Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Jonatan Kifle Assefa Aalen

The Monodromy Leak: Analysis of a New Side-Channel Attack on Elliptic Curve Cryptography

Master's thesis in Cyber Security and Data Communication
Supervisor: Tjerand Silde
Co-supervisor: Caroline Sandsbråten & Jonathan Komada Eriksen
July 2026

NTNU
Norwegian University of
Science and Technology

Jonatan Kifle Assefa Aalen

The Monodromy Leak: Analysis of a New Side-Channel Attack on Elliptic Curve Cryptography

Master's thesis in Cyber Security and Data Communication
Supervisor: Tjerand Silde
Co-supervisor: Caroline Sandsbråten & Jonathan Komada Eriksen
July 2026

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Problem description

Elliptic curve cryptography (ECC) is widely used on the internet today in services such as Transport Layer Security (TLS), Signal, and Bitcoin. Although the theory behind ECC is mathematically robust, side-channel attacks can leak information. A recently proposed attack by Damien Robert shows a more effective technique to retrieve the secret key used in ECC. When an implementation uses projective coordinates and the Montgomery ladder, which are commonly used in scalar multiplication algorithms for ECC, the secret can be recovered from only one side-channel leakage.

This master's thesis investigates this new projective-coordinate leakage attack in both theory and practice. The work aims to clarify how the attack operates, reformulate it using more accessible mathematical tools, and assess its relevance for real-world cryptographic implementations. In particular, the thesis examines whether widely used ECC libraries are vulnerable, how such leakage could occur in practice, and which countermeasures remain effective against this attack. The goal is to improve understanding of the attack and to provide insight into its practical implications for the security of modern ECC-based systems.

Approved: 2026-02-19 – Tjerand Silde, NTNU (Main supervisor)

Abstract

Elliptic Curve Cryptography (ECC) secures a large share of internet traffic, yet the soundness of its underlying mathematics does not guarantee the safety of any particular implementation. Code that performs scalar multiplication with projective coordinates can expose the secret key, because a projective representation retains more information than the affine point it ultimately yields.

This thesis studies the monodromy leak attack of Damien Robert (IACR ePrint 2024), which recovers an entire secret scalar from a single leaked projective coordinate when scalar multiplication relies on a ladder. This is a far stronger result than earlier projective coordinate attacks, which extracted only a few bits from many observations.

The first aim of the thesis is to make the attack intelligible without the cubical-torsor and biextension theory of its original presentation. Instead, this thesis explains the attack with cubical points, canonical lifts, and an explicit formula for the deviance between the Montgomery and cubical ladders. The attack extracts the secret through a quadratic congruence solved with three discrete logarithms in the prime field together with the Chinese Remainder Theorem (CRT). These discrete logarithms can be solved for key sizes used in practice, as a core mechanic of the attack is moving the logarithms from the elliptic curve to a finite field. Executable SageMath code accompanies each step, helping the reader grasp the underlying concepts and confirm the reformulation empirically.

The second aim is to evaluate real-world exposure through a static-analysis survey of 28 open source libraries and 69 scalar multiplication routines. Of the 27 routines that satisfy the preconditions, almost all perform constant-time inversion, a clear improvement over the state reported by Aldaya et al. at TCHES in 2020. This defense, however, only closes the channel in which the coordinates are inferred from the timing of the field inversion. Under a broader model, in which an adversary obtains the coordinates directly, for example from unsanitized memory or fault injection, pre-ladder coordinate masking is the only effective defense, and it is absent from 23 of the 27 routines. One routine, the x25519 fallback in the Reticulum networking stack, is exposed even under the narrowest assumptions. Limitations in the search strategy and static analysis mean some widely used libraries fell outside the survey's scope, and compiler optimization may leave some analyzed libraries vulnerable, so these figures are a lower bound rather than a guarantee of security.

Sammendrag

Elliptisk kurvekryptografi sikrer store deler av verdens internettrafikk. Selv om den underliggende matematikken er solid, garanterer det ikke at implementasjoner er trygge. Kode som utfører skalarmultiplikasjon med projektive koordinater kan avsløre den hemmelige nøkkelen, fordi den projektive representasjonen har mer informasjon enn det affine punktet som det til slutt reduseres til.

Denne oppgaven studerer monodromilekkasjeangrepet til Damien Robert (IACR ePrint 2024) som utleder hele den hemmelige skalaren fra én eneste lekkasje av et projektivt koordinat når skalarmultiplikasjonen bruker en stige. Det er langt sterkere enn tidligere lekkasjebaserte angrep på projektive koordinater som kun klarer å oppdage noen få bits fra hver observasjon.

Det første målet med oppgaven er å gjøre angrepet forståelig uten innviklet teori om kubiske torsorer og biekstensjoner fra den opprinnelige utledningen. I stedet forklarer oppgaven angrepet ved hjelp av kubiske punkter, kanoniske løft og en eksplisitt formel for avviket mellom Montgomerystigen og den kubiske stigen. Angrepet finner nøkkelen gjennom en kvadratisk kongruens som løses med tre diskrete logaritmer i primkroppen sammen med det kinesiske restteoremet. Disse diskrete logaritmene kan løses fordi nøkkelstørrelsene som brukes i elliptiske kurver er små, og at angrepet flytter dem til en endelig kropp. SageMath-kode følger med i hvert steg slik at leseren kan forstå de underliggende konseptene på en mer empirisk måte.

Det andre målet er å vurdere eksponeringen til angrepet gjennom en statisk analyse av koden i 28 biblioteker med åpen kildekode og 69 skalarmultiplikasjonsprosedyrer. Av de 27 prosedyrene som oppfyller forutsetningene for angrepet, utfører nesten alle inversjon i konstant tid, som er en betydelig forbedring fra tilstanden Aldaya et al. rapporterte om i TCHES i 2020. Dette tiltaket lukker imidlertid bare inversjonskanalen. Under en bredere modell, der en angriper får tak i koordinatene direkte, for eksempel fra ikke-nullstilt minne eller feilinjeksjoner, er maskering av koordinatene før stigen det eneste effektive forsvaret, og det mangler i 23 av 27 prosedyrer. x25519 i Reticulum-nettverksstakken er den eneste prosedyren som er eksponert under den smaleste trusselmodellen. Begrensninger i søkestrategien og den statiske analysen gjør at noen populære biblioteker kan ha falt utenfor undersøkelsen, og kompilatoroptimalise-

ring kan føre til at noen av de analyserte bibliotekene likevel er sårbare. Dermed er disse tallene heller en nedre grense enn en sikkerhetsgaranti.

Preface

This thesis marks the end of my Master of Science in Cyber Security and Data Communication at the Norwegian University of Science and Technology (NTNU). The thesis was supervised by Tjerand Silde and Caroline Sandsbråten at the Department of Information Security and Communication Technology (IIK) at NTNU, and Jonathan Komada Eriksen at KU Leuven.

Acknowledgements

Above all, I'm very thankful to my supervisors. Thank you, Tjerand, for making the Crypto-lab available for use throughout the semester. It has been a blessing to have my own desk and to be sitting next to fellow cryptography students, who have always helped me out and been my opponents in countless matches of ping pong. Thank you, Jonathan, for your assistance in understanding the monodromy leak, and for motivating me to pull through the most dense part of the thesis. And most of all, thank you, Caroline, for making me comfortable in asking for help, for reading through and correcting my drafts, and for motivating me throughout the pre-project and master's.

I would like to thank my family for always making me believe in myself, and my dad in particular for tricking me into liking computers from a young age.

Finally, I am thankful to the friends I've made here in Trondheim throughout these five years and to my old friends from back home, for keeping my spirits up, making me smile, and waiting on the other side of long days at the lab.

Contents

List of Figures

List of Tables

List of Acronyms

AI Artificial Intelligence.

CRT Chinese Remainder Theorem.

CVE Common Vulnerability and Exposure.

DLP Discrete Logarithm Problem.

ECC Elliptic Curve Cryptography.

ECDLP Elliptic Curve Discrete Logarithm Problem.

ECDSA Elliptic Curve Digital Signature Algorithm.

HSM Hardware Security Module.

NIST National Institute of Standards and Technology.

NTNU Norwegian University of Science and Technology.

PLWC Partially-Long Weierstrass Curve.

SCA Side-Channel Attack.

SEC Standards for Efficient Cryptography.

SPA Simple Power Analysis.

TLS Transport Layer Security.

Chapter 1
Introduction

1.1 Motivation

ECC lies at the core of modern internet security. It protects web traffic through Transport Layer Security (TLS) [Res18], secures end-to-end encrypted messaging in the Signal protocol [Sig25], and authorizes transactions in Bitcoin [Bit]. A central reason for its widespread adoption is that it offers the same level of security as older public-key systems while using substantially smaller keys, which makes it especially attractive for constrained devices. The mathematical foundations of ECC are robust, and the Discrete Logarithm Problem (DLP) on which its security rests is believed to be computationally infeasible for the key sizes used in practice [Aal25].

A concrete implementation, however, can leak secret information even when the underlying mathematics is sound. Such side-channel leakage may arise from execution timing, cache behavior, or sensitive values left in memory after use. One particular weakness concerns the projective coordinates that implementations commonly use to speed up scalar multiplication. The projective representation of a computed point carries more information than the affine value it is ultimately reduced to, and an adversary who observes it can attempt to recover the secret scalar.

The severity of this projective coordinate leakage has changed dramatically over the past two decades. Naccache, Smart, and Stern [NSS04] first defined the projective coordinate leak attack in 2004 and gave a concrete method for recovering part of a secret scalar from a leaked projective point. Sixteen years later, Aldaya et al. [CPB20b] brought the attack from theory into practice: they surveyed real-world cryptographic libraries, demonstrated a working exploit against a widely deployed implementation, and filed Common Vulnerability and Exposure (CVE) reports that prompted maintainers to respond. More recently, Robert [Rob24] described a substantially more powerful attack called the monodromy leak attack, which is the subject of this thesis[1] . Where the earlier attacks needed many leaked outputs

[1] *Monodromy* is a notion from geometry and topology for the transformation an object undergoes

to recover only a handful of secret bits, the monodromy leak attack can recover the entire secret scalar from a *single* leaked projective coordinate, provided the implementation combines a ladder with projective coordinates, a combination that is exceedingly common in practice.

Despite its power, Robert's result has so far attracted little attention from the developers it most concerns. It appears at the end of a dense paper whose primary subject is fast pairing computation, expressed in a dense mathematical language that does not entice any library maintainer to read past the abstract. No practical demonstration of the attack exists, no CVEs have been filed, and the libraries that might be affected have not visibly responded. This is the same position the 2004 attack occupied before Aldaya et al. acted on it, and in that instance sixteen years passed before a correct theoretical result became a recognized practical concern. There is little reason to expect the monodromy leak attack to be treated with greater urgency on its own.

This thesis is motivated by the gap between what the monodromy leak attack can do and how little is currently understood about its real-world relevance. It aims to make the attack accessible beyond the small community familiar with its underlying mathematics, and to determine whether widely used ECC libraries are in fact exposed to it.

1.2 Research Questions and Objectives

This section lays out the goals of the thesis. The work is organized around two overarching objectives, one theoretical and one practical, each refined into a set of more concrete research questions that guide the chapters that follow. Both the objectives and the questions are carried over from this thesis's pre-project [Aal25].

1.2.1 Objective 1: Theoretical Analysis and Reformulation

We aim to conduct a comprehensive study of Robert's [Rob24] projective coordinate leak attack, with the aim of reformulating the attack using more accessible mathematical tools. This objective emphasizes both technical rigor and conceptual clarity, ensuring that the mechanics of the attack can be understood without reliance on highly advanced mathematical machinery.

1.2.2 Objective 2: Practical Evaluation and Relevance

We aim to assess the real-world applicability of Robert's [Rob24] attack by investigating its potential impact on widely used ECC libraries. This includes identifying

when carried once around a closed loop. It is used here only as the name Robert [Rob24] gave the attack. The theory behind the name is beyond the scope of this thesis.

how projective coordinate leakage can occur, evaluating existing countermeasures, and determining whether current implementations remain vulnerable under realistic conditions.

1.2.3 Research Questions

To reach the aforementioned objectives, we have developed a set of research questions. They act as a helpful guide, being smaller but still tangible goals that contribute to solving the main problem.

Questions addressing Objective 1:

Research Question 1. What advantages are gained by combining the Montgomery ladder with projective coordinate systems, and how do these design choices influence efficiency and vulnerability to side-channel attacks?

Research Question 2. How does Robert's attack exploit leakage of projective coordinates?

Research Question 3. How does the effectiveness of Robert's attack compare to earlier projective coordinate leakage attacks in terms of feasibility, efficiency, and practical impact?

Questions addressing Objective 2:

Research Question 4. Which countermeasures currently exist to mitigate projective coordinate leakage, and which of these remain effective against Robert's attack?

Research Question 5. How does a projective coordinate leak occur in practice?

Research Question 6. Which widely used ECC libraries implement projective coordinates, and to what extent are they vulnerable to Robert's attack under realistic operational scenarios?

1.3 UN Sustainable Development Goals

The United Nations' Sustainable Development Goals set out seventeen objectives for global development [Uni26c]. Although this thesis is technical in nature, the security of the cryptographic systems it studies directly supports two of these goals: Goal 9, Industry, Innovation and Infrastructure, and Goal 16, Peace, Justice and Strong Institutions.

Goal 9 calls for the development of "quality, reliable, sustainable and resilient infrastructure [...] with a focus on affordable and equitable access for all" (Target 9.1) [Uni26b]. The internet and the secure communication channels built upon it are now critical infrastructure, and ECC is one of their foundational components (see Section 1.1). The reliability of this infrastructure depends not only on the soundness of the underlying mathematics, but also on the correctness of the implementations that deploy it. By examining how the monodromy leak attack threatens real cryptographic libraries, and by identifying which countermeasures remain effective against it, this thesis contributes to keeping infrastructure resilient against an emerging class of attacks. The decision to study open source libraries is itself aligned with this goal: cryptographic security should rest on public scrutiny rather than on obscurity, and reproducible analysis of openly available code is what allows weaknesses to be found and repaired before they are exploited.

Goal 16 includes the aim to "ensure public access to information and protect fundamental freedoms, in accordance with national legislation and international agreements" (Target 16.10) [Uni26a]. Encryption supports both sides of this target. The ECC that will be studied in this thesis secures web traffic through TLS, and private messaging through Signal, allowing people to access information without their communications being read or altered. This ensures both privacy and public access to information. A citizen who knows that their private communications are exposed will be less likely to access information over the internet. Therefore, enhancing privacy is correlated with increasing public access to information. A side-channel attack that recovers a secret key undermines these protections directly, exposing private communications and the individuals who rely on them. By improving the understanding of the monodromy leak attack and by clarifying the defenses available against it, this thesis supports the continued ability of cryptography to protect privacy and to keep access to information both open and secure.

1.4 Contributions

This thesis makes four main contributions. The first is an accessible reformulation of the monodromy leak attack. Robert's original presentation [Rob24] relies on cubical torsors, biextensions, and related machinery that is unfamiliar to most developers. Chapter 3 presents the attack using a smaller set of concepts and explains them through their role in the attack, with less emphasis on their full mathematical foundations. The reformulation also derives an explicit formula for the deviance between the Montgomery and cubical ladders. Executable SageMath code accompanies the explanation throughout, demonstrating each concept in practice and convincing readers empirically of the re-formulation's correctness.

The second contribution is a survey of projective coordinate leakage in real

libraries. A static-analysis review of 28 widely used open source cryptographic libraries is presented, covering 69 directly analyzed scalar multiplication routines. Each routine is assessed against the preconditions for the monodromy leak attack and the countermeasures that defend against it. It provides an up-to-date picture of how exposed the current ECC ecosystem is. The methodology is described in Chapter 4 and the results in Chapter 5.

From this survey follows the third contribution, the identification of a vulnerable implementation. Of the 27 routines that meet the attack's preconditions, all but one implement an effective countermeasure. The exception is the internal `x25519` routine in the Reticulum networking stack, whose Python fallback uses a non-constant-time inversion and implements no masking, leaving it exposed to the monodromy leak attack.

Finally, the thesis offers a comparison with prior attacks and an analysis of countermeasures in practice. Chapter 6 compares the monodromy leak attack to the earlier attacks of Naccache, Smart, and Stern [NSS04] and Aldaya et al. [CPB20b] in terms of attacker requirements, mathematical mechanism, and practical impact. It further examines which countermeasures remain effective against Robert's attack and the channels through which projective coordinates may leak in deployed systems.

Chapter 2
Background

2.1 Finite Fields

Elliptic curve cryptography is built on arithmetic in finite fields, so we begin by establishing the relevant structures and the operations performed on them. A *field* is a set equipped with addition, subtraction, multiplication, and division by nonzero elements. Addition and multiplication are associative, commutative and have identity elements (0 for addition, 1 for multiplication). Every element has an additive inverse, and every nonzero element has a multiplicative inverse. A *finite field* is, as the name suggests, a field with finitely many elements [MvOV97, Chapter 2.5].

The most important finite field for our purposes is the field of integers modulo a prime p, written $\mathbb{F}_p$ (or equivalently GF(p), for "Galois field"). Its elements are the integers $\{0, 1, 2, \ldots, p-1\}$, and addition and multiplication are carried out modulo p [MvOV97, Chapter 2.6]. A key detail is that $\mathbb{F}_p$ is a field *precisely because p is prime*. When the modulus is prime, every nonzero element has a multiplicative inverse (shown in Section 2.1.2). This is the reason ECC is commonly built over prime moduli. Throughout this thesis, the example curve curve25519 is defined over $\mathbb{F}_p$ with $p = 2^{255} - 19$, a prime of roughly 256 bits [LHT16].

2.1.1 The Multiplicative Group and Generators

The nonzero elements of $\mathbb{F}_p$ form a group under multiplication, denoted $\mathbb{F}_p^*$. Since the only element excluded is 0, this group has exactly $p-1$ elements. A fundamental result is that $\mathbb{F}_p^*$ is *cyclic*, meaning there exists an element g such that every nonzero element of the field can be expressed as a power of g [MvOV97, Chapter 2.4]:

$$\mathbb{F}_p^* = \{\, g^0, g^1, g^2, \ldots, g^{p-2} \,\}.$$

Such an element g is called a *generator* of the field. The powers of a generator cycle with period $p-1$; that is, $g^{p-1} = 1$, and the exponents may be reduced modulo $p-1$.

For example, consider $\mathbb{F}_7^* = \{1, 2, 3, 4, 5, 6\}$. The element $g = 3$ is a generator, since its successive powers

$$3^1 = 3, \quad 3^2 = 2, \quad 3^3 = 6, \quad 3^4 = 4, \quad 3^5 = 5, \quad 3^6 = 1 \pmod 7$$

run through every nonzero element before returning to 1 after $p - 1 = 6$ steps.

Because $\mathbb{F}_p^*$ is cyclic, every nonzero element a can be written uniquely as $a = g^e$ for some exponent $e \in \{0, 1, \ldots, p-2\}$. This exponent is called the *discrete logarithm* of a to the base g. The difficulty of recovering e from a is the basis of an entire family of cryptographic problems, discussed in Section 2.4.

2.1.2 Modular Inversion

The *multiplicative inverse* of a nonzero element $a \in \mathbb{F}_p$ is the element $a^{-1} \in \mathbb{F}_p$ satisfying

$$a \cdot a^{-1} \equiv 1 \pmod p.$$

Division in the field is then simply multiplication by an inverse: $x/y \equiv x \cdot y^{-1} \pmod p$. In finite-field arithmetic, division is therefore commonly referred to as inversion.

Inversion is a central operation in ECC. As will be shown in Section 2.5, converting a point from projective coordinates back to the affine plane requires computing $x = X \cdot Z^{-1} \bmod p$. The way this inverse is computed turns out to have direct security consequences, so it is worth introducing the two standard methods here.

The first is the *extended Euclidean algorithm*, which computes the inverse from the relationship $\gcd(a, p) = 1$. It is efficient, but the sequence of operations it performs depends on the value being inverted [MvOV97, Chapter 2.4].

The second method relies on *Fermat's little theorem* [MvOV97, Chapter 2.4], which states that for a prime p and any integer a not divisible by p,

$$a^{p-1} \equiv 1 \pmod p.$$

Since $a^{p-1} = a \cdot a^{p-2}$, it follows that a^{p-2} is the multiplicative inverse of a:

$$a^{-1} \equiv a^{p-2} \pmod p.$$

The inverse can thus be obtained by modular exponentiation, raising a to the fixed power $p - 2$. Unlike the extended Euclidean algorithm, this performs the same sequence of operations regardless of the input value, a property that becomes important when discussing side-channel countermeasures in Section 5.2.1.

2.2 Elliptic Curves

This section introduces elliptic curves and the arithmetic defined on them. We begin with the general Weierstrass form and the group law, which gives the points of a curve the structure of an algebraic group. We then specialize to Montgomery curves which is the curve class of curve25519 and the setting in which the monodromy leak attack is developed in Chapter 3. Finally, we describe the order of a curve and the large prime-order subgroup on which both its security and the attack against it depend. The material on Weierstrass curves and the group law is reused from the thesis pre-project [Aal25] and follows Washington [Was08].

2.2.1 The Weierstrass Equation

An elliptic curve E is often written in the form:

$$y^2 = x^3 + Ax + B,$$

where A and B are constants (satisfying Equation 2.1 below) in some field K, and x and y are the affine coordinates of some point on E. This is known as a curve on *Weierstrass form*. Many of the curves analysed in this thesis are given in this form, including secp256k1 and the National Institute of Standards and Technology (NIST) prime curves examined in Chapter 5.

If $A, B \in K$, E *is defined over* K, and if we want to consider points with coordinates in some field $L \supseteq K$, we write $E(L)$:

$$E(L) = \{\infty\} \cup \{(x, y) \in L \times L \mid y^2 = x^3 + Ax + B\}.$$

Note that the set always contains the point ∞, which acts as the identity element of the group law introduced below.

To avoid singularities (non-distinct roots) such as cusps and self-intersections (see Figure 2.1), we must be sure that:

$$4A^3 + 27B^2 \neq 0. \tag{2.1}$$

2.2.2 The Group Law

Elliptic curve addition (in the classical sense) can be described geometrically, as shown in Figure 2.2. Assume $P_1 \neq P_2$, with $P_1 \neq \infty$ and $P_2 \neq \infty$, where $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$.

Draw the line L through P_1 and P_2. Assuming for now that $x_1 \neq x_2$, the line has slope

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1},$$

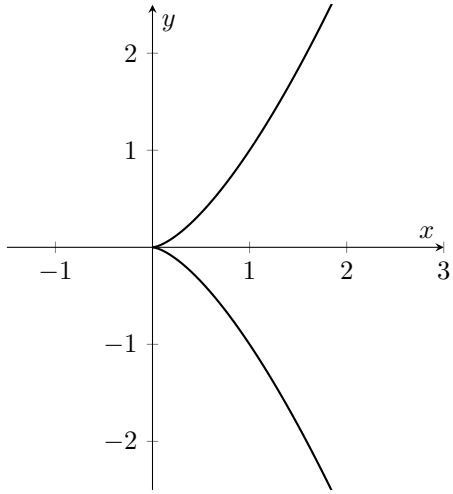


Figure 2.1: A cusp in the graph $y^2 = x^3$ (where $4A^3 + 27B^2 = 0$).

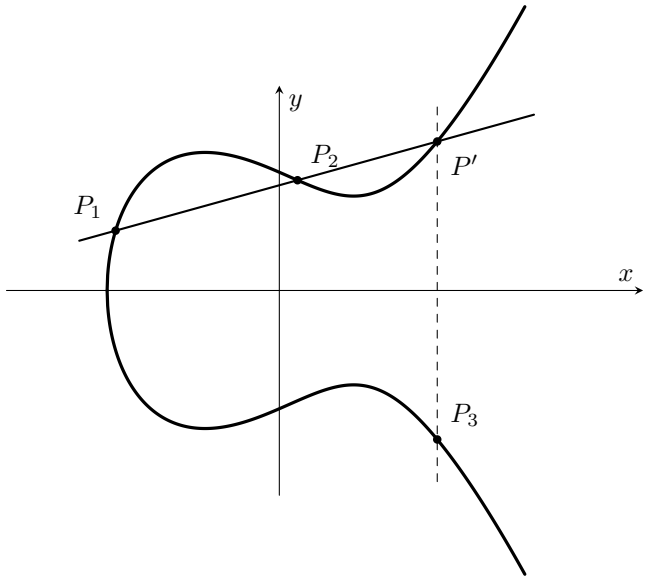


Figure 2.2: Addition on an elliptic curve defined over $\mathbb{R}$: the sum $P_1 + P_2 = P_3$ is obtained by intersecting the line through P_1 and P_2 with the curve at a third point P' and reflecting it across the x-axis.

and is given by

$$y = \lambda(x - x_1) + y_1.$$

Find the third intersection P'. Substituting the line into the curve equation $y^2 = x^3 + Ax + B$ gives

$$\big(\lambda(x - x_1) + y_1\big)^2 = x^3 + Ax + B,$$

which rearranges into

$$0 = x^3 - \lambda^2 x^2 + \cdots .$$

Here "$\cdots$" does not denote an infinite series, but the lower-order terms that are not needed for what follows. The roots of this cubic are the x-coordinates of the points where L meets E. The roots of P_1 and P_2 are already known.

For a cubic $x^3 + ax^2 + bx + c$ with roots r, s, and t:

$$x^3 + ax^2 + bx + c = (x - r)(x - s)(x - t) = x^3 - (r + s + t)x^2 + \cdots$$

Comparing the x^2 coefficients gives $a = -(r + s + t)$, so a known pair of roots determines the third:

$$t = -a - r - s.$$

Applying this with $a = -\lambda^2$ and the known roots x_1, x_2, the third root is the x-coordinate of $P' = (x', y')$:

$$x' = \lambda^2 - x_1 - x_2, \qquad y' = \lambda(x' - x_1) + y_1.$$

Reflect P' across the x-axis. The sum $P_3 = (x_3, y_3)$ is the reflection of P':

$$x_3 = x' = \lambda^2 - x_1 - x_2, \qquad y_3 = -y' = \lambda(x_1 - x_3) - y_1.$$

In conclusion,

$$P_3 = \big(\lambda^2 - x_1 - x_2,\ \lambda(x_1 - x_3) - y_1\big), \qquad \lambda = \frac{y_2 - y_1}{x_2 - x_1}.$$

Special cases of elliptic curve addition

Case $x_1 = x_2$, $y_1 \neq y_2$. The line through P_1 and P_2 is vertical and meets E only at ∞. Reflecting ∞ across the x-axis gives ∞ again, so

$$P_1 + P_2 = \infty.$$

Case $P_1 = P_2 = (x_1, y_1)$ *(point doubling).* As P_2 approaches P_1, the line through them becomes the tangent to E at P_1. Its slope is found by differentiation of the curve equation:

$$y^2 = x^3 + Ax + B \implies 2y_1 \frac{dy}{dx} = 3x_1^2 + A \implies \lambda = \frac{dy}{dx} = \frac{3x_1^2 + A}{2y_1}.$$

If in addition $y_1 = 0$, the tangent is vertical, so $P_1 + P_2 = \infty$.

Assume $y_1 \neq 0$. As before, substituting the tangent line $y = \lambda(x - x_1) + y_1$ into the curve gives

$$0 = x^3 - \lambda^2 x^2 + \cdots .$$

This time we only know one root, x_1. However, since P_2 approaches P_1, this is a double root. Following the same argument as earlier then yields x_3 and y_3:

$$x^3 - \lambda^2 x^2 + \cdots = (x - x_1)(x - x_1)(x - t) = x^3 - (2x_1 + t)x^2 + \cdots$$

$$\lambda^2 = 2x_1 + t,$$

$$x_3 = t = \lambda^2 - 2x_1, \qquad y_3 = \lambda(x_1 - x_3) - y_1.$$

Case $P_1 = (x_1, y_1)$, $P_2 = \infty$. The vertical line from P_1 goes through ∞ and meets E at the reflection of P_1 over the x-axis, P_1'. To obtain the final result P_1' is reflected back to P_1, so

$$P_1 + \infty = P_1$$

for every point P_1 on E (including ∞ itself).

Summary of group addition

For $E : y^2 = x^3 + Ax + B$ with $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, and $P_1 + P_2 = P_3$:

If $x_1 \neq x_2$:

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1, \quad \lambda = \frac{y_2 - y_1}{x_2 - x_1}.$$

If $x_1 = x_2$ but $y_1 \neq y_2$:

$$P_1 + P_2 = \infty.$$

If $P_1 = P_2$ and $y_1 \neq 0$:

$$x_3 = \lambda^2 - 2x_1, \quad y_3 = \lambda(x_1 - x_3) - y_1, \quad \lambda = \frac{3x_1^2 + A}{2y_1}.$$

If $P_1 = P_2$ and $y_1 = 0$:

$$P_1 + P_2 = \infty.$$

For all points P on E:

$$P + \infty = P.$$

2.2.3 Montgomery Curves

While the Weierstrass form is very common for elliptic curves, the curves used in this thesis are more naturally described in another form, introduced by Montgomery [Mon87]. A *Montgomery curve* over a field K is defined by

$$By^2 = x^3 + Ax^2 + x,$$

with $A, B \in K$ and the non-singularity condition $B(A^2 - 4) \neq 0$. The running example throughout this thesis, curve25519, is the Montgomery curve with $B = 1$ and $A = 486662$ defined over $\mathbb{F}_p$ with $p = 2^{255} - 19$, using the base point with $x = 9$ [LHT16].

Montgomery curves are relevant because they allow for particularly efficient arithmetic. As shown in Section 2.7, scalar multiplication on a Montgomery curve can be carried out using only the x-coordinate of a point, discarding y entirely. The constant $(A + 2)/4$ appearing in the y-independent doubling formulas further below is from the Montgomery form given here.

2.2.4 Group Order and the Prime-Order Subgroup

Over a finite field, an elliptic curve has only finitely many points. The number of points on E over $\mathbb{F}_p$, including the point at infinity, is called the *order* of the curve, denoted by $\#E(\mathbb{F}_p)$. Hasse's theorem bounds this count close to p [Was08, Theorem 4.2]:

$$\left|\#E(\mathbb{F}_p) - (p + 1)\right| \leq 2\sqrt{p},$$

so the group is roughly the size of the underlying field.

For cryptographic use, a curve is chosen so that its order contains a large prime factor (see Section 2.4). The order is written as a product

$$\#E(\mathbb{F}_p) = h \cdot \ell,$$

where ℓ is a large prime and h, the *cofactor*, is a small integer. A base point G of order ℓ then generates a cyclic subgroup

$$\langle G \rangle = \{\, \infty, G, 2G, \ldots, (\ell - 1)G \,\}$$

of prime order ℓ, similar to the cyclic structure of $\mathbb{F}_p^*$ described in Section 2.1.1. All cryptographic operations take place within this subgroup. For curve25519 the cofactor is $h = 8$, and the prime subgroup order is

$$\ell = 2^{252} + 27742317777372353535851937790883648493,$$

which is the value used as the subgroup order in the implementations presented in Chapter 3 [LHT16].

Two properties of ℓ will matter later. First, the security of a curve depends on ℓ, as recovering the scalar m from a point mG is dependent on the subgroup size. We want ℓ to be as large as possible, as attacks like Pollard's rho takes time proportional to $\sqrt{\ell}$ (see Section 2.4). Second, the canonical lift in Section 3.3 requires ℓ to be coprime to $p - 1$. The coprimality is not guaranteed just because ℓ is a large prime, as it is still smaller than $p - 1$. However, if $\ell \mid p - 1$, the curve has embedding degree 1, meaning it is highly vulnerable to the MOV attack [BL24]. This means that all secure curves make sure $\gcd(\ell, p - 1) = 1$.

2.3 Scalar Multiplication and the Montgomery Ladder

The central operation in ECC is scalar multiplication: computing the point $[m]P$ from a scalar m and a point P[1]. Its efficiency determines the speed of every ECC protocol, and the way it is implemented determines whether the secret scalar can leak through a side channel. First, this section introduces the standard double-and-add method and demonstrates why a naive implementation is vulnerable to side-channel analysis. It then presents the Montgomery ladder as a constant-operation alternative to mitigate this leakage, and notes that this specific algorithm is the target of the monodromy leak attack. The material in this section is reused from the thesis pre-project [Aal25], which follows Washington [Was08].

2.3.1 Double-and-Add

Suppose we have a point $P = (x, y)$ on the curve E and some integer m, and that we want to calculate $[m]P$. Unfortunately the group operator of E is $+$, and m is not even a point on the curve. A trivial method of calculating $[m]P$ is to perform addition of P to itself m times:

$$[m]P = \pm P \pm P \pm P \pm P ... \ (m \text{ times}).$$

When computing $[m]P$ in ECC, common sizes of m are 256 and 384 bits. For these sizes it is practically impossible to add P to itself repeatedly, as 2^{256} or 2^{384} additions

[1] In order to differentiate between standard point multiplication $m \cdot P = (m \cdot x, m \cdot y)$ and elliptic curve scalar multiplication, we will use this notation with brackets around the scalar: $[m]P$.

Algorithm 2.1 Double-and-Add

Input: The point at infinity ∞, point P, scalar m

1: $b \leftarrow$ bit representation of m ▷ from LSB to MSB
2: $S \leftarrow \infty$ ▷ Start with identity
3: $T \leftarrow P$
4: **for** each *bit* in b **do**
5: **if** $bit = 1$ **then**
6: $S \leftarrow S + T$ ▷ Point addition
7: **end if**
8: $T \leftarrow T + T$ ▷ Point doubling
9: **end for**

Return $S = [m]P$

Figure 2.3: Double-and-Add algorithm for computing $[m]P$.

are needed. Instead a technique called *successive doubling* is used. For example, to compute $[23]P$:

$$[2]P = P + P, \quad [4]P = [2]P + [2]P, \quad [8]P = [4]P + [4]P,$$
$$[16]P = [8]P + [8]P, \quad [23]P = [16]P + [4]P + [2]P + P.$$

This allows for faster computation of $[m]P$ for large m. In a field such as $\mathbb{R}$ the coordinates of large $[m]P$ may be impractically large; however, in a finite field (as used in practice) such growth is contained.

Letting b be the binary representation of m, the double-and-add algorithm for computing $[m]P$ is given in Algorithm 2.1. This algorithm performs successive doubling and adding depending on the binary representation of the secret scalar m. Returning to the example where $[23]P$ was computed, the bit representation of $m = 23$ is 10111. Working from the least significant bit to the most significant bit, we perform both a double and an add if the bit is 1, and only a double otherwise (see Figure 2.4).

2.3.2 Side-Channel Attacks

A Side-Channel Attack (SCA) is an exploit where an attacker accesses information from a side channel of a running cryptographic implementation and uses statistical analysis to gather information about the underlying operations and their sequence. The goal is to find information about a secret that is otherwise inaccessible. Typical side channels are power usage, execution time, and electromagnetic radiation.

Considering Algorithm 2.1 above, a side-channel attack can trivially be performed. If an algorithm's execution depends on the bits of a secret key, it is very probable

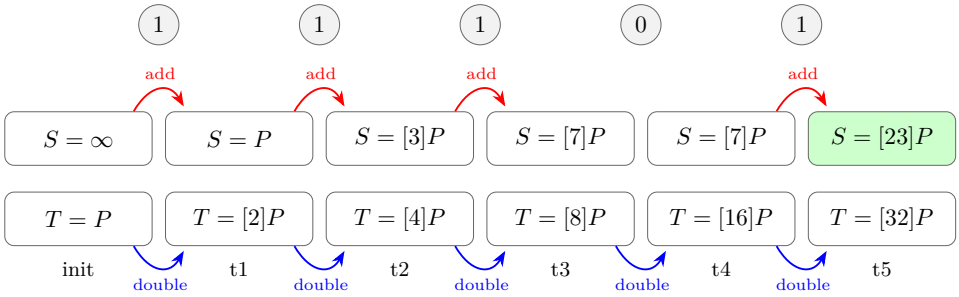


Figure 2.4: Step-by-step computation of $[m]P$ for $m = 23 = (10111)_2$ with Double-and-add going from least to most significant bit.

that an SCA can be performed successfully. In the double-and-add algorithm, a point addition is performed only if a bit in the secret value m is 1. Imagine an adversary with access to a power trace of a run of the algorithm at sufficient resolution. Because a point addition followed by a point doubling consumes noticeably more power than a single point doubling, an adversary would be able to differentiate 0s from 1s, meaning they could discover the entire secret m bit by bit.

The vulnerability occurs because the code performs different operations based on the contents of m. To prevent such attacks, one can use the Montgomery ladder, introduced next, which performs both a doubling and an addition at every bit of m regardless of its value.

2.3.3 The Montgomery Ladder

Like double-and-add, the Montgomery ladder receives a point P and the bit representation of a secret scalar m, and computes $[m]P$. The difference is that in the Montgomery ladder, a doubling and an addition are always performed, no matter the current bit. The bit instead determines which of two internal variables is doubled and which is added to the other; these variables are swapped to guarantee the correct result while ensuring a doubling and an addition occur at every step. See Algorithm 2.2. Again, it is helpful to look at a concrete run. In Figure 2.6, $[m]P$ is computed with $m = 23 = (10111)_2$.

2.4 The Discrete Logarithm Problem

Both the security of ECC and the attack studied in this thesis depend on the DLP. In a cyclic group generated by an element g, combining g with itself m times is easy, but recovering m from the result is the discrete logarithm problem. We first

Algorithm 2.2 Montgomery ladder

Input: The point at infinity ∞, point P, scalar m

1: $b \leftarrow$ bit representation of m ▷ from MSB to LSB
2: $R_0 \leftarrow \infty$
3: $R_1 \leftarrow P$
4: **for** each *bit* in b **do**
5: **if** $bit = 0$ **then**
6: $R_1 \leftarrow R_0 + R_1$ ▷ Point addition
7: $R_0 \leftarrow [2]R_0$ ▷ Point doubling
8: **else**
9: $R_0 \leftarrow R_0 + R_1$ ▷ Point addition
10: $R_1 \leftarrow [2]R_1$ ▷ Point doubling
11: **end if**
12: **end for**

Return $R_0 = [m]P$

Figure 2.5: Montgomery ladder algorithm for computing $[m]P$.

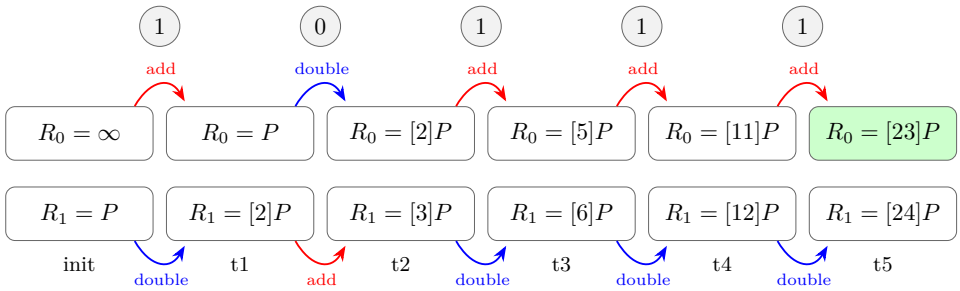


Figure 2.6: Montgomery ladder computation of $[m]P$ for $m = 23 = (10111)_2$. Each step alternates doubling and adding on R_0, R_1 depending on the bit value.

met this problem in Section 2.1.1 in $\mathbb{F}_p^*$, where the discrete logarithm of $a = g^m$ is the exponent m. For elliptic curves, Section 2.3 shows that computing mG is straightforward. We will show that recovering the scalar m from the resulting point is the Elliptic Curve Discrete Logarithm Problem (ECDLP). The same problem can be posed in either group, but its difficulty differs greatly between them, and that difference is what the attack exploits.

Let G be a base point of large prime order ℓ (Section 2.2.4) and let $Q = [m]G$ for a secret scalar m. The ECDLP is the task of recovering m given only G and Q. This is exactly what an attacker must do to derive a private key from a public key, which is why its presumed hardness secures ECC. The best known algorithms on a well-chosen curve are *generic*, exploiting no structure of the group [JOP14, p.31]. Pollard's rho method takes time proportional to $\sqrt{\ell}$, about 2^{128} operations for a 256-bit ℓ. Also, choosing ℓ to be a large prime blocks the Pohlig-Hellman method, which helps only when the group order has small prime factors [Was08, Chapter 5.2].

In the multiplicative group $\mathbb{F}_p^*$ the same generic algorithms apply, but they are not the best available. The arithmetic structure of the field admits index-calculus methods, whose strongest form, the number field sieve, solves the discrete logarithm in *sub-exponential* time [Was08, Chapter 5.1]. No analogue is known on elliptic curves, which is why the ECDLP is harder than the finite-field DLP, and consequently why ECC achieves comparable security with much smaller keys [JOP14, p. 31].

This asymmetry is the foundation of the monodromy leak attack. Rather than attacking the ECDLP directly, the attack moves the recovery of m to a problem that involves discrete logarithms in $\mathbb{F}_p^*$ (Sections 3.6 and 3.8). At the typical size of elliptic curves, roughly 256 bits, these are feasible to solve even though the corresponding ECDLP is not. The monodromy leak attack in Appendix A demonstrates this gap directly: a consumer grade laptop recovers the secret scalar from Curve25519 in about 15 minutes, even though breaking the ECDLP on this curve would require on the order of 2^{126} operations.

2.5 Projective Coordinates

Scalar multiplication, introduced in Section 2.3, repeatedly adds and doubles points on the curve. When performed directly in affine coordinates, each such operation requires a field inversion (see Section 2.2.2). We will show that this is by far the most expensive arithmetic operation in $\mathbb{F}_p$. Projective coordinates remove these intermediate inversions at the cost of a single inversion at the very end, and are for this reason used in most ECC implementations (see Section 5.4.1). This section explains why inversions are worth avoiding and introduces the projective representation of a point. The material in this section is adapted from the thesis pre-project [Aal25].

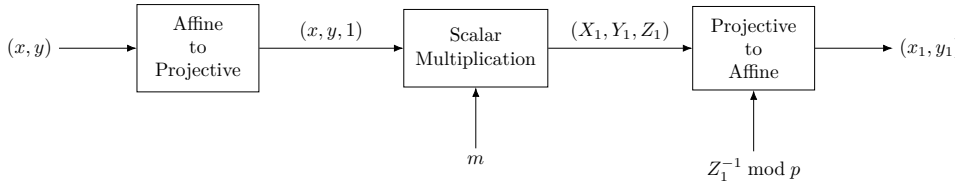


Figure 2.7: A general method for using projective coordinates for scalar multiplication [CPB20a].

2.5.1 Motivation: Avoiding Inversions

Adding or doubling points in affine coordinates requires a field division. Recall from Section 2.1.2 that division in $\mathbb{F}_p$ is carried out as multiplication by a modular inverse, $x/y \equiv x \cdot y^{-1} \pmod{p}$. The *Handbook of Elliptic and Hyperelliptic Curve Cryptography* estimates that a single inversion takes between 9 and 40 times as long as a multiplication, while a squaring takes roughly 0.8 times as long as a multiplication [CFAD06, p. 282]. Additions and subtractions are usually not counted, as they are far cheaper than the remaining operations [Was08, p. 42]. This means that field inversions are the most costly operations, and reducing the number of times they are performed is therefore the dominant concern when optimizing scalar multiplication.

A scalar multiplication $[m]P$ for a 256-bit scalar performs at least 256 doublings, together with the additions dictated by the bits of m (Section 2.3). When carried out in affine coordinates, each of these operations would require its own inversion. Projective coordinates avoid this entirely: all intermediate additions and doublings are computed without any inversion, and a single inversion is needed only at the end, to return the final result to the affine plane. For a visual representation, see Figure 2.7. For a 256-bit scalar this replaces several hundred inversions with one, and the modest increase in the number of multiplications and squarings is a worthwhile trade.

2.5.2 Projective Representation and Equivalence Classes

A point given in affine coordinates as (x, y) is moved to projective coordinates by writing

$$x = \frac{X}{Z}, \qquad y = \frac{Y}{Z}, \tag{2.2}$$

and representing the point by the triple $(X : Y : Z)$. Substituting these into the Weierstrass equation $y^2 = x^3 + Ax + B$ and clearing denominators gives the curve in

projective form,

$$Y^2Z = X^3 + AXZ^2 + BZ^3. \tag{2.3}$$

Explicit formulas for adding and doubling points in this representation are given by Washington [Was08, p. 42]. They are more involved than their affine counterparts. A point addition costs 12 multiplications and 2 squarings, and a doubling 7 multiplications and 5 squarings. Crucially, they involve no field inversions.

The projective representation is not unique. For any nonzero $\lambda \in \mathbb{F}_p^*$, the triples $(X : Y : Z)$ and $(\lambda X : \lambda Y : \lambda Z)$ describe the same affine point, because the common factor λ cancels in the ratios X/Z and Y/Z. A single affine point therefore corresponds not to one triple but to an entire *equivalence class*

$$(X : Y : Z) \;=\; \big\{\, (\lambda X : \lambda Y : \lambda Z) \;:\; \lambda \in \mathbb{F}_p^* \,\big\}, \tag{2.4}$$

containing $p - 1$ distinct representatives. The projective form of a point exists in such a class, where every representative is equally valid. As shown in Section 3.3, the *canonical lift* selects one standardized representative from the class, and comparing a leaked representative against this canonical one is what exposes information about the secret scalar.

To recover the affine point from any representative, we compute a single inversion, $(x, y) = (X/Z,\, Y/Z)$, exactly the operation anticipated in Section 2.1.2. This is the only inversion that the projective approach motivated above cannot avoid.

When scalar multiplication is carried out with the Montgomery ladder, the Y-coordinate is not needed and can be discarded entirely (Section 2.7). The point is then represented by the pair $(X : Z)$ with $x = X/Z$, and its equivalence class is $\{(\lambda X : \lambda Z) : \lambda \in \mathbb{F}_p^*\}$. This y-independent projective form is the representation used throughout the explanation of the monodromy leak attack in Chapter 3.

Finally, some implementations use a different projective system, the *Jacobian* representation, which is relevant to one of the countermeasures discussed later (see Section 6.4). Here the affine coordinates are recovered as $x = X/Z^2$ and $y = Y/Z^3$, so that two triples represent the same point exactly when

$$(X : Y : Z) \;=\; (\lambda^2 X : \lambda^3 Y : \lambda Z), \qquad \lambda \in \mathbb{F}_p^*. \tag{2.5}$$

The post-ladder masking countermeasure of Section 5.2.2, which re-randomizes an output by replacing (X, Y, Z) with $(\lambda^2 X, \lambda^3 Y, \lambda Z)$ for a random nonzero λ, is simply the choice of a fresh representative within this Jacobian equivalence class.

2.6 Point Compression

According to NIST, “Point compression allows for a shorter representation of elliptic curve points in affine coordinates by exploiting algebraic relationships between the

coordinate values ...". Furthermore, NIST states that point compression followed by its inverse operation "point decompression" [CMR+23, p. 45]. Explained simply, point compression involves representing curve coordinates in a shorter format. The algebraic relationship between the coordinate values that allow for point compression is generally the fact that the absolute value of the y-coordinate can be calculated from the x-value. Observe that elliptic curves can generally be brought to the form: $y^2 = f(x)$. This means that for any value of x there exists only two possible values of y, namely $\pm\sqrt{y^2}$.

For affine coordinates in $\mathbb{R}$, the general idea is to send information about the polarity of the y-coordinate without sending the whole value. A similar strategy can be performed for finite fields. Point compression effectively halves the key size at the cost of sending one more bit or byte for information about y. According to Standards for Efficient Cryptography (SEC), a practical procedure for performing **point compression** of an elliptic curve point is as follows [Sta09, p. 10]:

1. Create an output buffer of length L, where $L = \log_2(p) + 1$
2. Assuming that we are working over the field $\mathbb{K} = \mathbb{F}_p$ with $p > 2$ (other fields are also possible), we set a prefix as follows:
   a) If y is even, set the prefix to 0x02.
   b) If y is odd, set the prefix to 0x03.
3. Output prefix + x, encoded into bytes

In the same scheme, **point decompression** can be performed by:

1. Parse the compressed point into a prefix of one byte, and the rest as the x coordinate.
2. Calculate $y^2 = x^3 + Ax + B$
   a) If y^2 is not a square (in the field), consider the coordinates invalid (not on the curve)
3. Calculate the square root of y^2
   a) If the prefix is 0x02, output (x, y)
   b) If the prefix is 0x03, output $(x, -y)$

2.7 Y-independent Arithmetic

If one could create a system without the y-coordinate entirely, keys could be even smaller, and modular square roots would not need to be calculated. However, for working ECC we require that formulas for doubling and adding work, so specialized formulas for a y-independent system are needed. This section will show that when

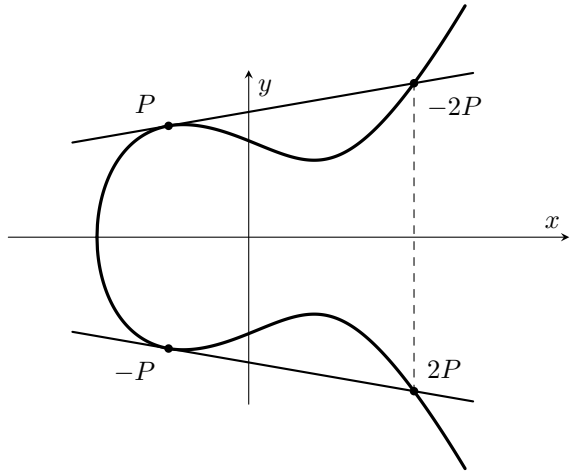


Figure 2.8: Doubling of points P and $-P$ on an elliptic curve defined over $\mathbb{R}$. The results $2P$ and $-2P$ have the same x-coordinate.

projective coordinates are used with the Montgomery ladder, working formulas for y-independent adding and doubling exist.

Observe that when performing a point doubling on a Weierstrass curve, the resulting x-coordinate will always be the same, no matter if the y-coordinate is positive or negative. For a visual example, see Figure 2.8.

When performing addition $P + Q$, the result will depend on the y-coordinate (see Figure 2.9). We know that $P + Q \neq P - Q$. Addition and subtraction on elliptic curves have different results, so to perform y-independent addition on elliptic curves, more information than only the x-coordinate is needed.

Peter L. Montgomery famously discovered Montgomery curves and the Montgomery ladder. In 1987 Montgomery would discover a link between these two concepts in the form of formulas for y-independent addition and doubling on Montgomery curves in projective space [Mon87, p. 261]. Montgomery's formulas for y-independent doubling on Montgomery curves are presented in Equation 2.6.

$$\begin{aligned} X_{P+P} &= (X_P + Z_P)^2 \cdot (X_P - Z_P)^2 \\ Z_{P+P} &= ((X_P + Z_P)^2 - (X_P - Z_P)^2) \cdot ((X_P - Z_P)^2 \\ &\quad + ((A+2)/4) \cdot ((X_P + Z_P)^2 - (X_P - Z_P)^2)) \\ &= 4X_P Z_P \cdot ((X_P - Z_P)^2 + ((A+2)/4) \cdot 4X_P Z_P) \end{aligned} \tag{2.6}$$

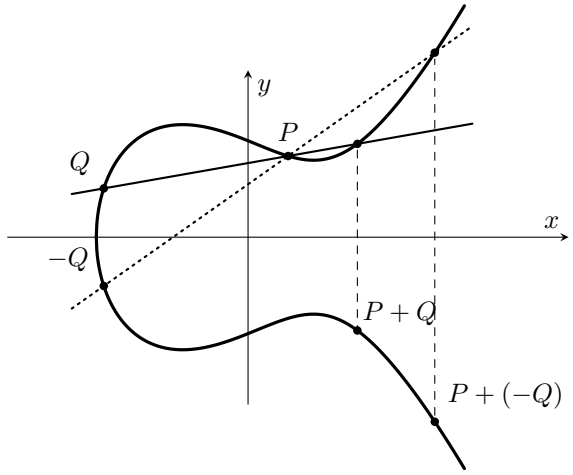


Figure 2.9: Additions $P + Q$ and $P + (-Q)$ on an elliptic curve defined over $\mathbb{R}$. The results have the different x-coordinates.

Here, X_P denotes the X-coordinate and Z_P denotes the Z-coordinate of point P. As explained earlier (Figure 2.8), formulas for doubling do not need information other than a point to be doubled. However this cannot be said for addition (Figure 2.9). Given a difference of points $D = P - Q$, we can calculate $P + Q$. Montgomery's formulas for y-independent addition on Montgomery curves [Mon87, p. 261], are given in Equation 2.7.

$$\begin{aligned} X_{P+Q} &= Z_D \cdot ((X_P - Z_P) \cdot (X_Q + Z_Q) + (X_P + Z_P) \cdot (X_Q - Z_Q))^2 \\ Z_{P+Q} &= X_D \cdot ((X_P - Z_P) \cdot (X_Q + Z_Q) - (X_P + Z_P) \cdot (X_Q - Z_Q))^2 \end{aligned} \tag{2.7}$$

It seems paradoxical that to calculate the sum $P + Q$ we need the difference $D = P - Q$. However, when the Montgomery ladder is used, the difference $P - Q$ is readily available. In the ladder (see Algorithm 2.2), $R_1 - R_0$ is always P. This means that when performing the addition in step 6 and step 9, we can use y-independent additions.

Listing 2.1 shows code written in SageMath that computes a scalar multiplication $[m]P$ on the curve25519 Montgomery curve using y-independent point compression. It uses the fastest formulas for the Montgomery ladder from the Explicit-Formulas Database by Daniel J. Bernstein and Tanja Lange [Exp26a; Exp26b]. These formulas are based on the original paper by Montgomery [Mon87]. The ladder works on projective points, and then returns the result to affine coordinates.

Listing 2.1: SageMath code that performs y-independent scalar multiplication on a curve25519 with the Montgomery ladder.

```
from sage.all import *
p = 2**255-19                 #curve25519
F = GF(p)
A = F(486662)
Gxz = (9 , 1)
m = randint(1, p - 1)

def double(a24, P):           #dbl-1987-m-3
    X1, Z1 = P

    A = X1+Z1
    AA = A*A
    B = X1-Z1
    BB = B*B
    C = AA-BB
    X3 = AA*BB
    Z3 = C*(BB+a24*C)
    return (X3, Z3)

def add(P, Q, D):             #dadd-1987-m-3
    X2, Z2 = P
    X3, Z3 = Q
    X1, Z1 = D

    A = X2+Z2
    B = X2-Z2
    C = X3+Z3
    D = X3-Z3
    DA = D*A
    CB = C*B
    X5 = Z1*(DA+CB)**2
    Z5 = X1*(DA-CB)**2
    return (X5, Z5)

def montgomery_ladder(P, b):
    a24 = (A + 2)/4
    R0 = (1, 0) #point at infinity in projective space
    R1 = P
    for bit in bin(b)[2:]: #remove 0b at start of string
        if bit == "0":
            R1 = add(R1, R0, P)
            R0 = double(a24, R0)
        else:
            R0 = add(R1, R0, P)
            R1 = double(a24, R1)
    return R0

mP_ladder  = montgomery_ladder(Gxz, m)
x_affine = mP_ladder[0]/mP_ladder[1] #X/Z
print(x_affine)
```

2.8 Using the CRT to Solve Modular Quadratics

Imagine we have a modular quadratic:

$$Ax^2 + Bx + C \equiv 0 \pmod{c}.$$

Suppose c is a large composite number that we can factor. Similar to how the CRT is used to speed up RSA private key operations by a factor of approximately four [MvOV97, Note 14.75], we can decompose the composite modulus into its prime power factors to solve the equation more efficiently. This approach replaces one computationally expensive problem with several smaller, faster sub-problems. The technique of solving modular quadratics in this way is applied in the monodromy leak attack in Section 3.8.

The prime power factorization of c is given by $c = \prod_{i=1}^{k} p_i^{e_i}$, where p_i are distinct primes. We can then solve the quadratic for each component:

$$\begin{aligned} Ax^2 + Bx + C &\equiv 0 \pmod{p_1^{e_1}} \\ Ax^2 + Bx + C &\equiv 0 \pmod{p_2^{e_2}} \\ &\vdots \\ Ax^2 + Bx + C &\equiv 0 \pmod{p_k^{e_k}} \end{aligned}$$

Each congruence yields a set of solutions $x \equiv r_{i,j} \pmod{p_i^{e_i}}$. The CRT is then used to *lift* these local solutions back to the original modulus. Since each of the k congruences typically contributes two local roots, there are up to 2^k such combinations $(r_1, r_2, \ldots, r_k)$, and the CRT lifts each one to a unique solution x modulo c, which gives up to 2^k solutions in total.

Chapter 3
Understanding the Monodromy Leak

This chapter reformulates the monodromy leak attack of Robert [Rob24], which recovers the entire secret scalar m from a single leaked projective coordinate. As noted in Chapter 1, the original paper explains the attack through advanced mathematical machinery that places it beyond the reach of most readers. The reformulation given here introduces each concept through the role it plays in the attack and is accompanied by SageMath code for a more empirical understanding.

The chapter develops the attack in stages. It begins with the cubical point and cubical arithmetic, which are *lifted* objects and operations on which the attack operates. Next comes the canonical lift, the standardization that makes two representatives of the same point comparable and moves the problem into $\mathbb{F}_p^*$. This concept is followed by the cubical ladder used to compute scalar multiples and canonical lifts. With these concepts in place, Section 3.5 establishes the crucial relationship between the scaling factors λ_1 and λ_2, and Section 3.6 gives a first, simplified version of the attack that assumes the Montgomery and cubical ladders coincide. They do not, and Section 3.7 quantifies the *deviance* between them. Finally, Section 3.8 combines these results into the complete attack.

3.1 Cubical Point

A cubical Point is in [Rob24, section 4.2.4] defined as a choice of rigidification of a line bundle, but this definition is outside the scope of this thesis. Recall that this thesis aims to explain the monodromy attack with simpler mathematical concepts, so more complex details will be left out.

A more colloquial definition is also mentioned in [Rob24, section 4.2.4]: "a cubical point $\tilde{P}$ is a 'point' lying above the projective point P, with a cubical arithmetic induced". For the scope of this thesis, a cubical point is defined as point which lies above another point on the elliptic curve, carrying extra information:

Definition 3.1. Let $(X : Y : Z) = P \in E$. A **cubical point** $\tilde{P}$ above P is defined as the triple

$$(X, Y, Z) \in (\mathbb{F}_p)^3.$$

We denote arbitrary cubical points as $\tilde{P}$, as opposed to $\hat{P}$ from Section 3.3 that is used specifically when the origin of the cubical point is from a canonical lift. Here $(X : Y : Z)$ denotes the projective representation of P introduced in Section 2.5.2, where it stands for an entire equivalence class of triples that differ by a common nonzero scaling factor. A cubical point instead fixes one specific representative (X, Y, Z) of that class. In Section 3.3, we explain how the *canonical lift* can lift a standard projective point P to a cubical point $\hat{P}$.

3.2 Cubical Arithmetic

Just as a cubical point is a *lifted* version of a projective point, cubical arithmetic is the *lifted* version of elliptic curve arithmetic. While the standard group law $(P, Q) \mapsto P + Q$ (see Section 2.2.2) is defined for projective points, cubical arithmetic acts on the triples (X, Y, Z) using formulas derived from the Riemann relations. A formal definition involves the *canonical isomorphism of line bundles*, but for the scope of this thesis, we can define it through comparison with standard elliptic curve arithmetic.

Definition 3.2. Let $\tilde{P}$ and $\tilde{Q}$ be cubical points above projective points $P, Q \in E$. **Cubical Arithmetic** is the set of algebraic operations that produce a new cubical point lying above a resulting projective point on the curve. These operations include:

1. **Cubical Doubling:** A function that maps a cubical point $\tilde{P}$ to a new triple $2\tilde{P}$ above the projective point $2P$.

2. **Cubical Differential Addition:** Given cubical points $\tilde{P}, \tilde{Q}$ and a choice of a cubical point above the difference $\tilde{P} - \tilde{Q}$, this operation computes a cubical point $\widetilde{P + Q}$ above $P + Q$.

3. **Cubical Scalar Multiplication:** For an integer m, the cubical point $[m]\tilde{P}$ is computed via the cubical ladder (see Section 3.4).

To understand what cubical arithmetic is in practice, it is useful to look at the concrete formulas that can be performed on the cubical points. Luckily, Robert has adapted the formulas in cubical arithmetic to the Montgomery model (with y-independent formulas) [Rob24, section 5.2]. This allows us to more easily compare

cubical arithmetic with the standard y-independent arithmetic (in Section 2.7). Cubical doubling in the Montgomery model [Rob24, Algorithm 5.4]:

$$\begin{aligned} X_{\tilde{P}+\tilde{P}} &= (X_{\tilde{P}} + Z_{\tilde{P}})^2 \cdot (X_{\tilde{P}} - Z_{\tilde{P}})^2 \\ Z_{\tilde{P}+\tilde{P}} &= ((X_{\tilde{P}} + Z_{\tilde{P}})^2 - (X_{\tilde{P}} - Z_{\tilde{P}})^2) \cdot ((X_{\tilde{P}} - Z_{\tilde{P}})^2 \\ &\quad + ((A+2)/4) \cdot ((X_{\tilde{P}} + Z_{\tilde{P}})^2 - (X_{\tilde{P}} - Z_{\tilde{P}})^2)) \\ &= 4X_{\tilde{P}}Z_{\tilde{P}} \cdot ((X_{\tilde{P}} - Z_{\tilde{P}})^2 + ((A+2)/4) \cdot 4X_{\tilde{P}}Z_{\tilde{P}}) \end{aligned}$$

Cubical addition in the Montgomery model [Rob24, Algorithm 5.5]:

$$X_{\tilde{P}+\tilde{Q}} = ((X_{\tilde{P}} + Z_{\tilde{P}})(X_{\tilde{Q}} - Z_{\tilde{Q}}) + (X_{\tilde{P}} - Z_{\tilde{P}})(X_{\tilde{Q}} + Z_{\tilde{Q}}))^2 \cdot 1/(4X_{\tilde{D}})$$

$$Z_{\tilde{P}+\tilde{Q}} = ((X_{\tilde{P}} + Z_{\tilde{P}})(X_{\tilde{Q}} - Z_{\tilde{Q}}) - (X_{\tilde{P}} - Z_{\tilde{P}})(X_{\tilde{Q}} + Z_{\tilde{Q}}))^2 \cdot 1/(4Z_{\tilde{D}})$$

Notice that cubical doubling is identical to Montgomery doubling (Equation 2.6), but that cubical addition is not the same (Equation 2.7).

3.3 Canonical lift

We have introduced cubical points and cubical arithmetic as *lying above* projective points and standard elliptic curve arithmetic. The concept of a canonical lift helps us understand what a point lifted above another point means. In mathematics as a whole, the term *canonical* refers to a normalization or a standard form. This is meant with a canonical lift as well.

Definition 3.3. Let $P \in E/\mathbb{F}_p$ be a point of order ℓ, and let $u \in \mathbb{Z}$ satisfy $u \equiv 0 \pmod{p-1}$ and $u \equiv 1 \pmod{\ell}$. We define the canonical lift as $\widehat{P} = [u]\tilde{P}$, where $\tilde{P}$ is any choice of cubical point above P.

This definition is based on [Rob24, Lemma 6.2]. More theoretical definitions of the canonical lift exist, but we deliberately choose a more practical one, as we aim to simplify the explanation.

3.3.1 Computing the Canonical Lift

To clarify the definition further, we will explain when the parameter u exists and how $[u]\tilde{P}$ is computed.

By the definition above, we can prove that u exists when $\gcd(\ell, p-1) = 1$. To see why this is the case, we note that the two conditions in the definition require u to be a multiple of $p-1$ while also being congruent to 1 modulo ℓ. If ℓ and $p-1$ shared a common factor $d > 1$, then every multiple of $p-1$ would also be $\equiv 0 \pmod{d}$, which contradicts the requirement $u \equiv 1 \pmod{d}$. If $\gcd(\ell, p-1) = 1$, no such conflict

arises. Because $p-1$ and ℓ are coprime, the CRT guarantees that a solution u exists and lets us compute it directly from the two congruences.

Crucially, as explained in Section 2.2.4, all secure curves satisfy $\gcd(\ell, p-1) = 1$, in order to be resistant against the MOV attack. This means that the canonical lift will be computable for the monodromy leak if the curve is considered otherwise secure.

The cubical multiplication $[u]\tilde{P}$ is then carried out with the cubical ladder (see Section 3.4), the cubical analogue of the Montgomery ladder for standard points. In SageMath:

```
def canonical_lift(P):
    u = crt([0,1], [p-1, ell])
    return cubical_ladder(P, u)
```

3.3.2 Usefulness of Canonical Lift

As mentioned, the canonical lift standardizes a point. More specifically, it standardizes any point in an equivalence class $(X : Y : Z)$ to one specific representative (X, Y, Z) (see Definition 3.1). This means we can use it to turn a standard projective point P into a cubical point $\hat{P}$. When we lift a point (P or $\tilde{P}$), we use the notation $\hat{P}$ to describe the resulting point. This is to differentiate it from an arbitrary cubical point $\tilde{P}$. Below, we will discuss how lifting P to $\hat{P}$ is used in the monodromy leak. First, we can illustrate that it removes the redundancy of an equivalence class by modeling the scenario as SageMath code that does the following:

1. Generate a random point $P = [k]G$ by multiplying a random number k with a generator point G. Here, G is the base point from *curve25519*, chosen as an example curve.
2. Create a list of random but equivalent points $Q_i = \lambda_i \cdot P$ for $i = 0, 1, .., n$ with random values for λ_i.
3. Compute the canonical lift of each point Q_i, $\hat{Q}_i$.
4. Return the number of unique Q_i and $\hat{Q}_i$.

Listing 3.1: SageMath Code which illustrates that a canonical lift is a standard representative of an equivalence class.

```
#curve25519 public parameters
p = 2**255-19
F = GF(p)
A = F(486662)
Gxz = (F(9) , F(1))
a24 = (A + 2)/4
ell = 2**252 + 27742317777372353535851937790883648493

# Generates n random points in the same equivalence class
def generate_Q_i_list(n):
    k = randint(1, p - 1)
    (P_x,P_z) = montgomery_ladder(Gxz, k) # Random point
    Q_i_list = []
    for i in range(n):
        # Choose a random integer from the range, excluding 0
        r = choice([i for i in range(-10000, 10001) if i != 0])
        Q_i_list.append((P_x * r, P_z * r))

    return Q_i_list

def canonical_lift(P):
    u = crt([0,1], [p-1, ell])
    return cubical_ladder(P, u)

Q_i_list = generate_Q_i_list(100)   #list of random equivalent points

Q_hat_i_list = []       #list of lifted points
for Q in Q_i_list:
    Q_hat = canonical_lift(Q)
    Q_hat_i_list.append(Q_hat)

print("Number of unique equivalent points: ", len(set(Q_i_list)))
# Output: 100

print("Number of unique canonically lifted points: ", len(set(
    Q_hat_i_list)))
# Output: 1
```

When the code is run, it shows that the number of unique projective points Q_i is generally equal to n, while the number of unique lifted points $\hat{Q}_i$ is 1. This illustrates clearly that random unique points in the same equivalence class are canonically lifted to the same point; $\hat{Q}_i = \hat{Q}_j$, for all i and all j.

In the monodromy leak attack, we assume that we have the base point of the curve G, as well as the leaked projective coordinates of the output of the Montgomery ladder, $[m]G$. A core part of the attack strategy is to compute the canonical lifts $\hat{G}$ and $\widehat{[m]G}$ as it allows us to compare the points to their canonical lifts. When we lift G to $\hat{G}$, we know that $\hat{G}$ still exists in the same equivalence class, meaning

there exists a λ_1 such that $G = \lambda_1 \hat{G}$. The same goes for $[m]G$ and $\widehat{[m]G}$, as the relationship can be expressed as $[m]G = \lambda_2 \cdot \widehat{[m]G}$. Furthermore, since we have chosen canonical representatives (as opposed to arbitrary representatives), there exists a relation between λ_1 and λ_2 which gives us information about m.

The security of elliptic curves relies on the ECDLP, preventing attackers from finding m. As described in Section 2.4, ECC uses shorter key sizes, as the ECDLP is harder to solve than the DLP in finite fields. When we set up an equation with λ_1, λ_2 and m, the attack scenario is moved from the elliptic curve to $\mathbb{F}_p^*$ while still having the smaller p from the elliptic curve. The canonical lift is crucial, as the comparison between G and $[m]G$, with $\hat{G}$ and $\widehat{[m]G}$ gives us λ_1 and λ_2, where the relation between λ_1 and λ_2 contains information about the secret scalar m (see Section 3.5). After having done the comparison, we can safely ignore the elliptic curve structure, as λ_1, λ_2 and m exist in $\mathbb{F}_p^*$.

3.4 Cubical Ladder

For any cubical point $\tilde{P}$, cubical arithmetic allows for scalar multiplication $[m]\tilde{P}$. Scalar multiplication is computed with the cubical ladder and is used to compute canonical lifts. The cubical ladder has a structure identical to the Montgomery ladder (see Algorithm 2.2), but uses cubical adds and cubical doublings [Rob24, Section 1.2]. Recall that cubical doublings are identical to standard point doubling in the Montgomery ladder, while cubical adds are different (Section 3.2).

Algorithm 3.1 Cubical ladder

Input: The point at infinity ∞, point P, scalar k
Output: $R_0 = kP$

$b \leftarrow$ bit representation of k (from MSB to LSB)
$R_0 \leftarrow \infty$
$R_1 \leftarrow P$
for each *bit* in b **do**
    if $bit = 0$ **then**
        $R_1 \leftarrow R_0 + R_1$ ▷ Cubical point addition
        $R_0 \leftarrow 2 \cdot R_0$ ▷ Cubical Point doubling
    else
        $R_0 \leftarrow R_0 + R_1$ ▷ Cubical point addition
        $R_1 \leftarrow 2 \cdot R_1$ ▷ Cubical Point doubling
    end if
end for
Return $R_0 = kP$

3.5 The Relation Between λ_1 and λ_2

In [Rob24, p. 77], it is shown that for λ_1 and λ_2 in $\tilde{G} = \lambda_1 \hat{G}$ and $[m]\tilde{G} = \lambda_2 \cdot \widehat{[m]G}$, we have the relation $\lambda_2 = \lambda_1^{m^2}$. A crucial detail mentioned by Robert is that this holds only for when $[m]\tilde{G}$ is the output of the cubical ladder (hence we use the notation $\tilde{G}$ instead of G). This means that this exact relation is not used in the complete attack on the Montgomery ladder. The reasons for the relation between λ_1 and λ_2 existing involve a good understanding of cubical theory, which is outside the scope of this thesis. To convince ourselves that the relation holds, the scenario can be modeled in SageMath:

Listing 3.2: Code which illustrates that the relation between λ_1 and λ_2 is as described when the cubical ladder is used for multiplication.

```
#curve25519 public parameters
p = 2**255-19
F = GF(p)
A = F(486662)
Gxz = (F(9) , F(1))
a24 = (A + 2)/4

G_ord = 2**252 + 27742317777372353535851937790883648493

G_hat = canonical_lift(Gxz)
lambda_1 = Gxz[1]/G_hat[1]   # Z_G / Z_G_hat

#random point from Generator
m = randint(1, p - 1)
mG = cubical_ladder(Gxz, m)

mG_hat = canonical_lift(mG)
lambda_2 = mG[1]/mG_hat[1] # Z_mG / Z_mG_hat

assert lambda_2 == lambda_1**(m**2 % (p-1))
print("\n\lambda_2 = \lambda_1 ** (m**2)")
```

1. Again, we use the base point $\tilde{G}$ from *curve25519*, chosen as an example curve. The code calculates $\hat{G}$ and $\lambda_1 = Z_{\tilde{G}}/Z_{\hat{G}}$.
2. Next, a random value m is chosen and used for calculating a new point $[m]\tilde{G}$. The scalar multiplication is calculated with the cubical ladder.
3. $[m]\tilde{G}$ is canonically lifted into $\widehat{[m]G}$, and $\lambda_2 = Z_{[m]\tilde{G}}/Z_{\widehat{[m]G}}$ is calculated.
4. Lastly the code asserts that the relation $\lambda_2 = \lambda_1^{m^2}$ holds.

3.6 Illustrative Attack

As explained in Section 3.5, the relation between λ_1 and λ_2 does not exists for an $[m]G$ calculated by the Montgomery ladder. The reason for this relation not existing is that the addition formulas for addition in the cubical and Montgomery ladders are not the same (see Section 3.2). We can say that the Montgomery ladder does *not coincide* with the cubical ladder.

In our attack scenario we are given the projective coordinates of $[m]G$ from the Montgomery ladder. We will in this illustrative attack pretend the Montgomery ladder and the cubical ladder coincides, and that the relation between λ_1 and λ_2 does exist for output from the Montgomery ladder $[m]G$. In addition to $[m]G$, we are also given the curve parameters and the base point G. The strategy is as follows:

1. Compute $\hat{G}$, the canonical lift of G.
2. We have the identity $G = \lambda_1 \cdot \hat{G}$. It follows that λ_1 can be computed as $\lambda_1 = Z_G/Z_{\widehat{G}}$.
3. Next, compute $\widehat{[m]G}$, the canonical lift of the leaked coordinates $[m]G$.
4. For this other canonical lift, we have the identity $[m]G = \lambda_2 \cdot \widehat{[m]G}$. This means that $\lambda_2 = Z_{[m]G}/Z_{\widehat{[m]G}}$.
5. Lastly, in this attack we pretend that $\lambda_2 = \lambda_1^{m^2}$. From this expression we obtain:
$$\log_{\lambda_1}(\lambda_2) = m^2.$$
This discrete log is not hard to compute, since we are no longer working on the curve, but in $\mathbb{F}_p^*$ (see Section 3.3.2). We can calculate the secret scalar m mod $p-1$ out of the solutions to
$$m = \sqrt{m^2} \mod (p-1).$$

3.7 The Ladder Deviance

As discussed above, the Montgomery ladder does not coincide with the cubical ladder. In this section, we will quantify the deviance between the two ladders by looking at how the addition formulas are different, and how the deviance in the ladders grows for each subsequent operation after an addition. To look at the difference in the addition formulas, we can define the variables:

$$u = (X_{\tilde{P}} + Z_{\tilde{P}})(X_{\tilde{Q}} - Z_{\tilde{Q}}),$$

$$v = (X_{\tilde{P}} - Z_{\tilde{P}})(X_{\tilde{Q}} + Z_{\tilde{Q}}).$$

Cubical add adopted to the Montgomery model (algorithm 5.5 in [Rob24]):

$$X_{C_{\tilde{P}+\tilde{Q}}} = (u+v)^2/(4X_D),$$
$$Z_{C_{\tilde{P}+\tilde{Q}}} = (u-v)^2/(4Z_D).$$

Montgomery ladder add from Equation 2.7, rewritten to be more comparable with the cubical add through variables u and v:

$$X_{M_{\tilde{P}+\tilde{Q}}} = Z_D \cdot (u+v)^2,$$
$$Z_{M_{\tilde{P}+\tilde{Q}}} = X_D \cdot (u-v)^2.$$

The deviance in the add operation between the ladders is therefore:

$$\frac{X_{M_{\tilde{P}+\tilde{Q}}}}{X_{C_{\tilde{P}+\tilde{Q}}}} = \frac{Z_D \cdot (u+v)^2}{(u+v)^2/(4X_D)} = 4X_D Z_D,$$
$$\frac{Z_{M_{\tilde{P}+\tilde{Q}}}}{Z_{C_{\tilde{P}+\tilde{Q}}}} = \frac{X_D \cdot (u-v)^2}{(u-v)^2/(4Z_D)} = 4X_D Z_D.$$

Remember that $D = \widetilde{P-Q}$ and that in the ladder structure for scalar multiplication $D = \tilde{G}$, where $\tilde{G}$ is the base point (see Section 2.7). Note that base points generally are of the form $\tilde{G} = (X_{\tilde{G}}, Z_{\tilde{G}}) = (x, 1)$. It is therefore fair to assume that $Z_D = 1$. From here on this thesis will assume $Z_D = 1$ and that the deviance between the ladders therefore is $4X_{\tilde{G}}$ for both the X and Z coordinate.

Since the point addition formulas are the only non-identical part of the ladders, one could assume that the relation between the output of the ladders is now known. This is, however, not the case. Imagine that an add is performed at some step in the middle of the ladders and that $4X_{\tilde{G}}$ is added to the deviance between the ladders. We can call $4X_{\tilde{G}}$ a *contribution* to the deviance. In the subsequent steps, this specific contribution will be modified (grow) when doubling and addition occur. In the next section, we will look at what happens to an arbitrary contribution $4X_{\tilde{G}}$. Later we will get a bigger picture of what happens for the whole ladder by tracking how many contributions we add to the deviance, and what happens to each contribution $4X_{\tilde{G}}$ throughout the ladder after it is added.

3.7.1 Tracking Contributions Throughout the Ladder

In [Rob24], the formulas for tracking how the $4X_{\tilde{G}}$ impact the deviance of the ladders are retrieved from [BGS22]. To explain these formulas, it first helps to look at what happens to any scalar λ that appears at the input for the add and double formulas in

the Montgomery ladder. Suppose that we have a point $P' = \lambda_P \cdot P = (\lambda_P X_P, \lambda_P Z_P)$. What happens to P' when it is doubled in the Montgomery ladder (Equation 2.6)?

$$\begin{aligned}
X_{P'+P'} &= (\lambda_P X_P + \lambda_P Z_P)^2 \cdot (\lambda_P X_P - \lambda_P Z_P)^2, \\
X_{P'+P'} &= \lambda_P^4 \cdot (X_P + Z_P)^2 \cdot (X_P - Z_P)^2, \\
X_{P'+P'} &= \lambda_P^4 \cdot X_{P+P}.
\end{aligned}$$

$$\begin{aligned}
Z_{P'+P'} &= (4\lambda_P X_P \lambda_P Z_P \cdot ((\lambda_P X_P - \lambda_P Z_P)^2 + (((A+2)/4) \cdot 4\lambda_P X_P \lambda_P Z_P), \\
Z_{P'+P'} &= (\lambda_P^2 \cdot 4 X_P Z_P \cdot ((X_P - Z_P)^2 \cdot \lambda_P^2 + (((A+2)/4) \cdot \lambda_P^2 4 X_P Z_P), \\
Z_{P'+P'} &= \lambda_P^4 (4 X_P Z_P \cdot ((X_P - Z_P)^2 + (((A+2)/4) \cdot 4 X_P Z_P) \\
Z_{P'+P'} &= \lambda_P^4 \cdot (Z_{P+P}).
\end{aligned}$$

This shows that when a scalar λ is introduced into the doubling formula, it is raised to the fourth power. For addition, we look at the formulas in Equation 2.7, and use $P' = \lambda_P \cdot P$ and $Q' = \lambda_Q \cdot Q$:

$$\begin{aligned}
X_{P'+Q'} &= Z_D \cdot ((\lambda_P X_P - \lambda_P Z_P) \cdot (\lambda_Q X_Q + \lambda_Q Z_Q) + (\lambda_P X_P + \lambda_P Z_P) \cdot (\lambda_Q X_Q - \lambda_Q Z_Q))^2, \\
X_{P'+Q'} &= \lambda_P^2 \lambda_Q^2 \cdot Z_D \cdot ((X_P - Z_P) \cdot (X_Q + Z_Q) + (X_P + Z_P) \cdot (X_Q - Z_Q))^2, \\
X_{P'+Q'} &= \lambda_P^2 \lambda_Q^2 \cdot X_{P+Q}.
\end{aligned}$$

$$\begin{aligned}
Z_{P'+Q'} &= X_D \cdot (\lambda_P X_P - \lambda_P Z_P) \cdot (\lambda_Q X_Q + \lambda_Q Z_Q) - (\lambda_P X_P + \lambda_P Z_P) \cdot (\lambda_Q X_Q - \lambda_Q Z_Q))^2, \\
Z_{P'+Q'} &= \lambda_P^2 \lambda_Q^2 \cdot X_D \cdot (X_P - Z_P) \cdot (X_Q + Z_Q) - (X_P + Z_P) \cdot (X_Q - Z_Q))^2, \\
Z_{P'+Q'} &= \lambda_P^2 \lambda_Q^2 \cdot Z_{P+Q}.
\end{aligned}$$

We should also not forget that when an add is performed, we are adding $4X_{\tilde{G}}$ to the deviance (comparing with the cubical ladder). We can therefore conclude that for addition, $P' + Q'$ will result in the deviance increasing to $\lambda_P^2 \lambda_Q^2 + 4X_{\tilde{G}}$ in both the X and Z coordinate.

Now that we have tracked what happens to arbitrary scalars in the double and add operations, we can get a picture of how the whole ladder operates. Recall that the ladder structure uses variables R0 and R1. Let $\mathrm{R0}_{M_i}$ and $\mathrm{R1}_{M_i}$ denote the variables used when bit number i is processed in the Montgomery ladder, and let $\mathrm{R0}_{C_i}$ and $\mathrm{R1}_{C_i}$ denote the variables at bit number i in the cubical ladder. We can express the accumulated difference at bit number i as:

$$\frac{\mathrm{R0}_{M_i}}{\mathrm{R0}_{C_i}} = \lambda_{R0_i} \qquad \frac{\mathrm{R1}_{M_i}}{\mathrm{R1}_{C_i}} = \lambda_{R1_i} \tag{3.1}$$

From studying what happens to arbitrary λ-values in the add and double formulas, we can tell that any accumulated difference λ_{R0_i} and λ_{R1_i} will become $\lambda_{R0_i}^4$ and

$\lambda^4_{R1_i}$ after a double, and that when $\mathrm{R0}+\mathrm{R1}$ is computed at bit number i, the output will have difference $\lambda^2_{R0_i}\lambda^2_{R1_i} + 4X_{\tilde{G}}$. Summarized:

For double:

$$\frac{2\mathrm{R0}_{M_i}}{2\mathrm{R0}_{C_i}} = \lambda^4_{R0_i} \qquad \frac{2\mathrm{R1}_{M_i}}{2\mathrm{R1}_{C_i}} = \lambda^4_{R1_i} \tag{3.2}$$

For add:

$$\frac{\mathrm{R0}_{M_i} + \mathrm{R1}_{M_i}}{\mathrm{R0}_{C_i} + \mathrm{R1}_{C_i}} = \lambda^2_{R0_i}\lambda^2_{R1_i} + 4X_{\tilde{G}} \tag{3.3}$$

Notice that the deviance between the two ladders will always be a power of $4X_{\tilde{G}}$, since every contribution is $4X_{\tilde{G}}$, and every contribution scales to be some power of $4X_{\tilde{G}}$. The number of doubles and adds depends on the secret m, so the power of $4X_{\tilde{G}}$ is also dependent on m.

3.7.2 The Ladder Deviance Formulas

Robert mentions in [Rob24, p. 79] that "... by [BGS22, Proposition 2] $[m\tilde{P}] = (4X_P)^{m2^{\mathrm{len}(m)}-m}m\tilde{P}$ where $\mathrm{len}(m)$ is the binary length of m". In Roberts notation, $[m\tilde{P}]$ denotes the cubical point computed by the Montgomery ladder, and $m\tilde{P}$ denotes the 'true' cubical point from the cubical ladder. In our notation this refers to $[m]P$ and $[m]\tilde{P}$ respectively. An important observation to be made after reading [BGS22, Proposition 2] is that Robert has by mistake not included a pair of parenthesis in the above expression. The correct expression for the deviance between the Montgomery and cubical ladders that will be proved below is:

$$[m]P = 4X_{\tilde{G}}{}^{m(2^{\mathrm{len}(m)}-m)} \cdot [m]\tilde{P} \tag{3.4}$$

[BGS22] does not concern or mention cubical ladders or cubical arithmetic, however Robert mentions in his article [Rob24] that readers should "see Section 6.3 for the link between division polynomials in the statement of [BGS22, Proposition 2], and cubical arithmetic". Unfortunately, [Rob24, section 6.3] contains mathematical concepts considered out of scope for this thesis. Luckily, the link between these concepts is not needed in order to explain the correctness of Equation 3.4.

To summarize the findings of [BGS22, Proposition 2] adapted to our current scenario of comparing the Montgomery and cubical ladder, we can express the equation with ladder variables $\mathrm{R0}_M$, $\mathrm{R1}_M$, $\mathrm{R0}_C$ and $\mathrm{R1}_C$ as the final values in their respective ladders.

$$\mathrm{R0}_M = 4X_{\tilde{G}}{}^{m(2^{\mathrm{len}(m)}-m)} \cdot \mathrm{R0}_C,$$

$$\mathrm{R1}_M = 4X_{\tilde{G}}{}^{(m+1)(2^{\mathrm{len}(m)}-(m+1))} \cdot \mathrm{R1}_C.$$

Furthermore, [BGS22, Proposition 2] uses the following functions to prove this statement:

$$f_m(x) = (4x)^{F_m} \quad \text{and} \quad g_m(x) = (4x)^{G_m} \quad \text{where} \quad \begin{cases} F_m := m(2^{\text{len}(m)} - m), \\ G_m := (m+1)(2^{\text{len}(m)} - (m+1)). \end{cases}$$

We see that $f_m(x)$ is equal to the deviance $\lambda_{R0} = \text{R0}_M/\text{R0}_C$, and $g_m(x)$ is equal to $\lambda_{R1} = \text{R1}_M/\text{R1}_C$ (see Equations 3.1). To prove that $f_m(x)$ and $g_m(x)$ indeed show the real difference between the ladders, we must show that for any m:

$$\begin{aligned} f_{2m}(x) &= f_m^4(x), \\ g_{2m+1}(x) &= g_m^4(x), \\ f_{2m+1}(x) &= 4x \cdot f_m^2(x) g_m^2(x). \end{aligned} \tag{3.5}$$

By showing that these equations hold, we show that $f_m(x)$ and $g_m(x)$ behave the same way as any λ going through adds and doubles (Equations 3.3 and 3.2). Observe that $f_{2m}(x)$ is the doubled point of $f_m(x)$, as if we double $\text{R0} = [m]G$, it becomes $2\text{R0} = [2m]G$, meaning doubling R0 equivalent to doubling m. The same goes for $g_{2m+1}(x)$, as $g_{2m+1} = f_{2m+2}$ which corresponds to $2(\text{R0} + G) = 2\text{R1}$. We can also see that $f_{2m+1}(x)$ corresponds to $\text{R0} + \text{R1}$, as R0 contains $[m]G$, and R1 contains $[m+1]G$.

3.7.3 Proof of Deviance

To prove that the Equations 3.5 hold for all m, [BGS22] shows that a base case with $m = 1$ holds, and uses induction to prove further correctness for $m > 1$.

Base Case

It suffices to show that for $m = 1$, $f_1(x) = \lambda_{\text{R0}}$, $g_1(x) = \lambda_{\text{R1}}$, where the λ-values denote the final deviances between the Montgomery and cubical ladders. The bit representation of m is $(1)_2$, so the ladders will compute $\text{R0} = \text{R1} + \text{R0}$ and $\text{R1} = 2\text{R1}$ (see algorithm 2.2). With the input of $\text{R0} = (1, 0)$, $\text{R1} = (x, 1)$ (base point assumed to be on the form $G = (x, 1)$) the ladders will compute:

Cubical ladder:

$$\begin{aligned} \text{R0}_C &:= \text{R1}_C + \text{R0}_C = (x, 1), \\ \text{R1}_C &:= 2\text{R1}_C = (x^4 - 2x^2 + 1, 4x^3 + 8x^2 \cdot ((A+2)/4) + 4x), \end{aligned}$$

Montgomery ladder:

$$\begin{aligned} \text{R0}_M &:= \text{R1}_M + \text{R0}_M = (4x^2, 4x), \\ \text{R1}_M &:= 2\text{R1}_M = (x^4 - 2x^2 + 1, 4x^3 + 8x^2 \cdot ((A+2)/4) + 4x). \end{aligned}$$

The deviances between the two ladders (in both X and Z coordinates) are:

$$\lambda_{\mathrm{R0}} = \frac{\mathrm{R0}_M}{\mathrm{R0}_C} = 4x, \qquad \lambda_{\mathrm{R1}} = \frac{\mathrm{R1}_M}{\mathrm{R1}_C} = 1.$$

This matches up with what we expect, as $\lambda_{\mathrm{R0}} = f_1(x) = 4x^{1(2^{\mathrm{len}(1)}-1)} = 4x$, and $\lambda_{\mathrm{R1}} = g_1(x) = 4x^{(1+1)(2^{\mathrm{len}(1)}-(1+1))} = 1$.

Inductive Proof

Firstly, to prove $f_{2m}(x) = f_m^4(x)$, it suffices to show that $F_{2k} = 4F_k$, as $f_m(x) = 4x^{F_m}$:

$$\begin{aligned} F_{2k} &= 2k(2^{\mathrm{len}(2k)} - 2k) \\ &= 2k(2 \cdot 2^{\mathrm{len}(k)} - 2k) \\ &= 4k(2^{\mathrm{len}(k)} - k) \\ &= 4F_k. \end{aligned}$$

Remember that $\mathrm{len}(2k) = \mathrm{len}(k) + 1$ (len is the binary length). To prove $g_{2m+1}(x) = g_m^4(x)$, it suffices to show that $G_{2k+1} = 4G_k$:

$$\begin{aligned} G_{2k+1} &= (2k+2)(2^{\mathrm{len}(2k)} - (2k+2)) \\ &= 2(k+1)(2 \cdot 2^{\mathrm{len}(k)} - 2(k+1)) \\ &= 4(k+1)(2^{\mathrm{len}(k)} - (k+1)) \\ &= 4G_k. \end{aligned}$$

Lastly, we need to show that the add operation holds, meaning $f_{2m+1}(x) = 4x \cdot f_m^2(x) g_m^2(x)$:

$$\begin{aligned} F_{2k+1} = G_{2k} &= (2k+1)(2^{\mathrm{len}(2k)} - (2k+1)) \\ &= 2(2k+1)(2^{\mathrm{len}(k)} - 2k) - 2k - 1 \\ &= 4k \cdot 2^{\mathrm{len}(k)} - 4k^2 + 2 \cdot 2^{\mathrm{len}(k)} - 2k - 2k - 2 + 1 \\ &= 2k(2^{\mathrm{len}(k)} - k) + 2k2^{\mathrm{len}(k)} - 2k^2 + 2 \cdot 2^{\mathrm{len}(k)} - 4k - 2 + 1 \\ &= 2F_k + 2^{\mathrm{len}(k)}(2k+2) + (-2k - 4k - 2) + 1 \\ &= 2F_k + 2^{\mathrm{len}(k)} \cdot 2(k+1) - 2(k+1)(k+1) + 1 \\ &= 2F_k + 2(k+1)(2^{\mathrm{len}(k)} - (k+1)) + 1 \\ &= 2F_k + 2G_k + 1. \end{aligned}$$

If $F_{2k+1} = 2F_k + 2G_k + 1$, then $f_{2m+1}(x) = 4x^{2F_k + 2G_k + 1} = 4x \cdot f_m^2(x) g_m^2(x)$. This is what we expected from the add formula. Combining this proof with the base case for $m = 1$, we see that $f_m(x)$ and $g_m(x)$ accurately model deviance between the ladders in R0 and R1 for all m. To convince ourself further, the scenario can be modeled as SageMath code. The code below asserts that $\lambda_{\mathrm{R0}} = f_m(x)$ and $\lambda_{\mathrm{R0}} = g_m(x)$ for different values of m.

Listing 3.3: SageMath code that tests the deviance formulas between the ladders.

```
# A, x and P defined symbolically
A = F(A)
x = F(x)
P = (x, F(1))

def Fm(m):
    return m * (2**(m.bit_length()) - m)

def Gm(m):
    return (m + 1) * (2**(m.bit_length()) - (m + 1))

def test_m(m):
    R0_m, R1_m = montgomery_ladder(P, m)
    R0_c, R1_c = cubical_ladder(P, m)

    lambda_R0 = (R0_m[0]/R0_c[0] , R0_m[1]/R0_c[1])
    lambda_R1 = (R1_m[0]/R1_c[0] , R1_m[1]/R1_c[1])

    print(f"{"="*100}\n For m = {m}, the difference is:\nlambda_R0 = {
        lambda_R0}\nlambda_R1 = {lambda_R1}")

    fm = F(4)**Fm(m) * x**Fm(m) # f_m(x) = 4x^(F_m)
    gm = F(4)**Gm(m) * x**Gm(m) # g_m(x) = 4x^(G_m)

    print(f"\nExpected from m formulas:\nf_m(x): {fm}\ng_m(x): {gm}")
    assert lambda_R0[0] == fm
    assert lambda_R0[1] == fm
    assert lambda_R1[0] == gm
    assert lambda_R1[1] == gm
    print("Passed for m = "+ m +"!")

for m in range(1,10):
    test_m(m)
```

3.8 The Monodromy Leak Attack

Assume again that we are given the curve parameters, a base point G and the leaked projective coordinates from the output of the Montgomery ladder, $[m]G = (X_{[m]G}, Z_{[m]G})$. Notice that the first four steps are almost identical to the procedure in the illustrative attack.

1. Calculate the canonical lift of G: $\hat{G}$

2. We have the identity $G = \lambda_1 \cdot \hat{G}$. From this equation λ_1 can be expressed as $\lambda_1 = Z_G / Z_{\widehat{G}}$

3. Next, compute the canonical lift of the leaked projective coordinate $[m]G$: $\widehat{[m]G}$.

4. For every canonical lift, there is a linear relation between a lifted and non lifted variables:
$$[m]G = \lambda_2 \cdot \widehat{[m]G}$$
Again, $\lambda_2 = Z_{[m]G} \ / \ Z_{\widehat{[m]G}}$

5. From Section 3.5 we know that the relation between λ_1 and λ_2 must involve the term $\lambda_1^{m^2}$, as is the case when a cubical ladder is used. From Section 3.7 we known the deviance between the Montgomery ladder and the cubical ladder is $4X_{\tilde{G}}^{m(2^{\text{len}(m)}-m)}$, so the total deviance between $[m]G$ and $\widehat{[m]G}$ can be expressed:
$$\lambda_2 = 4X_{\tilde{G}}^{m(2^{\text{len}(m)}-m)} \cdot \lambda_1^{m^2}$$
Where $\text{len}(m)$ is the bit length of m [Rob24, equation 22].

6. To solve for m, we choose a generator γ of $\mathbb{F}_p^*$ and take the logarithm base γ of the equation above.
$$\begin{aligned}
\log_\gamma(4X_{\tilde{G}}^{m(2^{\text{len}(m)}-m)} \cdot \lambda_1^{m^2}) &\equiv \log_\gamma(\lambda_2) \mod (p-1) \\
m(2^{\text{len}(m)} - m) \cdot \log_\gamma(4X_{\tilde{G}}) + m^2 \log_\gamma(\lambda_1) &\equiv \log_\gamma(\lambda_2) \mod (p-1) \\
m^2 \cdot (\log_\gamma(\lambda_1) - \log_\gamma(4X_{\tilde{G}})) + m2^{\text{len}(m)} \cdot \log_\gamma(4X_{\tilde{G}}) &\equiv \log_\gamma(\lambda_2) \mod (p-1)
\end{aligned}$$

7. Since we are working in the multiplicative finite field $\mathbb{F}_p^*$, the logarithms in the above equation are not hard to compute when the p is of order typical to that of elliptic curves. Let $\log_\gamma(4X_{\tilde{G}}) = d_x$, $\log_\gamma(\lambda_1) = d_{\lambda_1}$ and $\log_\gamma(\lambda_2) = d_{\lambda_2}$. We are left with the equation:
$$m^2(d_{\lambda_1} - d_x) + m(d_x 2^{\text{len}(m)}) - d_{\lambda_2} \equiv 0 \mod (p-1)$$

8. Finally, to solve for m we can follow the steps in Section 2.8. We split $p-1$ into factors $p_i^{e_i}$ for $i \in 1, 2, \ \ldots \ , k$. We set $A = d_{\lambda_1} - d_x$, $B = d_x 2^{\text{len}(m)}$ and $C = -d_{\lambda_2}$. Observe that $\text{len}(m)$ is not known and all possible values must be tested. Luckily m is bounded by ℓ, the order of the generator. We can construct k equations on the form:
$$m^2 A + mB + C \equiv 0 \mod (p_i^{e_i})$$
When the composite $p-1$ is split up this way, the total computational cost for the k congruences is lower than calculating the solution for m outright [MvOV97, Note 14.75]. Each congruence has a set of solutions $r_{i,j}$. The CRT is used to combine all solutions, finding the unique solution
$$m \mod p - 1.$$

Appendix A provides a SageMath implementation of the monodromy leak attack on Curve25519, following the exact procedure described above.

Chapter 4
Methodology

4.1 Research Goal

One of the objectives of this thesis is to evaluate the applicability of the monodromy leak attack in practice, as outlined in Research Question 6 in the introduction:

Research Question 6. Which widely used ECC libraries implement projective coordinates, and to what extent are they vulnerable to Robert's attack under realistic operational scenarios?

To answer Research Question 6, we have conducted a systematic review of popular open source libraries. Static analysis was selected as the evaluation method for each library. The applicability of the monodromy leak attack depends on preconditions and countermeasures, such as the use of projective coordinates and constant time division. These are directly visible in the source code, and do not require the code to be executed to be assessed, hence static analysis was seen as sufficient.

Open source libraries were chosen for several reasons. Firstly, their source code is publicly available, which is a prerequisite of the static analysis that will be performed. Secondly, open source libraries tend to be widely deployed, especially cryptographic libraries where security through obscurity is not preferred. Lastly, since the code is publicly available, the findings of this review are reproducible and verifiable by others.

In summary, the relevant scalar multiplication functions in each open source library were evaluated using static code analysis against a set of criteria related to attack preconditions and countermeasures (see Section 5.3), with our findings summarized in Section 5.4.

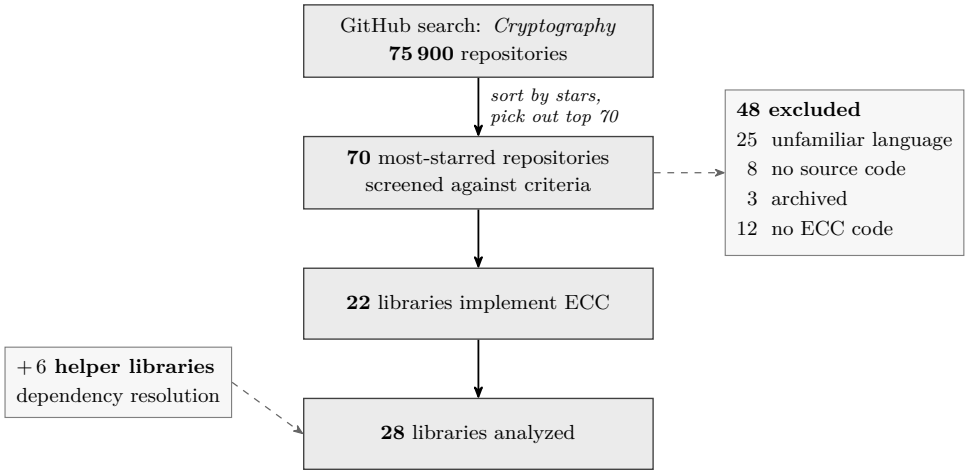


Figure 4.1: Overview of the library selection process. The *Cryptography* search on GitHub returned 75 900 repositories, of which the 70 most-starred were screened against the criteria in Section 4.2.3. After excluding 48 repositories, 22 libraries implemented ECC. Resolving dependencies (Section 4.2.4) added 6 helper libraries, giving the 28 libraries (69 scalar-multiplication routines, Table 5.2) analyzed in Chapter 5.

4.2 Library Selection

This section outlines the process of selecting the libraries to be analyzed. It describes how the programming languages were chosen, how popular open source libraries were found on GitHub, how the results were filtered to relevant candidates, and how dependencies were resolved. An overview of the final selection process is illustrated in Figure 4.1.

4.2.1 Language Scope

Before searching for popular open source libraries, we first decided which programming languages to analyze. Firstly, a practical restriction was that not all programming languages were familiar. Considering the time it takes to become familiar with a new programming language, familiar programming languages were preferred. The other restriction was empirical. Since the focus of research was on popular libraries, the popularity of each programming language and its most popular library was considered (metrics for popularity are described below). To understand which programming languages were the most viable in the empiric sense, a quick popularity survey was performed (see Table 4.1). A search for *Cryptography* on GitHub was the basis of

Share of *Cryptography*			Most popular library (stars)	
Language	**Results**	**Share**	**Stars**	**Name**
Python	22,200	29.25%	9,900	OpenMinded
C/C++	9,300	12.25%	88,800	Bitcoin
Java	6,400	8.43%	14,900	Cryptomator
JavaScript	4,500	5.93%	34,100	The Algorithms
Rust	3,000	3.95%	21,300	Ciphey
C#	2,700	3.56%	2,400	DNSCrypt
TypeScript	2,500	3.29%	16,000	Javascript Obfuscator
Golang	2,000	2.64%	13,800	OpenNHP
PHP	649	0.86%	8,200	PrivateBin
Kotlin	459	0.60%	2,000	ToolsFx

Table 4.1: GitHub repositories in the search for *Cryptography* [Git26b] by language, and the most-starred library per language. The search was performed 13 April 2026, with 75,900 results. The 10 most popular languages are shown.

the data. The same search would eventually be used as the main source for libraries as well.

Considering the share of the *Cryptography* search and the largest projects, the most dominant languages are Python, C/C++, Rust, JavaScript, and Java. Of these, the most familiar languages were Python, Java and JavaScript, and so they were natural choices for programming languages to include. Apart from these, C/C++ was the most popular choice, and was chosen as the fourth and final language to include. Rust was discarded due to the unfamiliarity with the language. C#, TypeScript, Golang, PHP and Kotlin were discarded due to not being popular enough. Overall, choosing four languages seemed like a good fit for the scope of a master thesis, considering the time constraint. The languages chosen were:

- Python, due to familiarity
- C/C++, due to popularity
- JavaScript, due to familiarity
- Java, due to familiarity

4.2.2 Source and Search Strategy

GitHub was chosen as the primary source for finding libraries. As of 2025, GitHub had 512 million open source projects, and is widely regarded as the largest open source platform in the world [Git25]. As such, it is a good fit for the scope of this project.

The number of stars on a repository page was chosen as a metric for popularity because it generally provides a rough measure of community interest. Even though stars are not a perfect metric because they reflect attention more than deployment and real world usage, we assume that highly starred libraries are more likely to be widely used in practice.

To exclude non-cryptographic libraries from our GitHub search, we ultimately selected the search term *Cryptography* [Git26b]. However, before settling on this approach, other strategies were first tried. A straightforward search for *Elliptic Curve Cryptography* resulted in 3,6 thousand results, and the most popular library had 3,8 thousand stars. From these numbers we can observe that the result excluded many popular and relevant libraries that use ECC, such as `Bitcoin` from Table 4.1.

We also tried browsing topics on GitHub, which are keywords with # present on library pages. The reasons for trying topics were that unlike searching, where the quality of the results depends on the efficiency of the search algorithm, the results of browsing topics are more discrete, either a tag is on the profile or it is not. GitHub topics can be added by the creator of a library, and it is easy to assume that creators would be better at classifying their own projects than a search algorithm. However, since adding topic-tags to repositories is optional, important ECC libraries with a considerable number of stars such as `Mbed-TLS` and `Signalapp` appeared in the *Cryptography* search but were missing from the *#Cryptography* [Git26a] topic. Consequently, browsing by topics was considered inferior for our use case.

Ultimately, among the most starred libraries, the *#Cryptography* topic had only a subset of the results from the *Cryptography* search. Because of broader results, the search for *Cryptography* was chosen as the search strategy for this library review. The results were sorted by number of stars, and the 70 most starred libraries were chosen for further analysis. The cutoff at 70 libraries was motivated by practical constraints, as analyzing each library requires a considerable effort. However, since libraries are ranked by popularity, the most relevant candidates were considered first, and results beyond the cutoff are less likely to be widely used in practice. Next, we detail how these results were filtered to match the scope of this project.

4.2.3 Initial Screening

To discard irrelevant libraries in an effective manner before starting time-consuming manual analysis, the list was processed with specific criteria. First, we excluded projects not written in `Python`, `C/C++`, `JavaScript` and `Java`. Second, we excluded repositories which did not implement code at all, like lists of resources or books. Next, all repositories marked as *Archived* were discarded because we aim to analyze ECC libraries that are widely in use, which is less likely for archived libraries, and

also, archived libraries are not maintained, so if a vulnerability is found it will not be patched.

If a library implements code in a preferred language and has not been abandoned, the library should be examined for ECC code as a final verdict before manual analysis. An effective method was to download the library and use a regular expression to search for ECC-related key words. Visual Studio Code was used as the preferred code editor for this project and its folder-wide search feature was used to scan all files in the repository with the regular expression. The regular expression that was used was:

```
ECC | Elliptic Curve | Curve25519 | Ed25519 | X25519 | ECDSA | ECDH | ECIES | SECP256k1 |
P-256 | secp256r1 | P-384 | secp384r1 | P-521 | secp521r1 | NIST | Brainpool | Koblitz |
Montgomery | OpenSSL | libsodium | BoringSSL | mbedTLS | libecc | libsecp256k1 | NaCl |
keypair | scalar multiplication | point addition | point doubling | modular arithmetic |
field arithmetic | signature | key exchange | public key | private key | curve | invert |
hash-to-curve | RFC 7748 | RFC 8032 | RFC 4492 | TLS 1.3 | PKCS #11
```

In regular expressions the vertical bar symbol | acts as a logical OR operator, meaning a search with the above regular expression would search for all key words present at the same time. We examined the results manually to be certain that ECC code was present.

Table 4.2 shows the result of the search for *Cryptography* on GitHub. We evaluated the first 70 results (sorted by number of stars). Out of these 25 libraries were written in an unfamiliar language, 8 were libraries without code, 3 were archived (abandoned) and 12 libraries fulfilled all criteria, but did not implement ECC code. In summary, 22 out of 70 libraries fulfilled all criteria and contained ECC code. To make the results in this library review as reproducible as possible, the exact version of each library (mostly identified through Git commit hashes) is recorded in Appendix B. These libraries will be analyzed further in the following sections and chapters.

4.2.4 Dependency Resolution

During the scalar multiplication identification step described in Section 4.3.2, it became apparent that not all libraries implement scalar multiplication themselves, but rather outsource the functions by calling other ECC libraries. These libraries were retroactively added to the analysis pool, and are described here as part of library selection for clarity.

An example of such a library is the well known `Bitcoin` cryptocurrency library, which uses the secp256k1 curve. It calls the `libsecp256k1` library for ECC operations such as scalar multiplication. To properly evaluate the security of a library such as `Bitcoin`, the libraries it calls must also be evaluated. Any library is only as safe as its dependencies.

Name	Criteria	Lang.	Stars
bitcoin	Uses ECC	C++	89.2k
The algorithms	No ECC	JavaScript	32.4k
anoma	Unfamiliar lang.	Elixir	33.8k
openssl	Uses ECC	C	30.2k
ciphey	Unfamiliar lang.	Rust	21.4k
gun	Uses ECC	JavaScript	19k
js-obfuscator	No ECC	TypeScript	16.1k
zama-ai	Not a library	—	15.4k
cryptomator	Uses ECC	Java	15.2k
opennhp	Unfamiliar lang.	Go	13.8k
libsodium	Uses ECC	C	13.7k
tink (java)	Uses ECC	Java	13.5k
xxHash	No ECC	C	11k
monero	Uses ECC	C++	10.6k
CryptoSwift	Unfamiliar lang.	Swift	10.6k
Pysyft	No ECC	Python	9.9k
PrivateBin	No ECC	PHP	8.3k
Lnd	Unfamiliar lang.	Go	8.1k
tachyohn	Uses ECC	C++	7.7k
pyca/Crypto.	Uses ECC	Python	7.6k
rustTls	Unfamiliar lang.	Rust	7.4k
awsome-crypto	Not a library	—	7k
RSACTFtool	No ECC	Python	7k
Mbed-TLS	Uses ECC	C	6.7k
upspin	Unfamiliar lang.	Go	6.4k
blacke3	Unfamiliar lang.	Assembly	6.2k
reticulum	Uses ECC	Python	5.9k
tendermint	Unfamiliar lang.	Go	5.9k
signalapp	Unfamiliar lang.	Rust	5.8k
osaurus-ai	Unfamiliar lang.	Swift	5.5k
crypto++	Uses ECC	C++	5.5k
cjdns	Uses ECC	C	5.4k
forge	Uses ECC	JavaScript	5.3k
rippled	Uses ECC	C++	5.2k
grin	Unfamiliar lang.	Rust	5.1k

Name	Criteria	Lang.	Stars
s2n-tls	Uses ECC	C	4.7k
ockam	Unfamiliar lang.	Rust	4.6k
snarkOS	Unfamiliar lang.	Rust	4.5k
shiro	No ECC	Java	4.4k
cli	Unfamiliar lang.	Go	4.2k
google end to end	Archived	JavaScript	4.1k
ring	Unfamiliar lang.	Assembly	4.1k
i2pd	Uses ECC	C++	4.1k
SEAL	No ECC	C++	4k
openssh-portable	Uses ECC	C	3.8k
cryptobook nakov	Not a library	CSS	3.8k
stegcloak	No ECC	JavaScript	3.8k
cardano-sl	Archived	Haskell	3.8k
cryptobook	Not a library	Python	3.7k
google trilian	Unfamiliar lang.	Go	3.7k
FHE (google)	Not a library	—	3.6k
golang crypto	Unfamiliar lang.	Go	3.4k
jsRSAsign	Archived	HTML	3.3k
botan	Uses ECC	C++	3.3k
helib	No ECC	C++	3.2k
pycryptodome	Uses ECC	C	3.2k
Proj. Wycheproof	Unfamiliar lang.	Go	3k
liboqs	No ECC	C	2.9k
padloc	No ECC	JavaScript	2.9k
yubikey agent	Unfamiliar lang.	Go	2.9k
wolfssl	Uses ECC	C	2.8k
awsome-bc rust	Not a library	—	2.8k
CS resources	Not a library	—	2.8k
memguard	Unfamiliar lang.	Go	2.7k
minisign	Uses ECC	C	2.7k
freenet-core	Unfamiliar lang.	Rust	2.7k
keybase-gpg	Not a library	—	2.6k
Nexus zkvm	Unfamiliar lang.	Rust	2.6k
Picocrypt	Unfamiliar lang.	Go	2.5k
vuvuzela	Unfamiliar lang.	Go	2.5k

Table 4.2: The first 70 results of the GitHub search *Cryptography* [Git26b] sorted by number of stars, 24 May 2026. The repositories were categorized by the criteria in Section 4.2.3.

We present an overview of libraries that delegate functionality to helper libraries in Table 5.1. Inversely, helper libraries that were not in the original *Cryptography* search results, were marked in italic font in both the results Table 5.2 and the delegation Table 5.1. Resolving dependencies added *6* helper libraries, bringing the total analyzed to 28.

4.3 Vulnerability Analysis

This section outlines how the 28 libraries that fulfill the criterion *Uses ECC* (or were added retroactively) were analyzed to determine their safety to the monodromy leak attack. The section highlights each step that was performed to reach the safety final verdict.

4.3.1 Use of GitHub Copilot

In practice, the analysis of libraries was a non-trivial task due to the amount of code that needed to be analyzed for the 28 libraries. Many of the libraries analyzed had sparse documentation, and/or were considered too large to be analyzed without external tools. `WolfSSL` had over 3500 files and over 600 folders, being over 900 MB in size. Many of the files had tens of thousands of lines. The file `sp_c64.c` implements among others, elliptic curve operations which must be analyzed to determine the safety of its scalar multiplication routines. It spans 49,945 lines, which illustrates the need for automated tools in the analysis of libraries.

GitHub Copilot was chosen as the preferred tool for automated analysis, as it is available directly in Visual Studio Code, which was the code editor used in this project. When a repository was opened in a Visual Studio Code window, the Artificial Intelligence (AI) model could navigate all folders and files in the repository, easing the process of navigation through vast repositories such as `WolfSSL`. Furthermore, through their GitHub Education program, students are offered access to what are referred to as *premium models* such as *Claude Haiku 4.5*, *Gemini 2.5 Pro* and *GPT-5.2 Codex* free of charge [Git]. These models are labeled *premium* due to their larger context sizes and generally stronger performance on reasoning and code analysis tasks, which makes them better suited for navigating large repositories with many scalar multiplication routines.

In this project, the model *GPT-5.2 Codex* by OpenAI, was used for analyzing libraries. The model was chosen because it was the newest model by OpenAI, and the one with the largest context size available with GitHub Education to date [Ope][Git].

While AI-assisted analysis significantly reduces the time needed to navigate large repositories, the outputs should be treated with skepticism. It is well known that large language models can produce convincing and plausible-sounding but incorrect outputs. This is referred to as hallucination [JLF+23]. In regard to the experimental part of this thesis, undetected hallucination could lead to incorrect judgment on the safety of the scalar multiplication routines present in a library. For example if an AI model incorrectly states that a routine is protected with constant time inversion when it is not. To mitigate this risk, all outputs produced by the model were validated manually. The strategies for verifying output when identifying scalar multiplication routines were mentioned in Section 4.3.2 and the strategies for detecting hallucinations when collecting criteria are mentioned in 4.3.3.

4.3.2 Identifying Scalar Multiplication Functions

We first identified all scalar multiplication functions present in a library, as the monodromy leak attack applies to scalar multiplication functions and not whole

libraries. As explained in Section 5.1.1, the vulnerability currently only applies to Montgomery and Weierstrass curves, however its application to other curves is possible. If the attack were to be adapted to other curves in the future successfully, it is important to document vulnerable implementations, so all curves will be analyzed. Recall that the attack itself works on a per-implementation basis. Consequently, if a library provides multiple implementations of one curve (such as small or blinded variants), each implementation will be treated as a separate entry as they may differ in their security properties.

As mentioned in the above section, this process was non-trivial due to the size and number of files in some repositories, so this part of the analysis depended more on the use of GitHub Copilot. The first general prompt to the model in the analysis of every library was:

> **Prompt:** *Help me find the scalar multiplication functions of the ECC in this library. Find the scalar multiplication functions for all curves that are supported.*

The model would then return a list of routines that were present in the library. As mentioned in the section above, verification of the outputs produced by large language models is important. In this case, verifying that the mentioned scalar multiplication functions actually were present was a trivial task since the model returns the function's location in files and lines. A simple read-through of the suggested code was enough to determine that the function was present.

A more difficult aspect to cover with the use of AI was to make sure that the model found *all* scalar multiplication functions that exist in a library. For some libraries, reading documentation and finding supported curves would give a rough idea as to which functions would be present. Other times files in the libraries would contain lists or function calls based on supported curves, which would again aid in verifying that the large language model found all curves in the library.

4.3.3 Evaluation Criteria

To evaluate if a scalar multiplication function is vulnerable, a set of criteria has been developed. In this section, we will explain how the value of each evaluation criterion was determined when analyzing code. The reasoning for why these exact criteria were chosen is detailed in Chapter 5.

The first criterion mentioned in Section 5.3 which is **the curve or general curve class**, was collected when scalar multiplication functions were identified. As men-

tioned above, verification of the curve type was performed by reading documentation, variable names and checking curve parameters.

Next, four *binary* criteria are mentioned. When we say that a criterion is binary, it means that it is either fulfilled (marked as ✓ in the results Table 5.2) or it is not fulfilled (marked as ✗ in the results Table 5.2).

The first binary criterion is the *presence of a ladder structure* explained in Section 5.1.2. To determine if a ladder is present or not, we can look for code loops where the results of doubles and additions are added to two separate variables, and for comments like `R0` and `R1` as well as checking function names and other comments. If the routine uses precomputed tables or a general double-and-add algorithm, it is not a laddered approach.

The second binary criterion is the *presence of projective coordinates* explained in Section 5.1.3. If the library is written in object-oriented code, it can be checked by looking for definitions of point classes to see how many variables are used to store points. Another way is to check if variable names containing z are used when calculating point addition and doubling.

The third binary criterion is the *presence of constant time inversion to affine coordinates* explained in Section 5.2.1. The return to affine coordinates (if projective are used) happens at the end of the scalar multiplication routine. To evaluate the criterion, we can check whether the modular inverse of the Z-coordinate is being calculated in constant time with respect to secret data or not. We can look for conditional branches based on the value of the Z coordinate to determine if it is constant time or not. We can also look for known algorithms for calculating modular inverses such as Fermat's little theorem (constant time) or the extended Euclidean algorithm (variable time).

The fourth and last binary criterion is the *presence of random masking of coordinates* explained in Section 5.2.2. For this countermeasure to work, the mask must be random. To identify the countermeasure, we can look for random number usage and/or generation and for comments mentioning masking or blinding of coordinates. Note that only *coordinate* masking was reported. Routines where *scalars* are masked were marked as ✗ in the results in Table 5.2. As explained in Section 5.2.2, there exists a difference between masking before performing scalar multiplication and after performing it (right before returning to affine coordinates). We refer to these as pre-ladder masking and post-ladder masking respectively. We will classify each masked routine into one of the two categories, and report the distinction in Section 5.4.2.

GitHub Copilot was also used in this part of the research, and its outputs were

validated manually using the same indicators described above. In general, one prompt was used to ask for the answers to all four binary criteria:

> **Prompt:** *For the _ scalar multiplication, which multiplication technique is used (ladder vs non ladder approach)? Are projective coordinates used? How is the inversion to affine coordinates performed? Is the inversion constant time? Are there any random masks or blinding factors?*

In this prompt, the underscore _ represents any routine that was collected during the identification of scalar multiplication functions above. The prompt was generally run once per routine. The AI model would respond with answers to the binary criteria, as well as the files and lines of code it used to answer them. From there, answers were validated with the methods above. Pre-ladder masking and post-ladder masking was determined manually. Follow-up prompts were sent to clear up nuances or errors in the response if needed. Manual analysis was performed if conclusive results could not be found after multiple follow-up prompts.

4.3.4 Exclusion Criteria

When identifying scalar multiplication routines that were to be analyzed, some routines were deliberately ignored. In this section we will go through the different criteria for excluding scalar multiplication routines.

- There are legitimate reasons for performing scalar multiplication without any secret data. A crucial operation in Elliptic Curve Digital Signature Algorithm (ECDSA) is to verify signatures with scalar multiplication involving only public data. In many cases a faster specialized version of scalar multiplication that runs in variable time is used. This is not a security violation, as no secret data is exposed. We can therefore ignore these routines when looking for functions susceptible to the monodromy leak attack.
- As explained in Section 5.1.1, binary curves are not vulnerable to the monodromy leak attack, and as such will be excluded from the analysis.
- Some functions that perform scalar multiplication are actually wrapper functions that use other scalar multiplication functions as subroutines. It is for example common that variable base scalar multiplication functions use faster fixed base implementations as subroutines and adjust results to match the variable base. In these cases only the subroutine was analyzed, as the monodromy leak generally applies to the routine performing the multiplication itself.
- Libraries that are written predominantly in a familiar language sometimes contain hardware-accelerated versions of scalar multiplication written in assembly. In these cases, the scalar multiplication functions were skipped, due to the language being unfamiliar.

4.3.5 Safety Determination and Result Collection

The direct results were collected in Table 5.2. Each row corresponds to a single scalar multiplication routine. The columns record the library name, the curve or general curve class, and the four binary criteria which are: the presence of a ladder structure, use of projective coordinates, constant time inversion when returning to affine coordinates, and the presence of random coordinate masking. For the binary criteria a ✓ indicates that a property is present, a ✗ indicates it is absent, and ∃ indicates that the property exists conditionally, for example depending on the specific curve passed to a general routine. Note that the column for projective coordinate usage is omitted from the table itself, as all directly analyzed routines fulfilled the criteria, making the column uninformative.

Scalar multiplications that depend on other libraries are listed in Table 5.1, with a reference to the library that does the actual computation.

To reach the final safety verdict, the curve or general curve class of the routine, as well as its four answers to the binary criteria were evaluated in accordance with Section 5.3. Vulnerable routines were discussed separately in Section 6.6.

4.4 Limitations

This section outlines the limitations of the methodology described in this chapter, covering both the library search strategy and the analysis approach.

4.4.1 Search Strategy Limitations

Firstly, a core limitation is the restricted search strategy. Only libraries written in Python, C/C++, JavaScript, and Java were included for further analysis due to language familiarity and popularity. This excluded popular libraries like `rustTLS` and `signalapp` which harms the completeness of the findings.

Another detail that limits the search strategy occurs when taking into account that many libraries are written in multiple programming languages. When searching for libraries in GitHub, it seemed that the preview of a library showed only what the predominant language was. As such, only the predominant language has been considered when filtering libraries based on language. However, the ECC code that is important to the analysis of the monodromy leak attack could be written in any minority language. This means that some libraries could have had their ECC code written in a familiar language, but were not included in the analysis due to the majority language being unfamiliar.

Furthermore, the search was limited to the first 70 libraries ranked by star count on GitHub. This excludes many libraries from GitHub as well as all libraries from other sources. The analysis also only concerns libraries available April and May of 2026. Libraries are actively maintained and their implementations may change. If a library is marked as resistant to the monodromy leak attack, this does not consider real-world systems using older updates or possible faults in newer updates.

As mentioned when introducing stars as a metric above, the star count of a library does not necessarily correlate to its adoption. Libraries that are in use on embedded devices, or libraries which are only adopted to a specific platform, may have many active uses but have little attention by the open source community on GitHub. While it is probable that a high amount of stars indicates a significant user base, a low amount of stars does not necessarily indicate a small user base.

When many libraries are excluded due to language, not being available on GitHub (in April or May of 2026), or not having a sufficient number of stars, the result of the review is less relevant. Research Question 6 sets the goal of analyzing widely used ECC libraries, however the excluded libraries could be widely deployed. This harms the understanding how applicable the monodromy leak attack is in practice.

4.4.2 Analysis Limitations

The method of analysis explained in Section 4.3 is limited to static analysis. No code was executed, no side channel information was collected, and no exploit was tested. According to Research Question 6, a goal of the review is to understand to what extent libraries are vulnerable under realistic operational scenarios. Since only static analysis was performed, a conclusion of how vulnerable libraries are, is somewhat limited. We never executed code, so we do not have a clear understanding of unexpected runtime behavior, such as edge cases or bugs, that could introduce vulnerability or bypass countermeasures. Additionally, even code that appears constant-time in the source may not remain so after compilation, as modern compiler optimizations can introduce data-dependent behavior into the resulting binary [GPS26].

Furthermore, the exclusion of hardware implementations in the analysis may cause unsafe libraries to appear safe. Libraries tend to prefer hardware specific paths at runtime if possible, however the assembly code may have different security properties than the analyzed code. Libraries that are marked as safe by this analysis may run unsafe assembly code and be susceptible to the monodromy leak attack in practice.

The decision to analyze core scalar multiplication routines, and not their wrapper functions may also have implications on the results. Although the monodromy leak attack targets scalar multiplication itself, a wrapper function could call a core routine

in an insecure way. For example, a wrapper function could use the same blinding for the same point multiple times, meaning coordinates are not properly masked, even though it may appear that way to a core function.

Finally, although AI was used to support the analysis, it did not produce the final verdicts on its own. Every classification suggested by the model was checked against the source code manually, using the indicators described in Section 4.3.3, so the conclusions rest on manual review rather than on the model's output. This manual review is still not perfect, and it is possible that a relevant detail in the code was missed, as can happen in any manual code review. Such an oversight could, for example, cause a constant-time routine to be labeled non-constant-time, or the reverse, so the results should be interpreted with this caveat in mind.

Chapter 5
Real-World Impact of the Monodromy Leak Attack

The monodromy leak attack has so far only been described in the preceding chapters as a theoretical attack. However, its practical significance depends entirely on whether real-world implementations meet the conditions it requires. In this chapter, the necessary preconditions and available countermeasures for the attack are first identified and formalized into a set of evaluation criteria, building on the work of Robert [Rob23] and Raychaudhuri [Ray25]. These criteria cover curve type, scalar multiplication technique, and use of projective coordinates as preconditions, and constant-time inversion and coordinate masking as countermeasures. The two types of masking are distinguished in Section 5.4.2. The criteria are then applied to 28 widely used open source cryptographic libraries, and the results are presented and discussed.

5.1 Vulnerability Preconditions

This section explains the necessary preconditions that are needed for the monodromy leak attack to be applicable, covering curve types and general curve classes, scalar multiplication techniques and projective coordinates.

5.1.1 Curve Types

To determine which curve types are susceptible to the monodromy leak attack, it is useful to examine how Robert [Rob23] discusses the generality of the underlying mathematical tools.

Robert explains that the "biextension and cubical torsor structure exists for all models and we can efficiently compute canonical lifts" [Rob23, p. 22]. When Robert mentions that the biextension and cubical torsor structure works for "all models", it is reasonable to infer that he implies they are applicable to all curves, as he refers to cubical arithmetic formulas adapted to Montgomery curves as the

"Montgomery model" (see Section 3.1). These structures allow for cubical arithmetic and, consequently, make sure the crucial relation $\lambda_2 = \lambda_1^{m^2}$ holds.

Based on these mathematical tools being available for all curves, Robert concludes that "It is plausible that the monodromy leak described here for the Montgomery ladder extends to more general scalar multiplication, albeit with a more complicated polynomial depending on the exact implementation of the scalar multiplication" [Rob23, p. 22]. To relate this plausibility to the sections above, an illustrative attack (Section 3.6) can theoretically be performed for all curves, provided that a model is successfully adapted to cubical arithmetic, exactly as Robert's adaptation of the Montgomery model in [Rob24]. Furthermore, the complete monodromy leak attack (Section 3.8) can be executed on any curve if the ladder of its adapted model can be successfully compared to the cubical ladder.

Binary curves are not mentioned by Robert in the discussion referenced above, however, their unique structure is enough to determine their relation to the monodromy leak attack. As noted by Pornin [Por22], non-supersingular binary curves have an even order. Recall that a large prime order subgroup is necessary for the calculation of canonical lifts (see Section 3.3). Since the canonical lift is fundamental to the function of the attack, binary curves cannot be vulnerable and are therefore excluded from real-world analysis.

Recall that the Montgomery model arithmetic nearly coincides with cubical arithmetic, differing only by a factor of $4X_{\tilde{G}}$ in the addition formulas. As such, an algebraic expression of the deviance between the ladders is relatively straightforward to derive (Section 3.7). The arithmetic of other models, however, could diverge much further from their cubical counterparts, potentially rendering this step highly complex or infeasible.

A recent thesis by Raychaudhuri [Ray25] demonstrates precisely that the attack can be generalized to other curve classes. Firstly, he reduces Short-Weierstrass and Montgomery curves to Partially-Long Weierstrass Curves (PLWCs) and evaluates them under a generalized Montgomery ladder setting. He successfully adapts the curve structure to cubical arithmetic, analogous to achieving the illustrative attack described in Section 3.6. Going further, Raychaudhuri derives an algebraic expression to calculate the deviance between the generalized Montgomery ladder and cubical ladder, analogous to the method shown in Section 3.7. By combining the adapted model and the resulting deviance expression, a complete monodromy leak attack can be achieved. Considering Raychaudhuri's findings [Ray25] alongside the insights from [Rob23], it can be concluded that while the attack is definitively not exclusive to Montgomery configurations, its application to all prime order elliptic curves is unproven but likely.

5.1.2 Scalar Multiplication Techniques

For the monodromy leak attack to work, it is crucial for the relation between λ_1 and λ_2 shown in Section 3.5 to exist. As explained by Robert [Rob24, p. 77], this relation stems from comparing the Montgomery ladder to the cubical ladder. In the aforementioned section, Robert theorizes that the attack can work for other curves as long as corresponding models are developed (such as the framework by Raychaudhuri [Ray25] for PLWCs). It is expected that subsequent models will continue to rely on a comparison between their respective ladder implementations and the cubical ladder, following the methodologies in [Rob24] and [Ray25]. While this does not definitively prove that the attack is impossible without a ladder, a non-ladder attack or framework has yet to be proposed in the literature. We will therefore determine a ladder structure in the scalar multiplication routine to be a precondition for the monodromy leak attack.

5.1.3 Projective Coordinates

The attack in its current form works only for implementations that use projective coordinates. Since both projective coordinates and cubical points use coordinates (X, Y, Z), they can be compared in equations like $[m]G = \lambda_2 \cdot \widehat{[m]G}$, which are essential for the attack to work. Furthermore, there is no clear way to compute canonical lifts of points in the affine space, making them impractical for this attack. Naturally, we determine that the use of projective coordinates is a precondition for the monodromy leak attack.

5.2 Countermeasures

This section explains the available countermeasures to the monodromy leak attack, covering constant-time inversion when returning to the affine plane and random masking of coordinates before the ladder step.

5.2.1 Constant-Time Return to Affine Coordinates

A number of countermeasures have been proposed by Robert in [Rob23]. Firstly, an obvious countermeasure would be to not give an attacker access to the projective coordinates (X, Z) which are the result of the scalar multiplication. Generally for ECC applications, we want our result to be in the affine plane, so we will compute $x = X/Z$. This division step removes the projective information, as one affine point x refers to a whole equivalence class of projective points of size $p - 1$ over the finite field $\mathbb{F}_p$ (see Section 2.5.2). Therefore there is no way to retrieve (X, Z) from x.

A crucial detail, however, is that this division step must be implemented securely as to not leak any information about (X, Z) in the process. Recall that division in a

field is really an inversion, $x = X \cdot Z^{-1} \mod p$. The extended Euclidean algorithm can be used to find $Z^{-1} \mod p$, however, it is well known that it is vulnerable to Simple Power Analysis (SPA) [CCS17]. Another common way of computing $Z^{-1} \mod p$ is with Fermat's little theorem, which, albeit slower than the extended Euclidean algorithm, is considered secure against SPA. One could also use other algorithms that combine SPA-resistance with speed, such as [Bos14].

It is important to note that constant-time inversion does not eliminate the underlying vulnerability. It only prevents the projective coordinates from leaking during the inversion step. If an attacker can obtain the coordinates through other means, constant-time inversion provides no protection. The projective coordinates are computed and held in memory throughout the ladder regardless of how the final inversion is performed, and the value contains information about m in either case. Random masking discussed in Section 5.2.2 addresses this more fundamental concern.

5.2.2 Random Masking of Projective Coordinates

Randomizing the projective representation of a point removes the structured factor that the monodromy leak attack relies on. Two placements of this randomization have been proposed: applying it to the base point *before* the ladder (pre-ladder masking), and applying it to the result *after* the ladder, before the return to affine coordinates (post-ladder masking). As shown below, the two are not equivalent in the protection they provide.

We will first evaluate pre-ladder masking. For our monodromy leak attack to work, we are dependent on knowing the starting point of the Montgomery ladder, since we calculate the starting point's canonical lift and find λ_1. Normally the starting point of the Montgomery ladder is the base point of the curve G, which is what we have assumed up until this point. However, Robert mentions in [Rob23, p. 22] that "A supplementary countermeasure is also to mask the projective coordinates of P by a random scalar at the beginning". Specifically, pre-ladder masking means that we multiply G with some random nonce, $n \cdot G = (n \cdot X_G, n \cdot Z_G)$ before the Montgomery ladder.

Observe that the result of the scalar multiplication would still be the same, as the masking n does not change the ratio of X and Z. The final step of scalar multiplication is to return to affine space with $x = X_{[m]G}/Z_{[m]G}$. When we have pre-ladder masked the generator with n, the ladder will carry and modify the mask like how contributions are modified in Section 3.7. Recall that the add and double operations modify any extra constant λ in the same way for both the X and Z coordinates. As a consequence, the mask will be modified into some constant n', and the result will be $(n' \cdot X_{[m]G}, n' \cdot Z_{[m]G})$, meaning the final affine result will be the same: $x = n'X_{[m]G}/n'Z_{[m]G} = X_{[m]G}/Z_{[m]G}$.

Pre-ladder masking completely thwarts the monodromy leak attack, as we depend on knowing the Montgomery ladder starting point. It is also easy to implement, as only a random value mask is needed, and the final result remains the same. Crucially, unlike constant-time inversion, this countermeasure addresses the vulnerability at its root. No matter when or how an attacker gains access to the coordinates, the masked starting point makes the canonical lift of G useless.

This cannot be said about post-ladder masking, which is to re-randomize the result of the scalar multiplication immediately before returning to affine coordinates. This countermeasure was proposed by [NSS04](see Section 6.4). Here the final point (X, Z) is replaced by (rX, rZ) for a freshly chosen non-zero r (or (r^2X, r^3Y, rZ) in the Jacobian case), so an adversary who obtains leaked coordinates observes a uniformly random representative of its equivalence class. In step 5 of the monodromy leak attack described in Section 3.8, recovering m relies on the relation $\lambda_2 = \lambda_1^{m^2}$, where $\lambda_2 = Z_{[m]G}/Z_{\widehat{[m]G}}$ is computed from the leaked coordinates. Post-ladder masking scales the leaked coordinate with the random r, and the attack computes $\lambda_2' = Z_{r\cdot[m]G}/Z_{\widehat{[m]G}} = r \cdot \lambda_2$. The attack cannot continue as m cannot be recovered from $r \cdot \lambda_2 \neq \lambda_1^{m^2}$ when r is unknown.

The affine result is unchanged, since r cancels in X/Z. Unlike masking before the ladder, however, it does not protect the final projective coordinates unconditionally. An adversary who manages to leak the coordinates before masking but after the ladder, for example through memory disclosure or fault injection (see Section 6.5) can recover the useful coordinates unmasked and the attack proceeds unchanged. The protection is therefore conditional on the leak occurring at the right time, similar to how constant-time inversion only protects against leakage through the inversion step (Section 5.2.1). Figure 5.1 summarizes where each countermeasure acts in the scalar-multiplication pipeline and which leakage points it neutralizes.

5.3 Criteria

Based on the preconditions and countermeasures identified in the preceding sections, we can derive a set of evaluation criteria to determine whether a given scalar multiplication routine is vulnerable to the monodromy leak attack. The criteria consist of one descriptive variable and four binary criteria:

- **The curve or general curve class** (see Section 5.1.1)
- **Ladder multiplication technique** (see Section 5.1.2)
- **Projective coordinate usage** (see Section 5.1.3)
- **Presence of constant-time inversion** (see Section 5.2.1)
- **Presence of random masking of coordinates** (see Section 5.2.2)

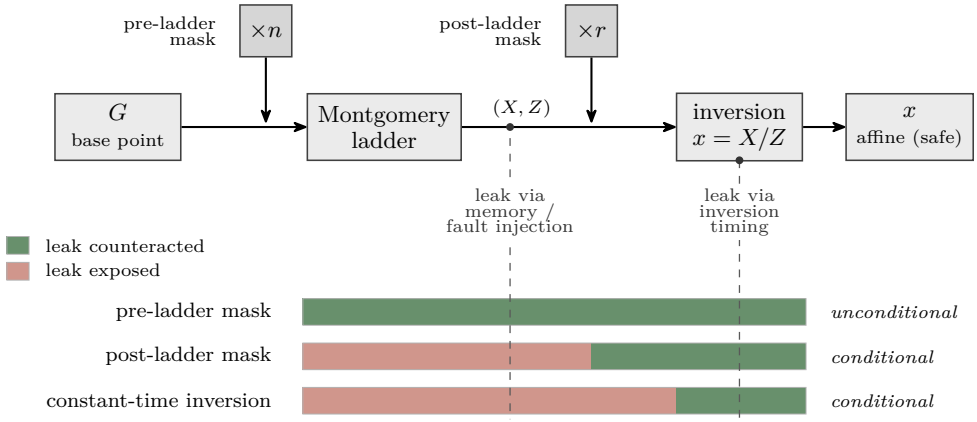


Figure 5.1: Effect of countermeasure placement in the scalar-multiplication pipeline. Pre-ladder masking randomizes the base point before the ladder, so the canonical lift of G is unknown and a leaked projective coordinates (X, Z) are useless regardless of where they are obtained; the protection is therefore unconditional. Post-ladder masking (Section 5.2.2) and a constant-time return to affine coordinates (Section 5.2.1) act only on the later stages, so the unmasked (X, Z) produced by the ladder remains exposed to a leak through memory disclosure or fault injection. Their protection is therefore conditional on the point at which the leak occurs.

To determine if a scalar multiplication routine is vulnerable to the monodromy leak attack, the preconditions of the attack must be met. During the evaluation of preconditions, curve class will be ignored, as if developed further, the attack may eventually work for any curve (see Section 5.1.1). This leaves a ladder structure and the usage of projective coordinates as necessary preconditions. Next, a routine is deemed resistant to the attack if it either uses a constant-time inversion algorithm when returning results to affine coordinates, or random masking of coordinates before computing the scalar product. It should be noted that this assessment assumes the attacker can only obtain the projective coordinates through the inversion step itself. Under a broader attacker model that includes fault injection or unzeroed memory, only random pre-ladder masking of coordinates provides unconditional protection, as discussed in Section 5.2.2. A routine is considered vulnerable if none of the above countermeasures are implemented and it fulfills the preconditions.

It should also be noted that even if a scalar multiplication routine is considered resistant to the monodromy leak attack, it may still be insecure in other respects. For example, if a routine implements a non-constant-time double-and-add algorithm it would be vulnerable to SPA, but because it does not meet the precondition of having a ladder structure, it will not be vulnerable to the monodromy leak attack.

5.4 Results

This section presents the results of applying the evaluation criteria from Section 5.3 to the scalar multiplication routines identified during the static analysis. The outcomes are collected in two tables. Table 5.2 lists every directly analyzed routine together with its curve or general curve class and its values for the binary criteria, while Table 5.1 lists the libraries that do not perform scalar multiplication themselves but instead delegate it to one of the directly analyzed routines. The presence of criteria in the results is then interpreted in the following subsections. The criteria determine which routines, if any, are susceptible to the monodromy leak attack.

Library	Curve / Function	Delegates to
Bitcoin	secp256k1	*libsecp256k1*
Google Tink	Gen. Weierstrass	Java func. (*OpenJDK*)
pyca/cryptography	Various	OpenSSL
cjdns	x25519	libsodium
Portable openssh	P-256, P-384, P-521	OpenSSL
minisign	ed25519	libsodium
reticulum	x25519 (external)	pyca/cryptography
reticulum	ed25519 (external)	pyca/cryptography
i2pd	x25519	OpenSSL
i2pd	P-256, P-384, P-521	OpenSSL
i2pd	GOST R 34.10	OpenSSL
gun	P-256	*WebCrypto*
Webcrypto	Gen. Weierstrass	*Node.js*
NodeJS	Various	OpenSSL
Cryptomator	—	*Nimbus-JOSE*
Nimbus-JOSE-JWT	P-256, P-384, P-521	Java func. (*OpenJDK*)
Nimbus-JOSE-JWT	x25519	Google Tink (Java)
s2n-tls	secp256r1	OpenSSL
s2n-tls	secp384r1	OpenSSL
s2n-tls	secp521r1	OpenSSL
s2n-tls	x25519	OpenSSL
rippled (XRP)	secp256k1	*libsecp256k1*
rippled (XRP)	ed25519	*ed25519-donna*

Table 5.1: Libraries and scalar multiplication routines that delegate scalar multiplication to another library. These are excluded from direct analysis as their properties are determined by the target library.

Library	**Curve / Function**	**Ladder**	**CT Inv.**	**Mask**
libsecp256k1	secp256k1 (General)	✗	✓	✗
libsecp256k1	secp256k1 (CT)	✗	✓	✗
libsecp256k1	secp256k1 (CT x-only)	✗	✓	✗
OpenSSL	secp224r1	✗	✓	✗
OpenSSL	secp256r1	✗	✓	✗
OpenSSL	secp256r1 (NIST z256)	✗	✓	✗
OpenSSL	secp384r1	✗	✓	✗
OpenSSL	secp521r1	✗	✓	✗
OpenSSL	SM2 P-256	✗	✗	✗
OpenSSL	Gen. Lad. Weierstrass	✓	✓	✓
OpenSSL	Gen. Wind. Weierstrass	✗	∃	∃
OpenSSL	x25519 variants	✓	✓	✗
OpenSSL	x448 Laddered	✓	✓	✗
OpenSSL	x448 Windowed	✗	✓	✗
OpenSSL	ed25519	✗	✓	✗
OpenSSL	ed448	✗	✓	✗
OpenJDK	secp256r1	✗	✓	✗
OpenJDK	Gen. Weierstrass	✗	✓	✗
libsodium	x25519 (ref10)	✓	✓	✗
libsodium	ed25519 (fixed base)	✗	✓	✗
libsodium	ed25519 (general)	✗	✓	✗
Google Tink	x25519	✓	✓	✗
Google Tink	ed25519	✗	✓	✗
Monero	ed25519 (fixed base)	✗	✓	✗
Monero	ed25519 (general)	✗	✓	✗
tachyon	Gen. Weierstrass	✗	✓	✗
tachyon	Weierstrass MSM	✗	✓	✗
tachyon	Variable base MSM	✗	✓	✗
Crypto++	Gen. Weierstrass	✗	✗	✗
Crypto++	x25519	✓	✓	✗
Crypto++	ed25519	✗	✓	✗
cjdns	ed25519	✗	✓	✗
forge-tls	ed25519	✓	✓	✗
ed25519-donna	ed25519	✗	✓	✗
Portable openssh	x25519	✓	✓	✗
Portable openssh	ed25519	✗	✓	✗
Botan	Gen. pcurve	✗	✓	✗
Botan	Gen. legacy	✗	✓	✗
Botan	x25519	✓	✓	✗
Botan	x448	✓	✓	✗
Botan	ed25519	✗	✓	✗
Botan	ed448	✗	✓	✗

Library	Curve / Function	Ladder	CT Inv.	Masking
Pycryptodome	Gen. Weierstrass	✗	✓	∃
Pycryptodome	x25519	✓	✓	✗
Pycryptodome	x448	✓	✓	✗
Pycryptodome	ed25519	✓	✓	✗
Pycryptodome	ed448	✓	✓	✗
WolfSSL	Gen. Weierstrass (ct)	✓	✓	∃
WolfSSL	SP P-256 (small)	✓	✓	✗
WolfSSL	SP P-256 (stripe)	✗	✓	✗
WolfSSL	SP P-384 (small)	✓	✓	✗
WolfSSL	SP P-384 (stripe)	✗	✓	✗
WolfSSL	SP P-521 (small)	✓	✓	✗
WolfSSL	SP P-521 (stripe)	✗	✓	✗
WolfSSL	x25519 (Main)	✓	✓	✗
WolfSSL	x25519 (Small)	✓	✓	✗
WolfSSL	x25519 (Blinded)	✓	✓	✓
WolfSSL	x448 (Small)	✓	✓	✗
WolfSSL	x448 (64 bit)	✓	✓	✗
WolfSSL	x448 (32 bit)	✓	✓	✗
WolfSSL	ed25519	✗	✓	✗
WolfSSL	ed448 (small)	✗	✓	✗
WolfSSL	ed448 (Main)	✗	✓	✗
Mbed-TLS	x25519/x448	✓	✓	✓
Mbed-TLS	Gen. Weierstrass	✗	✓	✗
Mbed-TLS	Optimized secp256r1	✓	✓	✗
reticulum	x25519 (internal)	✓	✗	✗
reticulum	ed25519 (internal)	✗	✗	✗
i2pd	ed25519	✗	✓	✗

Table 5.2: Results from direct analysis of scalar multiplication routines in open source libraries from GitHub regarding the monodromy leak attack. For the attributes **Ladder**, **Constant Time Inversion**, and **Random Masking of Coordinates**: ✓ means Yes, ✗ means No, ∃ means the property is conditional, depending on curve or build flags. Only coordinate masking is recorded in the *Masking* column, scalar masking is not. The distinction between pre-ladder masking and post-ladder masking is discussed for relevant routines below. Libraries in *italic* are helper libraries delegated by libraries from the original search (see Table 5.1). The projective coordinate criterion is omitted from the table, as all functions analyzed fulfilled it, making the column uninformative.

In total, 28 libraries were analyzed, of which 6 were retroactively added as helper libraries (marked in *italic* in Table 5.2 and Table 5.1). A total of 69 scalar

multiplication routines were directly analyzed and presented in Table 5.2, while the 23 routines that delegate their scalar multiplication to a directly analyzed routine are presented in Table 5.1. Note that the column for projective coordinate usage has been omitted. This is because all 69 directly analyzed implementations use projective coordinates, making such a column uninformative. Furthermore, the table does not contain a safety assessment of each scalar multiplication routine, as this is left for the discussion in Chapter 6.

5.4.1 Observed Preconditions

As discussed in section 5.1, the monodromy leak attack is applicable when a scalar multiplication routine uses a ladder structure and projective coordinates. To date, the attack can also only be performed on Weierstrass and Montgomery curves, however, this fact will be ignored, as the attack has potential to work on any curve. All 69 directly analyzed functions use projective coordinates, which satisfies one precondition. Of these, 27 implementations use a ladder as the scalar multiplication method, and 42 do not, meaning 27 implementations met the required preconditions (ignoring curve classes). These 27 routines could be vulnerable to the monodromy leak attack if further countermeasures are not implemented.

5.4.2 Observed Countermeasures

In general, constant-time inversion was present in 64 of the 69 directly analyzed routines, absent in 4 implementations, and conditional based on curve types or build flags in 1 implementation. As for random coordinate masking, it was present in 3 implementations, conditionally present in 3, and not present in 63 implementations. An overview of the countermeasure distribution for all 69 directly analyzed routines is available in Figure 5.2.

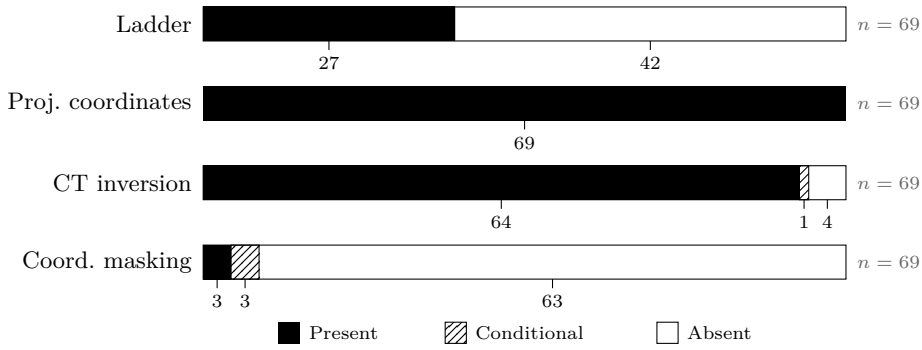


Figure 5.2: Attribute coverage across all 69 directly analyzed scalar multiplication routines.

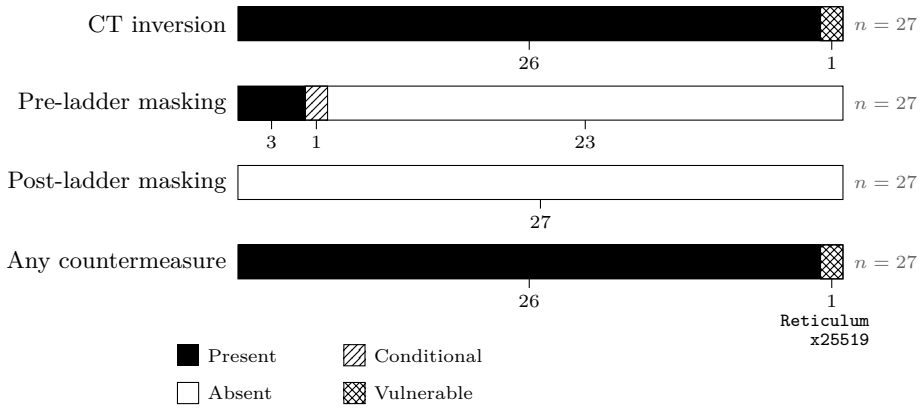


Figure 5.3: Countermeasure coverage among the 27 routines with ladder and projective coordinate preconditions.

Focusing more specifically on the *27 routines with preconditions* described in the section above, masking was only implemented in `OpenSSL - Gen. Laddered Weierstrass`, `WolfSSL - x25519 (Blinded)`, `MbedTLS - x25519/x448`, and conditionally in `WolfSSL - General Weierstrass (ct)`. All the observed routines with preconditions that deployed masking used pre-ladder masking, which is the strongest form of masking (see Section 5.2.2). Of these 27 routines, 26 routines had implemented constant-time inversion, while 1, namely `Reticulum - x25519 (internal)` had not. An overview of the countermeasure distribution for the 27 routines with preconditions is available in Figure 5.3.

To conclude, 26 out of 27 routines with preconditions to the monodromy leak attack implemented at least one countermeasure, of which 3 routines implemented both countermeasures unconditionally. The `Reticulum - x25519 (internal)` routine did not implement either countermeasure, and is therefore susceptible to the monodromy leak attack. This finding will be discussed in Section 6.6.

Chapter 6
Discussion

So far, this thesis has provided a simplified explanation of the monodromy leak in Chapter 3, and a library review of the monodromy leak in practice in Chapter 5. Drawing on both, the sections that follow answer the six research questions from Chapter 1. Lastly, we will close with a synthesis of the broader implications for ECC security.

6.1 Montgomery Ladder and Projective Coordinate Synergy

To answer Research Question 1, we first consider the individual benefits of using projective coordinates and the Montgomery ladder. For projective coordinates, they offer a significant speedup when performing scalar multiplication. By moving to projective coordinates, we can avoid using formulas with costly modular inversions in the intermediate steps, only needing one at the end (see Figure 2.7). The Montgomery ladder offers SCA resistance by ensuring that the sequence of operations is independent of the secret scalar bits (see Section 2.3). A straightforward reason for using both together is that they provide two different benefits and do not interfere with each other. A goal in cryptography is to combine speed with security, which is precisely what combining projective coordinates with the Montgomery ladder achieves.

A more interesting observation, however, is that when paired together, they provide an additional benefit. To save key space, y-independent formulas for scalar multiplication are preferred, but they are generally not available without projective coordinates. Equation 2.7 and Equation 2.6 show y-independent formulas for calculating addition and doubling on Montgomery curves. However, the formulas are not useful in themselves, as the y-independent addition is dependent on the difference $D = P - Q$, which is not readily available. The Montgomery ladder, however, has the invariant property that the difference between the two variables R0 and R1 used in computation is always G, the value of the base point (see Algorithm 2.2). This means that only if projective coordinates are used *together* with the Montgomery

ladder, can y-independent arithmetic be utilized.

Even though the synergy between the Montgomery ladder and projective coordinates has beneficial properties, it is exactly what exposes systems to the monodromy leak attack. As discussed in the section on preconditions (Section 5.1), the use of projective coordinates and the use of a ladder are precisely the two preconditions needed for the attack to work. A ladder structures the information of the secret m in such a way that it is comparable to the cubical ladder, and the projective coordinates accumulate this information in the pair (X, Z).

6.2 Exploiting Projective Coordinate Leakage

To perform the monodromy leak attack on a Montgomery curve, the curve parameters G, p, A, and the order ℓ are needed, as well as the leaked projective coordinates $(X_{[m]G}, Z_{[m]G})$ of a single multiplication (see Section 3.8). Only *one* projective coordinate leak is needed for the attack to work. This is an improvement over earlier projective coordinate leak attacks and is discussed in Section 6.3.

In Chapter 3, we see that a projective pair (X, Z) contains more information than just the final result $x = X/Z$. Specifically, we know that we can retrieve the extra information in the final projective coordinates by comparing them to the output of a cubical ladder. Crucially, the comparison between the two ladders is dependent on the secret scalar m. By computing the canonical lifts $\hat{G}$ and $\widehat{[m]G}$, we can obtain the lambda values λ_1 and λ_2. We know that $X_{[m]G}$ from the Montgomery ladder differs from $X_{[m]\tilde{G}}$ from the cubical ladder by a factor of $(4X_{\tilde{G}})^{m(2^{\text{len}(m)}-m)}$ (see Section 3.8). By combining lambda values with this deviance formula, we obtain an equation dependent on m. To solve for m, we first compute three discrete logarithms and obtain the coefficients of a modular quadratic. The final solution $m \mod (p-1)$ is found using the CRT.

Crucially, the three discrete logarithms above are computed in $\mathbb{F}_p$ rather than on the elliptic curve. As established in Section 2.4, this asymmetry makes the attack feasible: at the typical curve sizes of roughly 256 bits (Table 5.2), the finite-field DLP is solvable while the corresponding ECDLP is not.

6.3 Comparison with Prior Projective Leakage Attacks

To answer Research Question 3, we examine the two papers that preceded Robert's attack on projective coordinate leakage. The first is Naccache, Smart, and Stern [NSS04], who in 2004 defined what a projective coordinate leak attack is, and proposed the first concrete method for exploiting one. The second is Aldaya et al. [CPB20b], who sixteen years later brought the attack to practice by demonstrating

a working exploit against real cryptographic libraries. Throughout this section, we compare the two papers to Robert's paper [Rob24] in terms of attacker requirements, mathematical mechanism, and practical impact.

6.3.1 Naccache, Smart, and Stern (2004): The Original Attack

This paper was a first of its kind, defining what a projective coordinate leak attack is. At a high level, the attack assumes it is given the final projective representation of a point $P = [m]G$, computed via a standard double-and-add algorithm. Starting from P, the attack works backwards by reversing the doubling and addition formulas step by step. It checks whether the required mathematical roots for both the double and for the add operations exist in the underlying finite field. It uses existence in the field as a probabilistic oracle to rule out either the add or the double operation, and by doing so, a bit is inferred. However, the probability of correctly ruling out operations is lowered for each bit guessed. In general, the attacker can successfully recover up to a few trailing bits of the secret scalar m for every P observed.

Several structural features distinguish the [NSS04] attack from Robert's [Rob24]. First, it targets the double-and-add algorithm, not the Montgomery ladder. Second, since only a maximum of a few bits are achievable from each run, it means that many projective outputs are needed in practice. Third, the attack is probabilistic and field-dependent: the success probability of ruling out a bit value depends on $p \bmod 12$, with the most vulnerable case being $p \equiv 1 \pmod{12}$. These properties make the [NSS04] attack considerably more constrained than the monodromy leak: it is tied to the double-and-add algorithm, requires many observations rather than a single one, and succeeds only probabilistically depending on the field.

[NSS04] also proposes two countermeasures: a sign-flip of the output coordinates and a post-ladder masking of projective coordinates. Their assessment against the existing attacks is discussed in Section 6.4.

6.3.2 Aldaya et al. (2020): Bringing the Attack to Practice

Sixteen years passed between the [NSS04] paper and Aldaya et al. [CPB20b]. Despite proposing a realistic attack with concrete countermeasures, [NSS04] received little attention from the cryptography community, and the status of countermeasures in widely used libraries remained unknown [CPB20b]. Aldaya et al. set out to fill precisely this gap.

Their contribution is significant in three ways. First, they surveyed real cryptographic libraries to determine which were vulnerable. Second, they extended the original [NSS04] by showing that the Montgomery ladder does not offer protection against the attack as previously believed [CPB20b]. Third, they demonstrated a

working exploit against `libgcrypt`, making this the first practical instance of a projective coordinate leak attack.

To attack `libgcrypt`, they exploited the binary extended Euclidean algorithm used for affine conversion, as it produced variable branching behavior that was tracked through cache access patterns. Using this side channel, they recovered the needed projective coordinate output from the ECDSA signature generation in `libgcrypt`. From there, the attack from [NSS04] recovered a few bits per observation, and a lattice was used to recover the full long-term private key. They estimated that fewer than 2k traces are sufficient for the secp256r1 curve [CPB20b], making the attack possible on commodity hardware.

During exploitation, two additional vulnerabilities were discovered and assigned to CVE-2020-10932 (Mbed-TLS) and CVE-2020-11735 (wolfSSL) [CPB20b]. The filing of CVEs was significant beyond the immediate fix, as it alerted library maintainers and established the vulnerability as a concrete security risk.

The response to [CPB20b] is visible in our results. By 2026, 64 of the 69 directly analyzed routines implement constant-time inversion (see Table 5.2). In 2020, Aldaya et al. found this countermeasure to be mostly absent [CPB20b]. Several libraries were analyzed in *both* [CPB20b, Table 1] and in this thesis (Table 5.2). Specifically `wolfSSL`, `Mbed-TLS` and `Crypto++` used variable-time inversions in 2020, but switched to constant-time inversion before the library review in this thesis.

It should be noted that while constant-time inversion was considered an effective countermeasure in [CPB20b], this does not mean that it is sufficient in all threat models. This distinction is discussed in Section 6.4.

6.3.3 Robert (2024): A Breakthrough in Attacker Capability

Robert's attack [Rob24] represents a qualitative leap in attacker capability over the prior work discussed above. Three structural features distinguish the monodromy leak from the prior attacks. First, it requires only a single projective coordinate observation, compared to the thousands of traces needed by [NSS04] and the estimated 2k traces needed by [CPB20b]. This is a qualitative reduction in attacker requirements, not merely an efficiency improvement. Second, the mathematical mechanism is fundamentally different. Where [NSS04] and [CPB20b] use root-existence testing and backtracking, Robert uses canonical lifts and cubical arithmetic to derive an equation for m in $\mathbb{F}_p$, which is then solved via discrete logarithms. Third, the attack targets the Montgomery ladder specifically. It is worth noting that the Montgomery ladder was adopted precisely for its SCA resistance, yet it is this same design choice that allows the monodromy leak attack to take effect.

The attack is currently proven for Montgomery curves and Weierstrass curves via the framework developed by Raychaudhuri [Ray25] for PLWCs. Based on Robert's work [Rob24], we consider extension to all prime-order curves possible (see Section 5.1.1). If this is realized, the attack becomes relevant to virtually all ECC implementations that use a ladder together with projective coordinates.

6.4 The Monodromy Leak and Previous Countermeasures

The attacks discussed in Section 6.3 assume that the final projective coordinates of the scalar multiplication result are available to an adversary. The countermeasures that have been proposed in the literature take two fundamentally different approaches to this problem: closing the side channel through which the projective coordinates reach an adversary (see Section 6.5 for leakage vectors), or by masking projective coordinates so they carry no useful information in the first place.

Four countermeasures have been proposed across the literature. Firstly, we will explain the sign-flip countermeasure proposed by [NSS04]. It randomly replaces the output (X, Y, Z) with $(X, \epsilon Y, \epsilon Z)$ for $\epsilon = \pm 1$. This disrupts the root-existence tests the [NSS04] attack relies on, however, [NSS04] note that it only thwarts their specific attack and does not admit a formal security proof. Maimuţ et al. [MMNT13] later showed that it only makes the attack harder rather than preventing it, as the attacker can treat ϵ as an additional secret bit and backtrack for both values. The sign-flip provides no protection against Robert's attack and is not evaluated in Chapter 5, so it will not be discussed further.

The second countermeasure we will look at is pre-ladder random coordinate masking, proposed by Robert in [Rob23] and discussed in Section 5.2.2. As the name suggests, the projective coordinates are masked before the ladder step. It works by multiplying the base point G by a secret random factor n before the scalar-multiplication loop. Robert determines that pre-ladder masking is an effective countermeasure to the monodromy leak attack, and in Section 5.2.2 we see that it defends against both the narrow and the broad threat models (see Section 6.5 for threat models). However, pre-ladder coordinate masking does not protect the earlier attacks of [NSS04] and [CPB20b] at all. Aldaya et al. demonstrates this explicitly in [CPB20b, Section 4.1]. They explain that the root existence test can still be performed because the attack does not require G, and a randomized G does not influence the order of operations.

The third countermeasure is post-ladder masking which was proposed by [NSS04] as the stronger of their two countermeasures. It works by randomizing the projective coordinates after the ladder and before the inversion step. In [NSS04] the countermeasure is explained through Jacobian coordinates, where (X, Y, Z) is replaced by

(r^2X, r^3Y, rZ) for some random r immediately before the return to affine coordinates. Against the attacks of [NSS04] and [CPB20b], the output from post-ladder masking is uniformly distributed and root-existence tests cannot proceed. It also prevents the monodromy leak attack, as discussed in Section 5.2.2.

Post-ladder masking therefore defeats all three attacks under the narrow attack model (see Section 6.5). However, it does not protect the coordinates that exist before the masking step. This means that intermediate ladder values and the unmasked projective result exists in memory, and could be leaked. This thesis has not covered any attacks which work on intermediate ladder values, and it is unclear whether the monodromy leak can use them. Either way, post-ladder masking does not cover the attack scenario where a leak of unmasked projective coordinates occurs after the ladder but before the masking step. This means its protection is conditional on the leak occurring at the output, placing it in the same category as constant-time inversion rather than alongside pre-ladder masking. For a visual representation of this distinction see Figure 5.1.

The fourth countermeasure is constant-time inversion, also evaluated in Chapter 5 and discussed in Section 5.2.1. It was not proposed by [NSS04], whose attack model assumed projective coordinates were obtained directly from memory rather than through the timing of the inversion step. Its role as a countermeasure only became clear after [CPB20b] identified the variable-time binary extended Euclidean algorithm as the practical leakage vector. In the [CPB20b] attack scenario, constant-time inversion effectively closes the side channel through which the projective coordinates are recovered. However, as discussed in Section 5.2.1, it does not eliminate the underlying vulnerability. All three attacks only require that the projective coordinates be available, regardless of how they were obtained. If they leak through any channel other than the timing of the inversion itself, constant-time inversion provides no protection.

The practical implication of this comparison is visible in the results of Chapter 5. Of the 27 routines that meet the preconditions for the monodromy leak attack, 26 implement constant-time inversion and only 3 implement pre-ladder masking unconditionally (see Figure 5.3). Constant-time inversion is therefore the dominant defense in the current library landscape. This may be because it was the main countermeasure discussed by [CPB20b]. Under a broader attacker model that includes leakage in other ways, 23 of the 27 routines have no remaining countermeasure against Robert's attack.

6.5 Projective Coordinate Leaks in Practice

To answer Research Question 5, we consider how an adversary could obtain the final projective coordinates of a scalar multiplication in practice. The attacks discussed in Section 6.3 assume these coordinates are available, but do not require a specific leakage mechanism. The way in which projective coordinates are obtained influences which types of implementations are practically at risk.

The most thoroughly documented leakage vector is variable-time inversion, exploited by Aldaya et al. [CPB20b]. The attack works by tracking data-dependent cache access patterns on shared hardware. This attack scenario did not require physical access to the device, but controlling the operating system (inside an Intel SGX enclave) was still needed. Brumley et al. [BT11] demonstrated that timing attacks against ECC scalar multiplication are possible over a network connection without physical access to the target. Together, the two papers suggest that it may be possible for purely software-based, remote attackers to obtain secret values. This makes variable-time inversion the largest threat for server-side deployments.

A distinct leakage vector is described by Naccache, Smart, and Stern [NSS04]. They observe that leakage may be *"caused by a computationally-constrained secure token that sub-contracts the affine conversion of P to the external world"* [NSS04]. Smart cards and secure tokens may have sufficient resources to perform scalar multiplication internally, yet lack the capacity to return the result to affine coordinates. In such cases, the device transmits the raw projective coordinates to an external host. Any adversary with access to the communication channel between the device and the host obtains the projective coordinates directly, without needing to attack the device at all.

A further concern described by [NSS04] is that implementations may leave projective coordinates in memory after execution. An adversary with sufficient access to memory, for example through a privileged co-process or a cold-boot attack, could retrieve the coordinates directly, without ever targeting the inversion step. Modern secure coding practices address this through explicit memory sanitization after use, but the practice is not universal.

Hardware fault injection represents a fourth leakage vector that is somewhat distinct from the above, but nevertheless relevant to the broader attack surface. Maimuţ et al. [MMNT13] showed that faults induced during the projective-to-affine conversion step can cause an implementation to return (X, Z) directly rather than the intended affine result $x = X/Z$. An adversary capable of triggering such a fault obtains the projective coordinates from an implementation that would otherwise only output affine coordinates.

Countermeasure	Inversion leakage	Broader leakage
Sign-flip	No	No
Constant-time inversion	Yes	No
Post-ladder masking [NSS04]	Yes	No
Pre-ladder masking [Rob23]	Yes	Yes

Table 6.1: Effectiveness of countermeasures **against the monodromy leak**, categorized by the channel through which the coordinates are leaked. Only masking applied before the ladder protects under the broader leakage channel.

A final consideration affects the reliability of the static analysis performed in Chapter 5. Even an implementation that appears constant-time at the source code level may not remain so after compilation. Gerlach, Pietsch, and Schwarz [GPS26] demonstrate that modern compiler optimizations can introduce data-dependent behavior into the binary even when the source is written with constant-time logic. This means that a routine identified as constant time by its developers may still be exploitable via a timing channel in the compiled binary.

Based on the threat vectors above, we have decided to define two qualitatively different threat models, which this discussion refers to as the *narrow* and *broader* leakage models. Under the narrow leakage model, the projective coordinates are never directly available and the adversary can only infer them from the execution time or cache access patterns from the of the final field inversion. The narrow leakage model is inspired by the Aldaya et al. [CPB20b] attack scenario. This is threat scenario is relevant for server-side deployments and remote attacks. The coordinate value itself never leaves the device, so constant-time inversion or post-ladder masking makes leaking coordinates impossible, rendering the monodromy leak and the earlier attacks from [NSS04] and [CPB20b] useless.

Under the broader leakage model, the adversary obtains an unmasked projective coordinates directly, rather than inferring it from timing or access patterns. This covers the remaining vectors: coordinates transmitted by a sub-contracting token, unsanitized memory, or leakage through fault injection. Pre-ladder coordinate masking still covers this broader leakage model for the monodromy leak, as discussed in Section 5.2.2. The coordinates may be obtainable, but they are useless due to the starting point of the ladder being masked. However, as discussed in Section 6.4, pre-ladder coordinate masking does protect the earlier attacks of [NSS04] and [CPB20b] under any leakage model.

The two columns of Table 6.1 summarizes the vulnerability under different leakage models and for different countermeasures for the monodromy leak, and Table 6.2

Countermeasure	Inversion leakage	Broader leakage
Sign-flip	No	No
Constant-time inversion	Yes	No
Post-ladder masking [NSS04]	Yes	No
Pre-ladder masking [Rob23]	No	No

Table 6.2: Effectiveness of countermeasures **against earlier projective coordinate leakage** attacks in [NSS04] and [CPB20b], categorized by the channel through which the coordinates are leaked. No countermeasure protects under the broader leakage channel.

summarizes them for the earlier [NSS04] and [CPB20b] attacks.

Evaluating our findings for our library review in light of the two threat models, the `Reticulum x25519` routine (Section 5.4.2) is vulnerable to the monodromy leak under the narrow model, while the 23 routines that rely solely on constant-time inversion are exposed under the broader model. Which countermeasures a given system should deploy depends on how exposed it is. Suggestions for countermeasures based on threat models is taken up in the recommendations of Section 6.7.

6.6 Real-World Exposure: The Monodromy Leak in the Wild

To answer Research Question 6, we interpret the results of the library review in Chapter 5 in the context of Robert's attack. The analysis covered 69 directly analyzed scalar multiplication routines across 28 open source libraries, of which 27 routines met the preconditions for the monodromy leak attack.

The overall picture is encouraging. Of the 27 routines with preconditions, 26 implement constant-time inversion, leaving only one routine without any countermeasure. As discussed in Section 6.3.2, this marks a substantial shift from the landscape Aldaya et al. surveyed in 2020 and reflects the ecosystem's response to their work. Among the routines protected with constant time inversion, the defense is narrower than it may appear: only 3 routines of 26 implement unconditional masking. This means that a large majority of the reviewed routines with preconditions have no countermeasure under the broader leakage model. This is not a vulnerability finding in itself, but it represents a systemic gap in defense depth across the ecosystem.

The single vulnerable routine is the internal `x25519` implementation in `Reticulum`. `Reticulum` ships with a fallback implementation of all cryptographic primitives written in Python. In the default installation, `x25519` is delegated to `pyca/cryptography`, which is backed by `OpenSSL`. The vulnerability therefore only manifests when the

Python fallback is active, for example when the user installs the *rnspure* package. The `Reticulum` documentation itself acknowledges this risk, warning that the Python primitives have "not seen the same amount of scrutiny, testing and review as those from `OpenSSL`", and that users wishing to use them should "have a good understanding of the risks that this poses, and make an informed decision on whether those risks are acceptable" [Qvi26]. This self-assessment by the maintainers is consistent with the finding that the fallback path is vulnerable to the monodromy leak attack.

Listing 6.1: The listing shows the vulnerable x25519 function in `Reticulum`. The vulnerability appears at line 13.

```
1  def _raw_curve25519(base, n):
2      zero = (1, 0)
3      one = (base, 1)
4      mP, m1P = zero, one
5
6      for i in reversed(range(256)):
7          bit = bool(n & (1 << i))
8          mP, m1P = _const_time_swap(mP, m1P, bit)
9          mP, m1P = _point_double(mP), _point_add(mP, m1P, one)
10         mP, m1P = _const_time_swap(mP, m1P, bit)
11
12     x, z = mP
13     inv_z = pow(z, P - 2, P)
14     return (x * inv_z) % P
```

The internal routine (see Listing 6.1) uses Fermat's little theorem for inversion by computing `pow(z, P - 2, P)`. While this is the correct algorithmic choice, CPython's implementation of `pow` has an execution time which depends on the value of the base `z` [Pyt26]. The inversion is therefore not constant-time in practice, despite appearing to use the correct algorithm.

The scope of the vulnerability within `Reticulum` is not limited to a minor feature. In the Python fallback code, the vulnerable scalar multiplication routine is used in link establishment and message encryption. This affects the core of `Reticulum`'s networking stack. The most plausible attack scenario is exploitation of the non-constant-time inversion, analogous to the cache attack demonstrated by Aldaya et al. against `libgcrypt` [CPB20b].

This analysis adds to what has been discovered by Aldaya et al. [CPB20b]. Their study was narrower in scope, covering fewer libraries, but deeper in execution, as they performed real end-to-end attacks. The present analysis covers a broader set of libraries through static analysis, without executing code or demonstrating a working exploit. Several limitations affect the interpretation of the results, as discussed in

detail in Section 4.4. From a search strategy perspective, libraries were excluded based on their language or their number of stars, meaning widely deployed implementations like `signalapp` and `rustTLS` were not analyzed. From an analysis perspective, a static analysis approach to vulnerability detection cannot detect compiler-introduced variable timing [GPS26], and hardware-accelerated assembly and wrapper functions were excluded (see Section 4.4). The true vulnerability rate may therefore be higher than the results suggest, and the findings should be interpreted with these caveats in mind.

6.7 Synthesis: Implications for ECC Security

The preceding sections reveal a pattern that extends beyond the individual research questions: a recurring gap between the publication of a correct cryptographic result and the practical response it warrants.

[NSS04] was important in 2004, yet it did not reach maintainers, and libraries remained unpatched for sixteen years. [CPB20b] broke this pattern by naming real libraries, demonstrating a working exploit, and filing CVEs that created accountability. The countermeasure landscape documented in Chapter 5 shows a genuine improvement over the landscape [CPB20b] documented in 2020. Robert's attack [Rob24] is now in the position [NSS04] occupied in 2004. It is the most capable of the three attacks, yet it is buried inside a paper about fast pairings and biextensions, with no CVEs, no practical demonstration, and no visible ecosystem response. Even though constant-time inversion is a genuine and effective defense against the monodromy leak attack, we argue that because of its increased potency, library maintainers in higher-threat settings should not treat constant-time inversion as sufficient.

This thesis attempts a modest analog of what [CPB20b] did for [NSS04]: reformulating the attack accessibly, analysing real libraries, and naming a vulnerable implementation. It is not a complete bridge, as no exploit was demonstrated, but it is a step in the right direction. The explicit hope is that this work, together with eventual practical demonstrations by others, will ensure it does not take another sixteen years for a theoretical result to become a practical response.

6.7.1 Recommendations

For libraries deployed in the broader threat model where the attacker may have physical access, such as embedded devices, Hardware Security Modules (HSMs) or smart cards, pre-ladder coordinate masking is the appropriate countermeasure and is currently only implemented completely in 3 out of 27 surveyed implementations with preconditions. A concrete example of needed improvement is in the `Mbed-TLS` library, which brands itself as "suitable for embedded systems" [Mbe26], yet its `secp256r1`

routine has the preconditions for the monodromy leak and does not implement any form of masking (see Table 5.2). In such cases, libraries are strongly advised to implement pre-ladder masking for protection.

For general-purpose software deployments (narrow leakage model), constant-time inversion remains the most practically important and widely applicable countermeasure, and its near-universal adoption is a real security gain. Nonetheless, its limitations under non-timing leakage models should be acknowledged explicitly in security documentation rather than treated as complete protection.

The `Reticulum` Python fallback, identified in Section 6.6, is the only routine observed without any countermeasure, and should be patched. As shown there, its inversion is algorithmically correct but not constant-time in CPython. It is understandable that `Reticulum` wants to keep compatibility with platforms where `OpenSSL` is unavailable. A fix could therefore be not to remove the Python fallback but to harden it by replacing the value-dependent inversion with a constant-time implementation. Given that the `Reticulum` documentation itself warns users about the reduced scrutiny of the Python primitives, the maintainers appear aware of the broader risk, and should take the next step in securing their implementation further.

6.7.2 Future Work

Two areas of focus would most directly extend this work. First, and most impactful: a practical end-to-end demonstration of Robert's attack against a real implementation, analogous to what Aldaya et al. achieved for the [NSS04] result. This would give the attack empirical credibility and needed attention that only CVE filings and working exploits can provide. Second, a systematic extension of Robert's framework to Edwards curves and other curve models, building on Raychaudhuri's work [Ray25], would clarify how broadly the attack applies.

Chapter 7
Conclusion

Elliptic curve cryptography is secure in theory, but not always in practice: the mathematics is sound, but the implementations that deploy it can leak the secret anyway. The monodromy leak attack is one of the strongest tools to leverage such a leak. Earlier projective coordinate attacks recovered a few bits of the secret from many observations, while Robert's attack [Rob24] recovers the whole scalar from a single leaked projective coordinate. The single-coordinate attack represents a fundamental shift in threat modeling, rather than a mere incremental improvement to attacker capability.

This thesis had two aims: to make the attack understandable without the machinery it was first formulated in, and to find out which real world libraries are actually exposed. It pursued both, reformulating and testing the attack with accompanying code and surveying the current ECC ecosystem for it.

On the theoretical side, the attack was reformulated around a leaner toolkit: cubical points, the canonical lift, and an explicit ladder-deviance formula. The original presentation utilized many of the same fundamental concepts, but explained them in relation to the advanced mathematical concepts of biextensions and cubical torsors (Chapter 3). The attack exploits the fact that a projective pair (X, Z) holds more information than the affine value $x = X/Z$ it reduces to, and comparing the Montgomery ladder against a cubical ladder turns that surplus into an equation which depends on the secret scalar m. Solving it takes three discrete logarithms in $\mathbb{F}_p^*$ and the CRT. At the roughly 256-bit sizes used in deployment, the finite-field discrete logarithm is within reach while the elliptic-curve one is not, which is what makes the attack feasible.

On the practical side, our static-analysis survey found 27 scalar multiplication routines that meet the attack's preconditions. 26 of these routines perform constant-time inversion, which is a clear improvement on what Aldaya et al. [CPB20b] found in 2020. However, due to the limitations in the methodology, these numbers constitute

a lower bound rather than an assurance of security. Static analysis cannot see compiler-introduced timing variation, and some widely deployed libraries fell outside the language and popularity criteria (Section 4.4), so the true rate of exposure may be higher.

We have seen that constant-time inversion closes the timing channel only, and that the way coordinates leak depends on the attack surface of the deployment. Since the monodromy leak attack is a breakthrough in attacker capability, it is advisable that implementations with greater potential of leakage implement further countermeasures. The strongest countermeasure for the monodromy leak is pre-ladder coordinate masking, which was absent in 23 of the 27 routines. Some libraries deployed in the broader leakage model context, such as the `Mbed-TLS` had routines that did not implement masking. This indicates that pre-ladder coordinate masking is not a common countermeasure, and that it is not being deployed by the exposed libraries that need it. One routine was also found to be exposed under the narrowest model: the Python fallback for `x25519` in the `Reticulum` networking stack performs inversion in non-constant time and applies no masking.

Together, these results constitute a significant improvement in the bridge between Robert's work and the developers it affects. The attack has been made more accessible, libraries have been analyzed in its context, a vulnerable implementation has been named and recommendations to developers have been proposed. The most useful next step is to close that gap with an end-to-end demonstration that brings attention and accountability and to extend the framework to Edwards curves and other models. With the contributions that this thesis brings, the question of whether the community's response takes another sixteen years or far less is no longer a matter of mathematical obscurity or doubts about its real-world applicability.

References

[Aal25] J. K. A. Aalen, *Analyzing a new devastating side-channel attack on elliptic curve cryptography*, Specialization project (TTM4502), Norwegian University of Science and Technology (NTNU), Unpublished, 2025.

[BGS22] G. Banegas, V. Gilchrist, and B. Smith, "Efficient supersingularity testing over gf(p) and csidh key validation", *Transactions on Mathematical Cryptology*, vol. 2, no. 1, pp. 21–35, 2022. [Online]. Available: https://journals.flvc.org/mathcryptology/article/view/132125.

[Bit] Bitcoin Wiki, *Bitcoin protocol documentation*, Accessed: 2025-10-29. [Online]. Available: https://en.bitcoin.it/wiki/Protocol_documentation.

[BL24] D. J. Bernstein and T. Lange, *Safe curves for elliptic-curve cryptography*, Cryptology ePrint Archive, Paper 2024/1265, 2024. [Online]. Available: https://eprint.iacr.org/2024/1265.

[Bos14] J. W. Bos, "Constant time modular inversion", *Journal of Cryptographic Engineering*, vol. 4, pp. 275–281, 2014. [Online]. Available: https://doi.org/10.1007/s13389-014-0084-8.

[BT11] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical", in *Computer Security – ESORICS 2011*, V. Atluri and C. Diaz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 355–371.

[CAD26] CADO-NFS Development Team, *Cado-nfs: An implementation of the number field sieve algorithm*, 2026. last visited: Jun. 12, 2026. [Online]. Available: https://github.com/cado-nfs/cado-nfs.

[CCS17] A. Cabrera Aldaya, A. J. Cabrera Sarmiento, and S. Sánchez-Solano, "Spa vulnerabilities of the binary extended euclidean algorithm", *Journal of Cryptographic Engineering*, vol. 7, pp. 273–285, 2017. [Online]. Available: https://doi.org/10.1007/s13389-016-0135-4.

[CFAD06] H. Cohen, G. Frey, et al., *Handbook of elliptic and hyperelliptic curve cryptography*. Chapman & Hall/CRC, 2006.

[CMR+23] L. Chen, D. Moody, et al., "Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters", National Institute of Standards and Technology, NIST Special Publication 800-186, 2023. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-186.

[CPB20a] A. Cabrera Aldaya, C. Pereida García, and B. B. Brumley. “From A to Z: Projective coordinates leakage in the wild”. Diagram at 1:35, IACR/CHES, last visited: Jun. 6, 2026. [Online]. Available: https://www.youtube.com/watch?v=je5XAXt_4Jk.

[CPB20b] A. Cabrera Aldaya, C. Pereida García, and B. B. Brumley, “From a to z: Projective coordinates leakage in the wild”, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2020, no. 3, pp. 428–453, 2020. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8596.

[Exp26a] Explicit-Formulas Database. “Explicit formulas database: Xz coordinates for montgomery curves — dadd-1987-m-3”. Addition formula dadd-1987-m-3, Accessed: 2026-02-02. [Online]. Available: https://www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html#diffadd-dadd-1987-m-3.

[Exp26b] Explicit-Formulas Database. “Explicit formulas database: Xz coordinates for montgomery curves — dbl-1987-m-3”. Doubling formula dbl-1987-m-3, Accessed: 2026-02-02. [Online]. Available: https://www.hyperelliptic.org/EFD/g1p/auto-montgom-xz.html#doubling-dbl-1987-m-3.

[Git] GitHub, *Students - github education*, Accessed: 2026-05-26. [Online]. Available: https://github.com/education/students.

[Git25] GitHub, “Octoverse 2025: The state of open source and ai on github”, GitHub, Inc., Tech. Rep., 2025, Accessed: 2026-05-24. [Online]. Available: https://octoverse.github.com.

[Git26a] GitHub, *Cryptography topic page*, Accessed: 2026-04-14, 2026. [Online]. Available: https://github.com/topics/cryptography.

[Git26b] GitHub, *Repositories matching Cryptography, sorted by stars*, https://github.com/search?q=Cryptography&type=repositories&s=stars&o=desc, Accessed: 24 May 2026, 2026.

[GPS26] L. Gerlach, R. Pietsch, and M. Schwarz, “Do compilers break constant-time guarantees?”, in *Financial Cryptography and Data Security*, C. Garman and P. Moreno-Sanchez, Eds., Springer Nature Switzerland, 2026, pp. 327–344.

[JLF+23] Z. Ji, N. Lee, et al., “Survey of hallucination in natural language generation”, *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, Mar. 2023. [Online]. Available: http://dx.doi.org/10.1145/3571730.

[JOP14] A. Joux, A. Odlyzko, and C. Pierrot, “The past, evolving present, and future of the discrete logarithm”, in *Open Problems in Mathematics and Computational Science*, Ç. K. Koç, Ed. Cham: Springer International Publishing, 2014, pp. 5–36. [Online]. Available: https://doi.org/10.1007/978-3-319-10683-0_2.

[LHT16] A. Langley, M. Hamburg, and S. Turner, *Elliptic Curves for Security*, RFC 7748, 2016. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7748.html.

[Mbe26] Mbed TLS Project. “Mbed tls”, Trusted Firmware, last visited: Jun. 11, 2026. [Online]. Available: https://github.com/Mbed-TLS/mbedtls.

[MMNT13] D. MaimuŢ, C. Murdica, et al., “Fault attacks on projective-to-affine coordinates conversion”, in *Constructive Side-Channel Analysis and Secure Design*, E. Prouff, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 46–61.

[Mon87] P. L. Montgomery, “Speeding the pollard and elliptic curve methods of factorization”, *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987.

[MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, 1st. CRC Press, 1997.

[NSS04] D. Naccache, N. P. Smart, and J. Stern, “Projective coordinates leak”, in *Advances in Cryptology - EUROCRYPT 2004*, C. Cachin and J. L. Camenisch, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 257–267.

[Ope] OpenAI, *Introducing gpt-5.2-codex*, Accessed: 2026-05-27. [Online]. Available: https://openai.com/index/introducing-gpt-5-2-codex/.

[Por22] T. Pornin, *Efficient and complete formulas for binary curves*, Cryptology ePrint Archive, Paper 2022/1325, 2022. [Online]. Available: https://eprint.iacr.org/2022/1325.

[Pyt26] Python Software Foundation, *CPython source code: `long_pow` in `Objects/longobject.c`*, https://github.com/python/cpython/blob/c720e18c941bd0cf4d64f1ac3ef098f73826f273/Objects/longobject.c#L4967, Accessed: 2026-05-26, 2026.

[Qvi26] M. Qvist, *Reticulum network stack: Cryptographic primitives*, Accessed: 2026-05-26, 2026. [Online]. Available: https://github.com/markqvist/Reticulum#cryptographic-primitives.

[Ray25] A. Raychaudhuri, “The monodromy leak for a generalized montgomery ladder”, M.Tech. thesis, Department of Cryptology and Security Research Unit (CSRU). In collaboration with KU Leuven (COSIC), M.S. thesis, Indian Statistical Institute, Kolkata, India, 2025.

[Res18] E. Rescorla, “The transport layer security (tls) protocol version 1.3”, RFC Editor, RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446.txt.

[Rob23] D. Robert, “Improving the arithmetic of kummer lines”, Working Note, Precursor to Fast pairings via biextensions and cubical arithmetic, 2023. [Online]. Available: https://www.math.u-bordeaux.fr/~damienrobert/pro/publications/notes/2023-11-kummer_lines.pdf.

[Rob24] D. Robert, “Fast pairings via biextensions and cubical arithmetic”, Preprint, HAL hal-04848028, 2024. [Online]. Available: https://hal.science/hal-04848028v1.

[Sig25] Signal Foundation, *Signal protocol documentation*, 2025. last visited: Oct. 29, 2025. [Online]. Available: https://signal.org/docs/.

[Sta09] Standards for Efficient Cryptography Group, “Sec 1: Elliptic curve cryptography”, Certicom Research, Tech. Rep. SEC 1, version 2.0, 2009, Available at https://www.secg.org/sec1-v2.pdf.

[Uni26a] United Nations. “Goal 16: Promote just, peaceful and inclusive societies”. Accessed: 2026-06-05, United Nations Department of Global Communications. [Online]. Available: https://www.un.org/sustainabledevelopment/peace-justice/.

[Uni26b] United Nations. “Goal 9: Build resilient infrastructure, promote sustainable industrialization and foster innovation”. Accessed: 2026-06-05, United Nations Department of Global Communications. [Online]. Available: https://www.un.org/sustainabledevelopment/infrastructure-industrialization/.

[Uni26c] United Nations. “The sustainable development goals”. Accessed: 2026-06-05, United Nations Department of Global Communications. [Online]. Available: https://www.un.org/sustainabledevelopment/.

[Was08] L. C. Washington, *Elliptic Curves: Number Theory and Cryptography, Second Edition*. Chapman & Hall/CRC, 2008.

Appendix A

Monodromy Leak Attack on Curve25519

Listing A.1: The monodromy leak attack on curve25519, recovering the secret scalar in roughly 15 minutes on a consumer laptop. The script relies on CADO-NFS for the number-field-sieve discrete logarithms [CAD26], which dominates the runtime.

```
# Step 0: Acquire leak [m]G
mG_montgomery = generate_leak(G, ell)

#Step 1: Calculate canonical lift of G
G_hat = canonical_lift(G)

#Step 2: la_1 = Z_G / Z_G_hat
la_1 = G[1] / G_hat[1]

#Step 3: Calculate canonical lift of [m]G"
mG_hat = canonical_lift(mG_montgomery)

#Step 4: la_2 = Z_mG / Z_mG_hat
la_2 = mG_montgomery[1] / mG_hat[1]

#Step 5: la_2 = 4X_G^{m * ( 2^{len(m)} - m )} * la_1^(m^2)

#Step 6: Simplify Equation

#Step 7: Find discrete logs of 4X_G and lambdas
X_G = G[0]
gamma = F.multiplicative_generator()
factors = factor(p - 1)

t0 = perf_counter()
print(f"Dlogs with CADO-NFS (mod p-1 via CRT over its factors)")
d_x, d_la_1, d_la_2 = cado_logs_mod_p_minus_1(
    p,
    gamma,
    [F(4 * X_G), la_1, la_2],
    threads=CADO_THREADS,
)
d_x %= (p - 1)
d_la_1 %= (p - 1)
d_la_2 %= (p - 1)
```

```
t1 = perf_counter()
print(f"CADO-NFS dlogs found! ({t1 - t0:.6f}s)")

#Step 8: Solve the modular quadratic with CRT

def check_candidate_m(candidate_m, mG_montgomery):
    if not (1 <= candidate_m <= ell):
        return None
    candidate_mG = montgomery_ladder(G, candidate_m)
    if candidate_mG[0] * mG_montgomery[1] == mG_montgomery[0] *
        candidate_mG[1]:
        print("Recovery complete!")
        print("m:", candidate_m)
        return candidate_m

def solve_and_test_quadratic(A, B, C, factors, mG_montgomery):
    mod_solutions = []
    moduli = []

    for p_i, e_i in factors:
        mod = p_i**e_i

        R = Integers(mod)['X']
        X = R.gen()

        f = (A % mod) * X**2 + (B % mod) * X + (C % mod)

        if f.degree() == -1:
            roots = [Integer(a) for a in range(mod)]
        else:
            roots = [Integer(r) for r in f.roots(multiplicities=False)]
            if len(roots) == 0:
                return None  # wrong bit length

        mod_solutions.append(roots)
        moduli.append(mod)

    for comb in itertools.product(*mod_solutions):
        candidate_m = crt(list(comb), moduli)
        result = check_candidate_m(candidate_m, mG_montgomery)
        if result:
            return result

    return None


coeff_A = d_la_1 - d_x
coeff_C = -d_la_2

for m_bit_length in range(ell.bit_length(), 1, -1):
    coeff_B = d_x * 2**m_bit_length
    found = solve_and_test_quadratic(coeff_A, coeff_B, coeff_C, factors
        , mG_montgomery)
    if found:
        break
```

Appendix B
Versions of Analyzed Libraries

Table B.1: Exact Git commits (or other source) of the analyzed libraries. Each hash links to the repository state at the time of analysis. Like in Table 5.2, the 6 libraries introduced by dependencies are marked in *italic*.

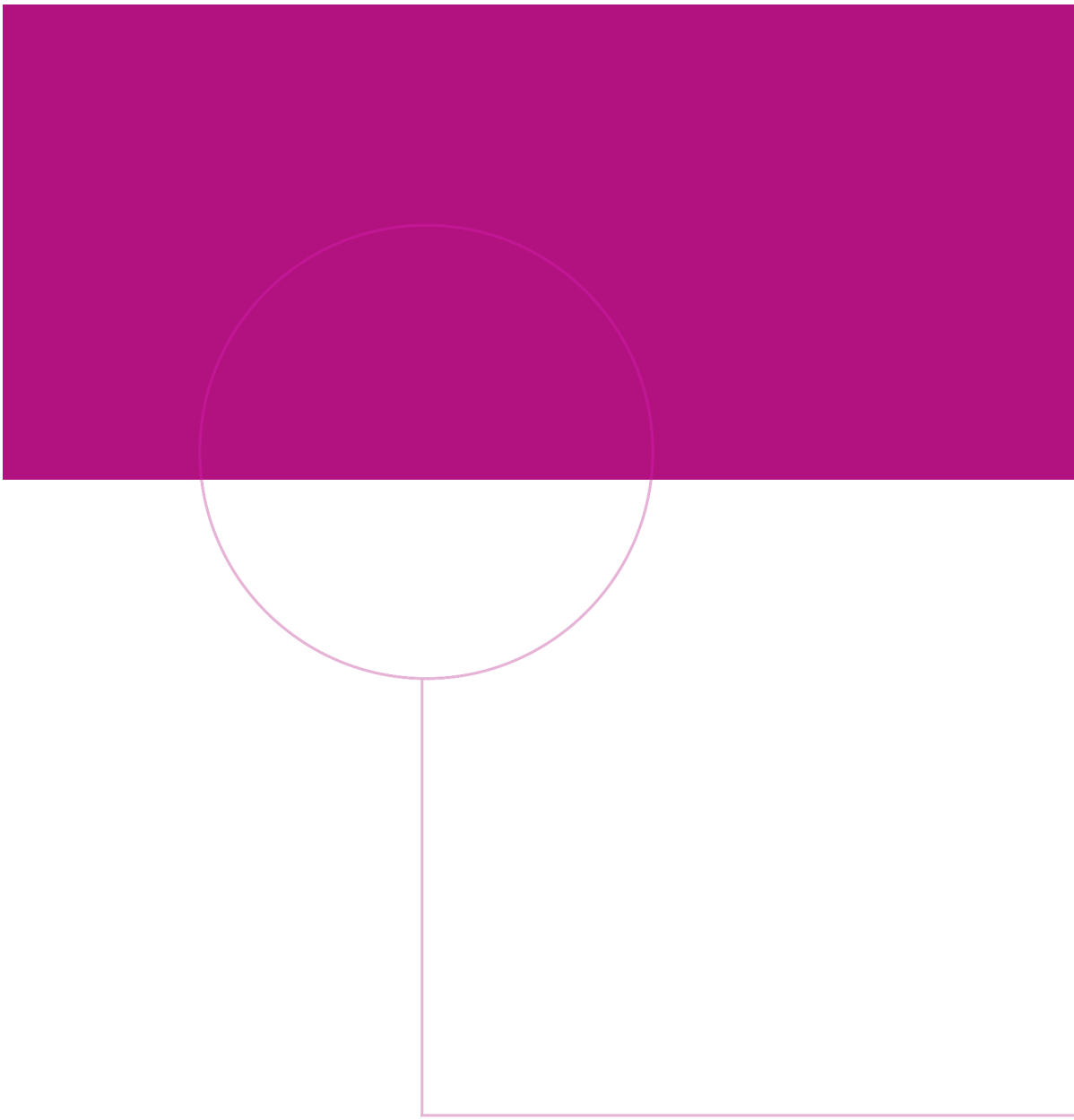
Library	Commit
Bitcoin	via *libsecp256k1*[a]
botan	`690916ff21850bbcc1f431d1c11f24475d1a9efe`
cjdns	`f5902acab6dd2f1a68bf6e4c89e352ed71f93698`
cryptomator	`ea86124c7d15dcf474a3462aab88ef598d1237e3`
Crypto++	`b5242667a24e3db8e4600e77b2e502ef204e5280`
ed25519-donna	`8757bd4cd209cb032853ece0ce413f122eef212c`
forge-tls	`7a43db987bd0ecdc5b41f6d73f58ba6ca5bf9ae1`
google-tink-java	`455abae35f21e6f7fcd661e315ef6e2e331e7606`
gun	`ed27b48569ef1fa65ec96ed97e32ea590d178dfb`
i2pd	`f930390e03a64d81d9e3436d55af0b4bbd1579d8`
libsecp256k1	`ea174fe045e1832548cd3b7090958afe9573ad2b`
libsodium	`df8802b013f9fa67e83eb90f184713477da68f32`
mbedTLS	`701132e5b333c5fc6dc4daf56e998a4564f47d78`
minisign	`41006b4f33e3277cafc17b28ddf83d4462fbda30`
monero	`9d7ba46a3606e4b8154695d1239b0ae8c81dd88a`
nimbus-jose-jwt	`d919f7bf5eef668c056cc01e6724799715213246`[c]
nodeJS	`bf7e79c264eef9bd0c743588aa6a158a26d90103`
openjdk	`102302ba87481d4ed90793ee43992c67699c11f1`
openssh-portable	`ac4a41265a3beccba7dc6f45c657298311280f01`
openssl	`35852da1d9e24cb74034b2f418cef3a58203b127`
pyca/cryptography	`99de8d186717dffb964cb48c59beb9e261b6f119`
pycryptodome	`d12264940c40417915fb85acb8c787ad9da58969`
Reticulum	`fa7699a37d141d82a33f52c447353ad9e43f98e8`
rippled	`8490206228a6413d41449a01d092ef3ab3b5e1d5`
s2n-tls	`4c11924c97ce61f0ed692fb15bb28acd42801721`

Continued on next page

Table B.1 (continued)

Library	Commit
tachyon	`e7b13063c6f110b9851655cca740ee62ffb41137`
Webcrypto	npm `@peculiar/webcrypto@1.5.0`[b]
wolfSSL	`7d1516f4db4864033abf8b23af2903c3ba6da2ad`

[a] Bitcoin Core was not checked out separately; its elliptic-curve operations are provided by `libsecp256k1`, analyzed at the commit listed for that entry.
[b] Obtained via `npm install` rather than a Git checkout: `@peculiar/webcrypto@1.5.0`.
[c] Hosted on Bitbucket rather than GitHub.



NTNU

Norwegian University of
Science and Technology