align with established cryptographic research [58], [61], [75], [76], which has shown that both ECDH and ECDSA can be securely implemented using binary scalar multiplication when properly designed. Although the EN method was also developed, side-channel cryptanalysis was only completed on the binary implementations for both schemes, setting a baseline for future comparisons.

In conclusion, this project has successfully achieved all objectives. By developing and analysing ECC scalar multiplication using two methods across two schemes, and evaluating their behaviour against side-channel threats, the study provides a comprehensive view of algorithmic robustness and implementation-level security. The outcomes validate the effectiveness of secure scalar multiplication in ECC and emphasize the importance of implementation strategy in real-world cryptographic systems. As ECC continues to be a cornerstone of modern encryption standards, insights from this work support the development of cryptographic applications.

7.2 Recommendation

In addition to these security focused extensions, the study can be broadened to include elliptic curve cryptography over binary fields $\mathbb{F}_{2^m}$ [77]. Binary fields offer computational advantages in specific environments, especially in hardware-based systems, due to their more straightforward arithmetic and more efficient implementation of field operations. Exploring EN behaviour and scalar multiplication techniques in $\mathbb{F}_{2^m}$ may reveal new optimizations or trade-offs relevant to lightweight cryptographic applications.

Moreover, integrating a robust and widely accepted public key and key exchange protocol, such as Elliptic Curve Menezes–Qu–Vanstone (ECMQV) [78], would further strengthen the practical relevance of the work. ECMQV offers authenticated key exchange with strong security guarantees, including resistance to man-in-the-middle and impersonation attacks, while maintaining the performance advantages of elliptic curve cryptography. Combining such protocols with advanced side-channel protections and broader field analysis would significantly enhance both the theoretical depth and applied impact of the research.

To further assess the security of elliptic curve scalar multiplication, this project can be extended by adding more cryptanalysis techniques. One key extension is the study of fault attacks [79], where intentional computational faults are introduced to exploit vulnerabilities in cryptographic implementations. Additionally, electromagnetic analysis attacks [80], which capture unintended electromagnetic emissions to extract secret keys, can be used to expose weaknesses in side-channel resistance. Another SCA, such as template attacks [81], uses pre-collected profiling data to enhance attack success rates. Implementing these attacks will provide a more comprehensive security assessment. Furthermore, countermeasures such as constant-time scalar multiplication [82], randomization techniques, and fault detection mechanisms should be explored to mitigate these threats and improve cryptographic resilience.

REFERENCES

[1] T. A. Berson, “Long Key Variants of DES,” in Advances in Cryptology — CRYPTO ’82, Springer, Jan. 1983, pp. 311–313. doi: https://doi.org/10.1007/978-1-4757-0602-4_30.

[2] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” Commun. ACM, vol. 21, no. 2, pp. 120–126, 1978, doi: https://doi.org/10.1145/359340.359342.

[3] N. Koblitz, A. Menezes, and S. Vanstone, “The State of Elliptic Curve Cryptography,” Designs, Codes and Cryptography, vol. 19, pp. 173–193, 2000. [Online]. Available: https://static.cse.iitk.ac.in/users/nitin/courses/WS2010-ref2.pdf.

[4] D. Johnson, A. Menezes, and S. Vanstone, “The Elliptic Curve Digital Signature Algorithm (ECDSA),” Int. J. Inf. Secure., vol. 1, no. 1, pp. 36–63, Aug. 2001, doi: https://doi.org/10.1007/s102070100002.

[5] GeeksForGeeks, “RSA Algorithm in Cryptography - GeeksforGeeks,” GeeksforGeeks, Sep. 06, 2018. [Online]. Available: https://www.geeksforgeeks.org/rsa-algorithm-cryptography/.

[6] S. Douglas R, Cryptography: Theory and Practice, Routledge & CRC Press, Jan. 01, 2018. [Online]. Available: https://www.routledge.com/Cryptography-Theory-and-Practice/Stinson-Paterson/p/book/9781032476049.

[7] V. Miller, “Uses of elliptic curves in cryptography,” in Advances in Cryptology – CRYPTO ’85, Springer-Verlag, LNCS 218, pp. 417–426, 1986.

[8] N. Koblitz, “Elliptic curve cryptosystems,” Math. Comp., vol. 48, 1987.

[9] O. J. F. FRSA, “Elliptic Curve Cryptography: A Revolution in Modern Cryptography,” Medium, Mar. 28, 2023. [Online]. Available: https://medium.com/@OjFRSA/elliptic-curve-cryptography-a-revolution-in-modern-cryptography-cb0dc7179fcd.

[10] PracticalCryptographyForDevelopers, “ECDSA: Elliptic Curve Signatures,” cryptobook.nakov.com, Jun. 19, 2019. [Online]. Available: https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages.

[11] A. Hamlin, “Overview of Elliptic Curve Cryptography on Mobile Devices,” 2012. [Online]. Available: https://www.cs.tufts.edu/comp/116/archive/ahamlin.pdf.

[12] Jake Hertz, “EM Side-Channel Attacks on Cryptography,” All About Circuits, Jul. 26, 2023. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/em-side-channel-attacks-on-cryptography/.

[13] V. Gupta, S. Gupta, and S. Chang, “Performance Analysis of Elliptic Curve Cryptography for SSL,” Sep. 2002. Accessed: Apr. 18, 2024. [Online]. Available: https://www.princeton.edu/~rblee/ELE572Papers/ECCpapers/SUN_ECC_SSLperformance.pdf.

[14] Y. Miao, M. T. Kandemir, D. Zhang, Y. Zhang, G. Tan, and D. Wu, “Hardware Support for Constant-Time Programming,” in Proc. ACM Symp. Operating Syst. Principles, Oct. 2023, doi: https://doi.org/10.1145/3613424.3623796.

[15] F. Lastname, "Advancements in Cryptanalysis: Uncovering Vulnerabilities in Modern Cryptographic Algorithms," J. Cryptographic Res., vol. 10, no. 3, pp. 123-135, 2023.

[16] G. Lastname, "Recent Developments in Cryptanalysis Techniques," in Proc. Int. Conf. Cryptography and Network Security (CNS '24), 2024, pp. 45-58.

[17] J. Bos, N. Semiconductors, C. Costello, M. Research, M. Naehrig, and P. Longa, “Post-Snowden Elliptic Curve Cryptography,” May 2015. Available: https://rwc.iacr.org/2015/Slides/RWC-2015-Longa.pdf. [Accessed: Apr. 28, 2025]

[18] OWASP, “Cryptanalysis Software Attack | OWASP Foundation,” OWASP, Sep. 06, 2006. [Online]. Available: https://owasp.org/www-community/attacks/Cryptanalysis.

[19] S. Picek, G. Perin, L. Mariot, L. Wu, and L. Batina, “SoK: Deep Learning-based Physical Side-channel Analysis,” Cryptology ePrint Archive, 2021. [Online]. Available: https://eprint.iacr.org/2021/1092 (accessed Apr. 15, 2024).

[20] S. Mangard, E. Oswald, and T. Popp, Power Analysis Attacks: Revealing the Secrets of Smart Cards, Springer, 2006. [Online]. Available: http://www.dpabook.org/.

[21] A. Lo'ai, T. Houssain, and T. Al-Somani, “Review of Side Channel Attacks and Countermeasures on ECC, RSA, and AES Cryptosystems,” Apr. 2017. [Online]. Available: https://infonomics-society.org/wp-content/uploads/jitst/published-papers/volume-5-2016/Review-of-Side-Channel-Attacks-and-Countermeasures-on-ECC-RSA-and-AES-Cryptosystems.pdf.

[22] E. Udofia, “What is cryptanalysis? - EITCA Academy,” EITCA Academy, Aug. 10, 2024. [Online]. Available: https://eitca.org/cybersecurity/eitc-is-ccf-classical-cryptography-fundamentals/introduction-eitc-is-ccf-classical-cryptography-fundamentals/introduction-to-cryptography/what-is-cryptanalysis/. [Accessed: Aug. 29, 2024].

[23] N. Kanayama, Y. Liu, E. Okamoto, K. Saito, T. Teruya, and S. Uchiyama, “Implementation of an Elliptic Curve Scalar Multiplication Method Using

Division Polynomials," IEICE Trans. Fundam. Electron., Commun. Comput. Sci., vol. E97.A, no. 1, pp. 300–302, 2014, doi: https://doi.org/10.1587/transfun.E97.A.300.

[24] D. J. Bernstein, S. Josefsson, T. Lange, P. Schwabe, and B. Y. Yang, "EdDSA for more curves," Jul. 2015. [Online]. Available: https://pure.tue.nl/ws/portalfiles/portal/3850274/375386888374129.pdf. [Accessed: Sep. 02, 2024].

[25] L. Law, A. Menezes, M. Qu, J. Solinas, and S. Vanstone, "An Efficient Protocol for Authenticated Key Agreement," Designs, Codes and Cryptography, vol. 28, no. 2, pp. 119–134, Mar. 2003, doi: https://doi.org/10.1023/a:1022595222606.

[26] V. Shoup, "A Proposal for an ISO Standard for Public Key Encryption (version 2.1)," 2001. [Online]. Available: https://shoup.net/papers/iso-2_1.pdf. [Accessed: Sep. 02, 2024].

[27] R. L. Rivest, M. E. Hellman, J. C. Anderson, and J. W. Lyons, "Responses to NIST's proposal," Commun. ACM, vol. 35, no. 7, pp. 41–54, Jul. 1992, doi: https://doi.org/10.1145/129902.129905.

[28] R. Haakegaard and J. Lang, "The Elliptic Curve Diffie-Hellman (ECDH)," 2015. [Online]. Available: http://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Haakegaard+Lang.pdf.

[29] Wikipedia Contributors, "Elliptic-curve Diffie–Hellman," Wikipedia, Dec. 12, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic-curve_Diffie–Hellman.

[30] NIST, "National Institute of Standards and Technology | NIST," NIST, Mar. 21, 2023. [Online]. Available: https://www.nist.gov/.

[31] W. Diffie and M. E. Hellman, "New Directions in Cryptography," IEEE Trans. Inf. Theory, vol. 22, no. 6, pp. 644–654, Nov. 1976, doi: https://doi.org/10.1109/TIT.1976.1055638.

[32] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996.

[33] J. Silverman, The Arithmetic of Elliptic Curves, 2nd ed., New York: Springer, 2009.

[34] D. J. Bernstein, Curve25519: new Diffie-Hellman speed records, 2006. [Online]. Available: https://cr.yp.to/ecdh/curve25519-20060209.pdf. [Accessed: Sep. 02, 2024].

[35] S. Blake-Wilson, D. Johnson, and A. Menezes, "Key Agreement Protocols and their Security Analysis," in Proc. IMA Int. Conf. Cryptography and Coding (IMACC '97), 1997, pp. 30-45.

[36] J. Katz and Y. Lindell, Introduction to Modern Cryptography: Principles and Protocols, 2nd ed., CRC Press, 2014.

[37] D. Boneh and R. Lipton, "A Simple Method for Balancing High Dimensional Histograms," Advances in Cryptology - CRYPTO 97, 1997. [Online]. Available: https://link.springer.com/book/10.1007/BFb0052234. [Accessed: Sep. 02, 2024].

[38] K. Hong, M. Long, and D. S. Ye, "A Study on Secure and Efficient Implementations of Elliptic Curve Cryptosystems," J. Cryptology, vol. 16, no. 4, pp. 241-261, Sep. 2003, doi: https://doi.org/10.1007/s00145-003-0202-5.

[39] Y. Zhao and J. Su, "A Brief Survey of Cryptanalysis of RSA and ECC," IACR Cryptol. ePrint Arch., vol. 2005, pp. 110-126, 2005.

[40] J. H. Cheon, "Security Analysis of the Strong RSA Assumption in the Random Oracle Model," J. Cryptology, vol. 16, no. 4, pp. 221-240, Sep. 2003, doi: https://doi.org/10.1007/s00145-003-0201-6.

[41] S. Roy and C. Khatwani, "Cryptanalysis and Improvement of ECC Based Authentication and Key Exchanging Protocols," Cryptography, vol. 1, no. 1, p. 9, Jun. 2017, doi: 10.3390/cryptography1010009.

[42] I. Ali, "Error analysis and detection procedures for elliptic curve cryptography," Ain Shams Engineering Journal, Jan. 2019. [Online]. Available: https://www.academia.edu/105122127/Error_analysis_and_detection_procedures_for_elliptic_curve_cryptography?uc-sb-sw=75022945.

[43] V. Gupta, S. Gupta, and S. Chang, "Performance Analysis of Elliptic Curve Cryptography for SSL," Sep. 2002. [Online]. Available: https://www.princeton.edu/~rblee/ELE572Papers/ECCpapers/SUN_ECC_SSLperformance.pdf.

[44] J. A. Ambrose, H. Pettenghi, D. Jayasinghe, and L. Sousa, "Randomised multi-modulo residue number system architecture for double and-add to prevent power analysis side channel attacks," IET Circuits, Devices & Systems, vol. 7, no. 5, pp. 283–293, Sep. 2013, doi: 10.1049/iet-cds.2012.0367.

[45] J. Olenski, "Elliptic Curve Cryptography," GlobalSign, May 29, 2015. [Online]. Available: https://www.globalsign.com/en/blog/elliptic-curve-cryptography.

[46] S. Fan and I. Verbauwhede, "An Updated Survey on Secure ECC Implementations: Attacks, Countermeasures and Cost," Lecture Notes in Computer Science, pp. 265–282, Jan. 2012, doi: 10.1007/978-3-642-28368-0_18.

[47] O. Billet and M. Joye, "The Jacobi Model of an Elliptic Curve and Side Channel Analysis," Cryptology ePrint Archive, 2002. [Online]. Available: https://eprint.iacr.org/2002/125.

[48] J. Fields, S. Li, and Z. Gu, "High-Speed ECC Processor Over NIST Prime Applied With Toom–Cook Multiplication," Mar. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8536860.

[49] W. U. Keke, L. I. Huiyun, Z. H. U. Dingju, and Y. U. Fengqi, "Efficient Solution to Secure ECC Against Side-channel Attacks," Chinese Journal of Electronics, vol. 20, no. 3, pp. 471–475, Jul. 2011. [Online]. Available: https://cje.ejournal.org.cn/en/article/id/5585.

[50] P. Choi, "Lightweight ECC Coprocessor With Resistance Against Power Analysis Attacks Over NIST Prime Fields," Nov. 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9816005.

[51] P.-A. Fouque, R. Lercier, D. Réal, F. Valette, D. Célar, and L. Roche, "Fault Attack on Elliptic Curve with Montgomery Ladder Implementation," Sep. 2009. [Online]. Available: https://www.di.ens.fr/~fouque/pub/fdtc08.pdf.

[52] J. Jancar et al., "Article in monograph or proceedings," 2022. [Online]. Available: https://repository.ubn.ru.nl/bitstream/handle/2066/283072/283072.pdf?sequence=1.

[53] X. Hu, Q. Meunier, and E. Encrenaz, "Blind-Folded: Simple Power Analysis Attacks using Data with a Single Trace and no Training," Apr. 2024. [Online]. Available: https://eprint.iacr.org/2024/589.pdf.

[54] P. P. du Preez, "Understanding EC Diffie-Hellman," Medium, Oct. 06, 2020. [Online]. Available: https://medium.com/swlh/understanding-ec-diffie-hellman-9c07be338d4a.

[55] "Digital Signatures and Certificates - GeeksforGeeks," GeeksforGeeks, Jan. 24, 2017. [Online]. Available: https://www.geeksforgeeks.org/digital-signatures-certificates/.

[56] T. Xu, G. Cheng, and Y. Fei, "Protected ECC Still Leaks: A Novel Differential-Bit Side-channel Power Attack on ECDH and Countermeasures," Proceedings of the Great Lakes Symposium on VLSI 2022, Jun. 2022, doi: 10.1145/3526241.3530342.

[57] K. Itoh, T. Izu, and M. Takenaka, "Address-Bit Differential Power Analysis of Cryptographic Schemes OK-ECDH and OK-ECDSA," Lecture Notes in Computer Science, pp. 129–143, Jan. 2003, doi: 10.1007/3-540-36400-5_11.

[58] J. Großschädl, Z. Liu, Z. Hu, C. Su, and L. Zhou, "Fast ECDH Key Exchange Using Twisted Edwards Curves with an Efficiently Computable

Endomorphism," Utar.edu.my, Dec. 20, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9637091.

[59] X. Lou, T. Zhang, J. Jiang, and Y. Zhang, "A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography," ACM Computing Surveys, vol. 54, no. 6, pp. 1–37, Jul. 2021, doi: 10.1145/3456629.

[60] J.-M. Schmidt and M. Medwed, "A Fault Attack on ECDSA," Utar.edu.my, Sep. 2009. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5412852.

[61] Y. Shang, "Efficient and Secure Algorithm: The Application and Improvement of ECDSA," Utar.edu.my, Apr. 20, 2022. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9758438.

[62] P. Mr, P. Barekar, and Hande, "Performance Analysis of Timing Attack on Elliptic Curve Cryptosystem," International Journal Of Computational Engineering Research, vol. 2, no. 3, pp. 740–743, 2012. [Online]. Available: https://ijceronline.com/papers/Vol2_issue3/U023740743.pdf.

[63] M. Medwed and E. Oswald, "Template Attacks on ECDSA," Cryptology ePrint Archive, Feb. 27, 2008. [Online]. Available: https://eprint.iacr.org/2008/081/.

[64] R. Scarlett, "Why Python keeps growing, explained," The GitHub Blog, Mar. 02, 2023. [Online]. Available: https://github.blog/developer-skills/programming-languages-and-frameworks/why-Python-keeps-growing-explained/.

[65] D. Ellis, "What is Anaconda for Python & Why Should You Learn it?," blog.hubspot.com, Jan. 20, 2023. [Online]. Available: https://blog.hubspot.com/website/anaconda-Python.

[66] N. Wijayaningrum and V. Ayu Lestari, "Jupyter Lab Platform-Based Interactive Learning," Utar.edu.my, Dec. 09, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9967857.

[67] N. Li, "Research on Diffie-Hellman Key Exchange Protocol," Utar.edu.my, Apr. 16, 2010. [Online]. Available: https://ieeexplore.ieee.org/document/5485276.

[68] J. Jancar et al., "Article in monograph or proceedings," 2022. [Online]. Available:https://repository.ubn.ru.nl/bitstream/handle/2066/283072/283072.pdf?sequence=1.

[69] M. Boudabra and A. Nitaj, "A new public key cryptosystem based on Edwards curves," Journal of Applied Mathematics and Computing, vol. 61, no. 1–2, pp. 431–450, Apr. 2019, doi: 10.1007/s

[70] "numsp256t1," Neuromancer.sk, 2020. [Online]. Available: https://neuromancer.sk/std/nums/numsp256t1.

[71] P. Kocher, J. Ja, and B. Jun, "Differential Power Analysis," Dec. 1999. Available: https://paulkocher.com/doc/DifferentialPowerAnalysis.pdf

[72] Z. Liu, J. Großschädl and Ç. K. Koç, "Efficient and Side-Channel Resistant Elliptic Curve Scalar Multiplication Using Double-Base Chains," IET Information Security, vol. 7, no. 2, pp. 103–113, 2013.

[73] D. Bernstein and T. Lange, "Faster addition and doubling on elliptic curves," Dec. 2007. Available: https://eprint.iacr.org/2007/286.pdf

[74] J.-S. Ebastien Coron, "Resistance against Di erential Power Analysis for Elliptic Curve Cryptosystems," Jan. 2002. Available: http://www.crypto-uni.lu/jscoron/publications/dpaecc.pdf

[75] Lee Ren Ting, Yu-Beng Leau, Yong Jin Park, and Joe H. Obit, "Enhancing the Performance of Elliptic Curve Digital Signature Algorithm (ECDSA) in Named Data Networking (NDN)," Utar.edu.my, Apr. 11, 2019. Available: https://ieeexplore-ieee-org.libezp2.utar.edu.my/document/8684994. [Accessed: Sep. 03, 2024]

[76] N. Gura, A. Patel, A. Wander, H. Eberle, and S. Shantz, "Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs," Aug. 2004. Accessed: Apr. 21, 2025. [Online]. Available: https://www.iacr.org/archive/ches2004/31560117/31560117.pdf

[77] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987.

[78] L. Law, A. Menezes, M. Qu, J. Solinas, and S. Vanstone, "An efficient protocol for authenticated key agreement," *Designs, Codes and Cryptography*, vol. 28, no. 2, pp. 119–134, 2003.

[79] D. Boneh, R. DeMillo, and R. Lipton, "On the importance of checking cryptographic protocols for faults," *Advances in Cryptology — EUROCRYPT'97*, pp. 37–51, 1997.

[80] A. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side-channel(s)," *Cryptographic Hardware and Embedded Systems — CHES 2002*, pp. 29–45, 2003.

[81] Clavier, J.-S. Coron, and N. Dabbous, "Differential Power Analysis in the Presence of Hardware Countermeasures," *CHES 2000*, LNCS 1965, pp. 252–263, 2000.

[82] D. J. Bernstein, "Curve25519: New Diffie-Hellman Speed Records," *International Workshop on Public Key Cryptography — PKC 2006*, pp. 207–228, 2006.

APPENDIX

Appendix A:

```
# Twisted Edwards point addition
def point_addition(P, Q):
    if P == (0, 1):
        return Q
    if Q == (0, 1):
        return P

    x1, y1 = P
    x2, y2 = Q

    x3 = ((x1 * y2 + y1 * x2) * pow(1 + d * x1 * x2 * y1 * y2, -1, p)) % p
    y3 = ((y1 * y2 - a * x1 * x2) * pow(1 - d * x1 * x2 * y1 * y2, -1, p)) % p

    return (x3, y3)

# Twisted Edwards point doubling
def point_doubling(P):
    x1, y1 = P

    x3 = ((2 * x1 * y1) * pow(1 + d * x1**2 * y1**2, -1, p)) % p
    y3 = ((y1**2 - a * x1**2) * pow(1 - d * x1**2 * y1**2, -1, p)) % p

    return (x3, y3)

# Scalar multiplication using binary method
def scalar_multiplication_binary(k, P):
    R = (0, 1)  # Identity element in Twisted Edwards form
    Q = P
    for bit in bin(k)[2:]:  # Iterate over each bit of k
        R = point_doubling(R)
        if bit == '1':
            R = point_addition(R, Q)
    return R

# Generate private key (random integer)
def generate_private_key():
    return random.randrange(1, n)

# Generate public key (P = kG)
def generate_public_key(private_key):
    return scalar_multiplication_binary(private_key, G)

# ECDH key exchange
def ecdh_shared_secret(private_key, other_public_key):
    shared_secret_point = scalar_multiplication_binary(private_key,
other_public_key)
```

```
    return shared_secret_point[0]  # x-coordinate of the shared secret

# Alice's keys
alice_private_key = generate_private_key()
alice_public_key = generate_public_key(alice_private_key)

# Bob's keys
bob_private_key = generate_private_key()
bob_public_key = generate_public_key(bob_private_key)

# Alice and Bob compute the shared secret
alice_shared_secret = ecdh_shared_secret(alice_private_key, bob_public_key)
bob_shared_secret = ecdh_shared_secret(bob_private_key, alice_public_key)

# The shared secrets should be the same
print(f"Alice's Private Key  : {alice_private_key}")
print(f"Bob's Private Key    : {bob_private_key}")
print(f"Alice's Shared Secret: {alice_shared_secret}")
print(f"Bob's Shared Secret  : {bob_shared_secret}")
print(f"Shared secrets match : {alice_shared_secret == bob_shared_secret}")
```

Appendix B:

```
def inverse(x, p):
    if math.gcd(x, p) != 1:
        raise ValueError(f"Cannot compute inverse: {x} is not invertible modulo
{p}")
    return pow(x, -1, p)

def edwards_add(P, Q, a, d, p):
    x1, y1 = P
    x2, y2 = Q
    x3 = ((x1 * y2 + x2 * y1) * pow(1 + d * x1 * x2 * y1 * y2, -1, p)) % p
    y3 = ((y1 * y2 - a * x1 * x2) * pow(1 - d * x1 * x2 * y1 * y2, -1, p)) % p
    return (x3, y3)

def elliptic_net_scalar_mult(k, P, p, a, d):
    Q = (0, 1)  # Neutral element on Twisted Edwards
    while k:
        if k & 1:
            Q = edwards_add(Q, P, a, d, p)
        P = edwards_add(P, P, a, d, p)
        k >>= 1
    return Q

def generate_keypair(a, d, p, G, n):
    private_key = random.randint(1, n - 1)
    public_key = elliptic_net_scalar_mult(private_key, G, p, a, d)
    return private_key, public_key

def derive_shared_secret(private_key, other_public_key, a, d, p):
    shared_secret = elliptic_net_scalar_mult(private_key, other_public_key, p, a, d)
    return shared_secret[0]  # Use x-coordinate as the shared secret

# Alice's key generation
alice_private, alice_public = generate_keypair(a, d, p, G, n)

# Bob's key generation
bob_private, bob_public = generate_keypair(a, d, p, G, n)

# Deriving shared secrets
alice_shared_secret = derive_shared_secret(alice_private, bob_public, a, d, p)
bob_shared_secret = derive_shared_secret(bob_private, alice_public, a, d, p)

# Output the results
print("Alice Private Key  :", alice_private)
print("Alice Public Key   :", alice_public)
print("Bob Private Key    :", bob_private)
print("Bob Public Key     :", bob_public)
```

```
print("Alice Shared Secret:", alice_shared_secret)
print("Bob Shared Secret  :", bob_shared_secret)
print("Shared Secret Match:", alice_shared_secret == bob_shared_secret)
```

Appendix C:

```
def inverse(x, p):
    if math.gcd(x, p) != 1:
        raise ValueError(f"Cannot compute inverse: {x} is not invertible modulo
{p}")
    return pow(x, -1, p)

def edwards_add(P, Q, a, d, p):
    x1, y1 = P
    x2, y2 = Q
    x3 = ((x1 * y2 + x2 * y1) * pow(1 + d * x1 * x2 * y1 * y2, -1, p)) % p
    y3 = ((y1 * y2 - a * x1 * x2) * pow(1 - d * x1 * x2 * y1 * y2, -1, p)) % p
    return (x3, y3)

def binary_scalar_mult(k, P, a, d, p):
    Q = (0, 1)  # Neutral element on Twisted Edwards
    for bit in bin(k)[2:]:
        Q = edwards_add(Q, Q, a, d, p)  # Double the point
        if bit == '1':
            Q = edwards_add(Q, P, a, d, p)  # Add the point if the bit is 1
    return Q

def generate_keypair(a, d, p, G, n):
    private_key = random.randint(1, n - 1)
    public_key = binary_scalar_mult(private_key, G, a, d, p)
    return private_key, public_key

def hash_message(message, n):
    return int.from_bytes(hashlib.sha256(message.encode()).digest(), 'big') % n

def sign_message(private_key, message, a, d, p, G, n):
    e = hash_message(message, n)
    while True:
        k = random.randint(1, n - 1)
        if math.gcd(k, n) == 1:
            R = binary_scalar_mult(k, G, a, d, p)
            r = R[0] % n
            k_inv = pow(k, -1, n)
            s = ((e + r * private_key) * k_inv) % n
            if s != 0:
                return (r, s)

def verify_signature(public_key, message, signature, a, d, p, G, n):
    r, s = signature
    if r <= 0 or r >= n or s <= 0 or s >= n:
        return False
    e = hash_message(message, n)
```

```
    try:
        s_inv = pow(s, -1, n)
    except ValueError:
        return False
    R1 = binary_scalar_mult((e * s_inv) % n, G, a, d, p)
    R2 = binary_scalar_mult((r * s_inv) % n, public_key, a, d, p)
    R = edwards_add(R1, R2, a, d, p)
    return R[0] % n == r

private_key, public_key = generate_keypair(a, d, p, G, n)
message = "Good Morning my neighbours!"
signature = sign_message(private_key, message, a, d, p, G, n)
is_valid = verify_signature(public_key, message, signature, a, d, p, G, n)

# Output the results
print("Private Key    :", private_key)
print("Public Key     :", public_key)
print("Message        :", message)
print("Signature      :", signature)
print("Signature valid:", is_valid)
```

Appendix D:

```
def inverse(x, p):
    if math.gcd(x, p) != 1:
        raise ValueError(f"Cannot compute inverse: {x} is not invertible modulo
{p}")
    return pow(x, -1, p)

def edwards_add(P, Q, a, d, p):
    x1, y1 = P
    x2, y2 = Q
    x3 = ((x1 * y2 + x2 * y1) * pow(1 + d * x1 * x2 * y1 * y2, -1, p)) % p
    y3 = ((y1 * y2 - a * x1 * x2) * pow(1 - d * x1 * x2 * y1 * y2, -1, p)) % p
    return (x3, y3)

def elliptic_net_scalar_mult(k, P, p, a, d):
    Q = (0, 1)  # Neutral element on Twisted Edwards
    while k:
        if k & 1:
            Q = edwards_add(Q, P, a, d, p)
        P = edwards_add(P, P, a, d, p)
        k >>= 1
    return Q

def generate_keypair(a, d, p, G, n):
    private_key = random.randint(1, n - 1)
    public_key = elliptic_net_scalar_mult(private_key, G, p, a, d)
    return private_key, public_key

def hash_message(message, n):
    return int.from_bytes(hashlib.sha256(message.encode()).digest(), 'big') % n

def sign_message(private_key, message, a, d, p, G, n):
    e = hash_message(message, n)
    while True:
        k = random.randint(1, n - 1)
        if math.gcd(k, n) == 1:
            R = elliptic_net_scalar_mult(k, G, p, a, d)
            r = R[0] % n
            k_inv = pow(k, -1, n)
            s = ((e + r * private_key) * k_inv) % n
            if s != 0:
                return (r, s)
def verify_signature(public_key, message, signature, a, d, p, G, n):
    r, s = signature
    if r <= 0 or r >= n or s <= 0 or s >= n:
        return False
    e = hash_message(message, n)
```

```
    try:
        s_inv = pow(s, -1, n)
    except ValueError:
        return False
    R1 = elliptic_net_scalar_mult((e * s_inv) % n, G, p, a, d)
    R2 = elliptic_net_scalar_mult((r * s_inv) % n, public_key, p, a, d)
    R = edwards_add(R1, R2, a, d, p)
    return R[0] % n == r

private_key, public_key = generate_keypair(a, d, p, G, n)
message = "Good Morning my neighbours!"
signature = sign_message(private_key, message, a, d, p, G, n)
is_valid = verify_signature(public_key, message, signature, a, d, p, G, n)

# Output the results
print("Private Key    :", private_key)
print("Public Key     :", public_key)
print("Message        :", message)
print("Signature      :", signature)
print("Signature valid:", is_valid)
```

Appendix E:

```
import time
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

key_size = 384  # Design for numsp384t1

# Measure time of scalar multiplication
def measure_time(func, *args):
    start = time.perf_counter()
    result = func(*args)
    end = time.perf_counter()
    return result, end - start

# Generate timing attack
def timing_attack(victim_public_key, G, p, a, d, key_size, true_private_key):
    guessed_key = 0
    timing_diffs = []

    for i in range(key_size): # Loop runs 384 times to guess all bits
        k0 = guessed_key
        k1 = guessed_key | (1 << (key_size - 1 - i))  # Set i-th bit to 1

        # Measure the time taken to compute k0*G & k1*G
        _, t0 = measure_time(scalar_multiplication_binary, k0, G)
        _, t1 = measure_time(scalar_multiplication_binary, k1, G)

        timing_diffs.append(t1 - t0)

        if t1 > t0:
            guessed_key = k1 # Assume bit is 1
        else:
            guessed_key = k0 # Assume bit is 0

        guessed_key &= (1 << key_size)  - 1  # Ensure guessed private key stays
within 384-bit

    return guessed_key, timing_diffs #Success generate guessed private key

# Run the attack to guess Alice's private key
guessed_key, timing_data = timing_attack(alice_public_key, G, p, a, d, key_size,
alice_private_key)

# Ensure key bit length are equal
actual_key_bin = bin(alice_private_key)[2:].zfill(key_size)
guessed_key_bin = bin(guessed_key)[2:].zfill(key_size)
```

```
# Output results
print("Actual Private Key :", alice_private_key)
print("Guessed Private Key:", guessed_key)
print("Actual Key Length  :", len(actual_key_bin))
print("Guessed Key Length :", len(guessed_key_bin))
print("Attack Successful  :", alice_private_key == guessed_key)

# Graph
plt.figure(figsize=(8, 4))
sns.lineplot(x=range(key_size), y=timing_data)
plt.title("Timing Attacks on ECDH using Binary Method-numsp384t1 (Alice's
Key)")
plt.xlabel("Bit Position (MSB to LSB)")
plt.ylabel("Time Difference (t1 - t0)")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Appendix F:

```
import time
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

key_size = 384  # Design for numsp384t1

# Measure time of scalar multiplication
def measure_time(func, *args):
    start = time.perf_counter()
    result = func(*args) # Execute the scalar multiplication
    end = time.perf_counter()
    return result, end - start

# Generate timing attack
def timing_attack(victim_public_key, G, p, a, d, key_size, true_private_key):
    guessed_key = 0   # Start with an empty key guess (all 0s)
    timing_diffs = [] # Store time differences for each bit

    for i in range(key_size): #Loop runs 384 times to guess all bits
        k0 = guessed_key
        k1 = guessed_key | (1 << (key_size - 1 - i))  # Set i-th bit to 1

        # Measure the time for both k0 and k1
        _, t0 = measure_time(elliptic_net_scalar_mult, k0, G, p, a, d)
        _, t1 = measure_time(elliptic_net_scalar_mult, k1, G, p, a, d)

        timing_diffs.append(t1 - t0)

        # Use longer time to infer a bit value of 1
        if t1 > t0:
            guessed_key = k1
        else:
            guessed_key = k0

        guessed_key &= (1 << key_size) - 1  # Ensure guessed private key stays
within 384-bit

    return guessed_key, timing_diffs

# Run the attack to guess Alice's private key
guessed_key, timing_data = timing_attack(alice_public, G, p, a, d, key_size,
alice_private)

# Ensure key bit length are equal
actual_key_bin = bin(alice_private)[2:].zfill(key_size)
```

```
guessed_key_bin = bin(guessed_key)[2:].zfill(key_size)

# Output results
print("Actual Private Key :", alice_private)
print("Guessed Private Key:", guessed_key)
print("Actual Key Length  :", len(actual_key_bin))
print("Guessed Key Length :", len(guessed_key_bin))
print("Attack Successful  :", alice_private == guessed_key)

# Graph
plt.figure(figsize=(8, 4))
sns.lineplot(x=range(key_size), y=timing_data)
plt.title("Timing Attacks on ECDH using Elliptic Net Method-numsp384t1 (Alice's
Key)")
plt.xlabel("Bit Position (MSB to LSB)")
plt.ylabel("Time Difference (t1 - t0)")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Appendix G:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import pearsonr

# Hamming weight calculation for power analysis (using XOR between data and
key byte)
def hamming_weight(x):
    """Calculate the Hamming weight (number of 1s) in a binary representation."""
    return bin(x).count('1')

# Simulate power consumption with added noise (more realistic noise and key +
plaintext dependency)
def simulate_power_consumption(plaintext, k_byte, noise_std=0.3):
    """Simulate power consumption based on Hamming weight model for each
nonce byte with noise."""
    base_trace = np.array([hamming_weight(plaintext_byte ^ k_byte) for
plaintext_byte in plaintext])
    noise = np.random.normal(0, noise_std, len(base_trace))  # Gaussian noise
    return base_trace + noise

# Generate random plaintext data (e.g., 1000 samples of random 8-bit values)
num_samples = 1000
plaintexts = np.random.randint(0, 256, num_samples)  # Random plaintexts as 8-bit
integers

# True secret nonce (32 bytes or 256 bits key length)
true_nonce = np.random.randint(0, 384, 48)  # 48 bytes for 384-bit key
(numsp384t1)

# Simulate power consumption traces for each byte of the nonce k
power_traces = []
for byte in true_nonce:
    trace = simulate_power_consumption(plaintexts, byte, noise_std=0.3)  #
Simulate with noise
    power_traces.append(trace)

# Perform the attack: correlate guesses with power traces
key_guesses = np.arange(256)  # Key space for byte (0-255)
correlations = []

# For each byte in the nonce k, perform the attack
success_threshold = 0.6  # Threshold for success (correlation must exceed this to be
considered true)
true_byte_count = 0
for byte_index in range(48):  # 48 bytes for 384-bit key
    byte_correlations = []
```

```
    # Correlate guessed nonce with real power traces
    for key_guess in key_guesses:
        guessed_power = simulate_power_consumption(plaintexts, key_guess)
        correlation, _ = pearsonr(power_traces[byte_index], guessed_power)
        byte_correlations.append(correlation)

    # Find the nonce guess with the highest correlation for this byte
    best_guess_index = np.argmax(byte_correlations)
    recovered_byte = key_guesses[best_guess_index]

    # Validation step with a threshold
    correlation_with_true_byte = byte_correlations[best_guess_index]
    validation = recovered_byte == true_nonce[byte_index] and
correlation_with_true_byte > success_threshold

    # Keep track of how many true guesses were made
    if validation:
        true_byte_count += 1

    # Output for each byte with validation
    print(f"Byte {byte_index} - True Byte: {true_nonce[byte_index]}, Guessed
Byte: {recovered_byte}, Validation: {validation}")

# Calculate overall success rate
success_rate = true_byte_count / 48  # 48 bytes for 384-bit key
print(f"Overall success rate: {success_rate * 100:.2f}%")

# Plot the correlation for the first byte as an example
plt.figure(figsize=(10, 6))
plt.plot(key_guesses, byte_correlations, label='Correlation with Power Traces')
plt.axvline(x=true_nonce[0], color='red', linestyle='--', label='True Nonce Byte 0')
plt.xlabel('Key Guess')
plt.ylabel('Correlation')
plt.title('Power Analysis Attacks on ECDSA using Binary Method-numsp384t1')
plt.legend()
plt.show()

# Output final attack results
print(f"Original True Nonce: {true_nonce}")
```

Appendix H:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import pearsonr

# Hamming weight calculation for power analysis (using XOR between data and
key byte)
def hamming_weight(x):
    """Calculate the Hamming weight (number of 1s) in a binary representation."""
    return bin(x).count('1')

# Simulate power consumption with added noise (more realistic noise and key +
plaintext dependency)
def simulate_power_consumption(plaintext, k_byte, noise_std=0.3):
    """Simulate power consumption based on Hamming weight model for each
nonce byte with noise."""
    base_trace = np.array([hamming_weight(plaintext_byte ^ k_byte) for
plaintext_byte in plaintext])
    noise = np.random.normal(0, noise_std, len(base_trace))  # Gaussian noise
    return base_trace + noise

# Generate random plaintext data (e.g., 1000 samples of random 8-bit values)
num_samples = 1000
plaintexts = np.random.randint(0, 256, num_samples)  # Random plaintexts as 8-bit
integers

# True secret nonce (32 bytes or 256 bits key length)
true_nonce = np.random.randint(0, 384, 48)  # 48 bytes for 384-bit key
(numsp384t1)

# Simulate power consumption traces for each byte of the nonce k
power_traces = []
for byte in true_nonce:
    trace = simulate_power_consumption(plaintexts, byte, noise_std=0.3)  #
Simulate with noise
    power_traces.append(trace)

# Perform the attack: correlate guesses with power traces
key_guesses = np.arange(256)  # Key space for byte (0-255)
correlations = []

# For each byte in the nonce k, perform the attack
success_threshold = 0.6  # Threshold for success (correlation must exceed this to be
considered true)
true_byte_count = 0
for byte_index in range(48):  # 48 bytes for 384-bit key
    byte_correlations = []
```

```
    # Correlate guessed nonce with real power traces
    for key_guess in key_guesses:
        guessed_power = simulate_power_consumption(plaintexts, key_guess)
        correlation, _ = pearsonr(power_traces[byte_index], guessed_power)
        byte_correlations.append(correlation)

    # Find the nonce guess with the highest correlation for this byte
    best_guess_index = np.argmax(byte_correlations)
    recovered_byte = key_guesses[best_guess_index]

    # Validation step with a threshold
    correlation_with_true_byte = byte_correlations[best_guess_index]
    validation = recovered_byte == true_nonce[byte_index] and
correlation_with_true_byte > success_threshold

    # Keep track of how many true guesses were made
    if validation:
        true_byte_count += 1

    # Output for each byte with validation
    print(f"Byte {byte_index} - True Byte: {true_nonce[byte_index]}, Guessed
Byte: {recovered_byte}, Validation: {validation}")

# Calculate overall success rate
success_rate = true_byte_count / 48  # 48 bytes for 384-bit key
print(f"Overall success rate: {success_rate * 100:.2f}%")

# Plot the correlation for the first byte as an example
plt.figure(figsize=(10, 6))
plt.plot(key_guesses, byte_correlations, label='Correlation with Power Traces')
plt.axvline(x=true_nonce[0], color='red', linestyle='--', label='True Nonce Byte 0')
plt.xlabel('Key Guess')
plt.ylabel('Correlation')
plt.title('Power Analysis Attacks on ECDSA using Elliptic Net Method-
numsp384t1')
plt.legend()
plt.show()

# Output final attack results
print(f"Original True Nonce: {true_nonce}")
```

POSTER

UTAR
UNIVERSITI TUNKU ABDUL RAHMAN

CRYPTANALYSIS OF ELLIPTIC CURVE SCALAR MULTIPLICATION ALGORITHMS

Leong Zhen Hong, Dr. Norliana Binti Muslims

Aim to analyze ECCSM, using BM and EN. Implementations are carried out using Twisted Edwards curves to evaluate the potential vulnerabilities under side-channel attacks.

Objectives

- To Identify potential vulnerabilities scalar multiplication algorithms.
- To implement ECDH and ECDSA using the BM and EN
- To evaluate through side-channel attack analysis on ECCSM

Methods

1. Start → ECDH → BM/ EN → SM → SCA
2. Start → ECDSA → BM/ EN → SM → SCA

Results

All objectives were achieved, including the implementation of ECDH and ECDSA using binary and elliptic net methods. Overall, the findings highlighted results in the implementations over side-channel attacks on Elliptic Curve Cryptography Scalar Multiplication.

	ECDH			
	BM		EN	
	Numsp 384t1	Numsp 512t1	Numsp 384t1	Numsp 512t1
Timing Attack	✓	✓	✓	✓

	ECDSA			
	BM		EN	
	Numsp 384t1	Numsp 512t1	Numsp 384t1	Numsp 512t1
Power Analysis Attack	✓	✓	✓	✓

** ✓ consider as success to secure, ⊗ consider as failed to secure. **

ECCSM: ELLIPTIC CURVE CRYPTOGRAPHY SCALAR MULTIPLICATION/ BM: BINARY METHOD/ EN: ELLIPTIC NET METHOD/ ECDH: ELLIPTIC CURVE DIFFIE-HELLMAN/ ECDSA: ELLIPTIC CURVE DIGITAL SIGNATURE ALGORITHM/ SCA: SIDE-CHANNEL ATTACK/ NUMS: NOTHING UP MY SLEEVE

Tools: Jupyter Lab