Masaryk University
Faculty of Informatics

PYECSCA: Reverse-engineering black-box Elliptic Curve Cryptography implementations

Master's Thesis

Bc. Ján Jančár

Brno, Spring 2020

Masaryk University
Faculty of Informatics

PYECSCA: Reverse-engineering black-box Elliptic Curve Cryptography implementations

Master's Thesis

Bc. Ján Jančár

Brno, Spring 2020

Declaration

Hereby I declare that this thesis is my original work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Ján Jančár

Advisor: doc. RNDr. Petr Švenda, Ph.D.

Acknowledgements

A great thanks to my advisor doc. RNDr. Petr Švenda, Ph.D. for allowing me to pursue such an ambitious topic and for guidance along the way.

I would like to thank Dr. Lejla Batina, Ph.D. for inviting me for an internship and welcoming me at the SCA lab at Radboud University. Thanks to Niels, Pedro, Leo and Łukasz for helping me in their lab, and thanks to everybody at the DiS group for their welcoming coffee breaks and borrels.

I would also like to thank my partner, for the many ideas born out of discussions with her, including the main research question of the thesis, and for her support.

This thesis was supported by the Grant Agency of Masaryk University, grant MUNI/C/1701/2018, this support was very appreciated.

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures. This access was greatly appreciated.

Abstract

The usual goals of offensive Side Channel Analysis are mostly two-fold. One is to assess and quantify the presence of side channel leakage, the other is to assume leakage and target the secret key. The second goal often requires very precise knowledge of the implementation details, which might not be very extensive for algorithms like AES or RSA. However, the space of possible equivalent implementations of Elliptic Curve Cryptography (ECC) is much larger, making such an attack difficult to mount without the knowledge of actual implementation.

We present an open-source toolkit for reverse engineering of black-box implementations of Elliptic Curve Cryptography. The toolkit implements state-of-the-art utilities for Side Channel Analysis, such as trace alignment, filtering, statistical tests and attacks. It is able to synthesize ECC implementations for micro-controllers given any implementation configuration. It also allows for precise simulation of ECC operations given any configuration, down to intermediate values and ordering.

Keywords

elliptic curve cryptography, ECDH, ECDSA, smart cards, JavaCard, side channel analysis, power analysis, reverse-engineering

Contents

Introduction **1**

Notation **3**

1 Elliptic curve cryptography **4**
1.1 Elliptic curves 4
1.2 Cryptosystems 10
1.3 Implementation details 17

2 Side-channel attacks **27**
2.1 Side-channels 28
2.2 Simple SCA 30
2.3 Advanced SCA 32
2.4 Countermeasures 36
2.5 Reverse-engineering 38

3 PYECSCA: Python Elliptic Curve Side Channel Analysis toolkit **39**
3.1 Architecture and functionality 39
3.2 Cryptographic library analysis 58
3.3 Reverse-engineering methods 62
3.4 Evaluation 67

Conclusions **75**

Appendices **76**
Data attachment 77
EFD data 78
Additional figures 81

Bibliography **84**

Introduction

Side-channel analysis, the study of secret information leakage of an algorithm running on a device through side channels like power consumption, is a fast-moving field. It was pioneered in 1999 by the paper of Kocher et al. [1] on Differential Power Attacks (DPA). Some of the recent advances were in regards to analysis and attacks on Elliptic Curve Cryptography (ECC), but a large part of the focus remains on symmetric algorithms such as AES or even 3DES. Asymmetric analysis often stops at RSA, short of ECC. Elliptic curve cryptographic algorithms are very hard to implement correctly and securely, and even harder to be resistant to side-channel attacks, as is evident from the history of attacks on them [2]. A naive implementation would leak the value of the whole private key with just one look at a power-trace of an operation.

Even though the focus of side-channel analysis has not been on elliptic curves for long, there are side-channel attacks targeting elliptic curve cryptography and attacks that are generic enough to be also applicable to elliptic curve cryptography. However, in regards to tools and the practical usability and readiness of attacks, the situation is worse. The tools are very scattered and most publications of attacks might not even provide examples, code, tools or data. There are also toolkits for side-channel analysis, however most are proprietary and prohibitively expensive (Riscure Inspector [3], Rambus DPAWS [4]) and contain very little functionality specific and usable with elliptic curve cryptography (the open-source ChipWhisperer framework [5], lascar [6], scared [7] or Jlsca [8]). Thus security researchers developing a novel attack have to either resort to proprietary tools, or reinvent the wheel many times. The same scarcity of tools is also present in datasets, while there are several well-known datasets [9, 10] for use in attack analysis and development against AES, there are no such datasets available for ECC.

The usual goals of offensive Side-Channel Analysis are mostly two-fold. One is to assess and quantify the presence of side-channel leakage, and the other is to assume leakage and target the secret key. Both approaches often require very precise knowledge of the implementation details. This knowledge is usually easy to guess for many cryptographic algorithms, like AES or RSA. However, the space of possible equivalent implementations of elliptic curve cryptography is much larger, making such an attack difficult to mount without the knowledge of the actual implementation. Furthermore, existing work on attacks assumes the attacker has all of this knowledge and does not solve the problem of obtaining it, given a black-box implementation.

Our contribution

We have developed an extensive open-source toolkit for side-channel analysis of black-box elliptic curve cryptography implementations, titled **pyecsca** (**P**ython **E**lliptic **C**urve **S**ide-**C**hannel **A**nalysis toolkit). This toolkit is also able to:

- Enumerate millions of implementation configurations of ECC, satisfying given properties, built from modular components such that new can be added and used in configurations.
- Synthesize, build, query and trace implementations of ECC for microcontrollers, given any implementation configuration.
- Simulate and trace the execution of key generation, ECDH and ECDSA, down to intermediate values of finite field operations.
- Reverse-engineer parts of a configuration of a black-box implementation of ECC, through side-channels only.
- Acquire, process and visualize power traces from the PicoScope and ChipWhisperer branded oscilloscopes.

We introduce the idea of reverse-engineering implementation details of black-box elliptic curve cryptography implementations through side-channels and provide several novel methods for this reverse-engineering. We evaluate a subset of these methods on both custom implementations, where the details are known, and on commercial smartcards that implement ECC.

Overview

Chapter 1 presents an overview of elliptic curve cryptography, from the mathematical basis of elliptic curves (Section 1.1) through cryptosystems (Section 1.2) to details of their implementations (Section 1.3). In Chapter 2 we summarize the field of side-channel attacks, focusing on aspects applicable to asymmetric cryptosystems and attacks on elliptic curve cryptosystems. The **pyecsca** toolkit for side-channel analysis of black-box elliptic curve cryptography implementations, which we developed, is showcased in Chapter 3.

Notation

Elliptic curves

$\mathcal{E}_E$	an elliptic curve in Edwards form
$\mathcal{E}_M$	an elliptic curve in Montgomery form
$\mathcal{E}_{SW}$	an elliptic curve in short-Weierstrass form
$\mathcal{E}_{TE}$	an elliptic curve in twisted-Edwards form
$\mathcal{E}_W$	a Weierstrass curve
$\mathbb{K}$	a field
$\mathcal{E}(\mathbb{K})$	a group of $\mathbb{K}$ rational points on an elliptic curve $\mathcal{E}$
$\mathcal{E}/\mathbb{K}$	an elliptic curve over the field $\mathbb{K}$
$\mathcal{O}$	a point at infinity
ord	the order function
G	a point of prime order n on $\mathcal{E}(\mathbb{F}_p)$
p	an odd prime

Side-channel attacks

Δ	a distinguisher
$\mathbf{m}$	leakage modelling function
φ	the Euler's totient function
M	a number of samples in a side-channel trace
N	a number of collected side-channel traces
$O(s, x)$	output of a computation dependent on the secret subpart s and input x
X	a random variable
x	a realization of the random variable X
$(\mathbf{l}_j)_j$	a collection of N measurements $j = 1, \ldots, N$
$\mathbf{L}$	a multivariate random variable representing the leakage
ρ	the Pearson's correlation coefficient
$\hat{s}$	a hypothesis on the value of the secret sub-part s
s	a secret sub-part

1 Elliptic curve cryptography

Elliptic curves are one of the objects of mathematics that are very rich in properties and connections to other mathematical objects. They lie at the connection of algebraic number theory and algebraic geometry and find use in integer factorization, cryptography or in the proof of the famous Fermat's Last Theorem [11].

In the following chapter, we explore the use of elliptic curves over finite fields in cryptography, from theoretical perspectives to practical implementation details.

1.1 ∞ Elliptic curves

Definition 1. An *elliptic curve* is nonsingular plane algebraic curve of genus one.

A full Weierstrass model of an elliptic curve E over a field $\mathbb{K}$ is given by coefficients $a_1, a_2, a_3, a_4, a_6 \in \mathbb{K}$ in the following affine equation [12]:

$$\mathcal{E}_W/\mathbb{K}: \qquad y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1.1}$$

where for every affine point $(x', y') \in \overline{\mathbb{K}}^2$ for which the above equation holds the partial derivatives $2y' + a_1x' + a_3$ and $3x'^2 + 2a_2x' + a_4 - a_1y'$ do not vanish simultaneously [12]. A projective closure can be constructed by complementing the powers in the affine equation to degree three:

$$\mathcal{E}_W/\mathbb{K}: \qquad Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{1.2}$$

In this form, points have projective coordinates, e.g. points are triplets $(X : Y : Z) \in \overline{\mathbb{K}}^3 \backslash (0,0,0)$ which denote a class of $\overline{\mathbb{K}}^3 \backslash (0,0,0)$ modulo the relation:

$$(X_1 : Y_1 : Z_1) \sim (X_2 : Y_2 : Z_2) \iff \exists \lambda \in \overline{\mathbb{K}}^* \text{such that:} \\ (X_1, Y_1, Z_1) = (\lambda X_2, \lambda Y_2, \lambda Z_2)$$

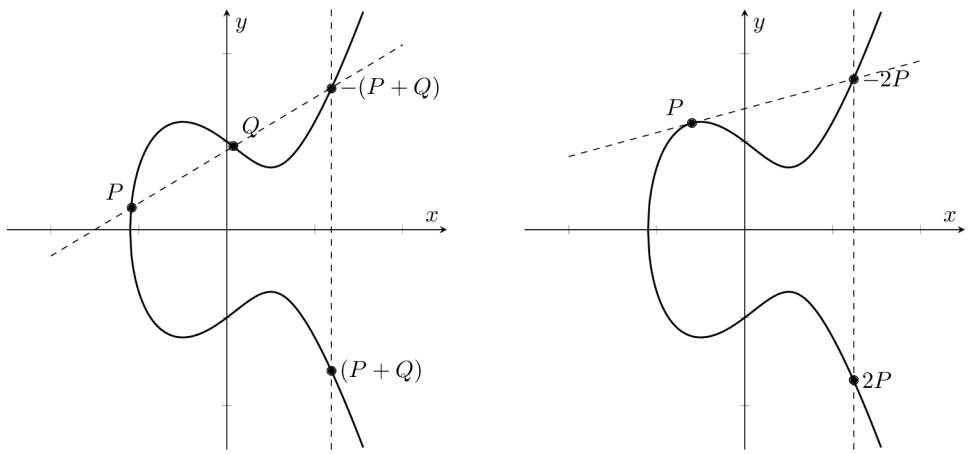
This means that each affine point has many representations in projective coordinates. A simple mapping between affine points and projective points exists. An affine point (x, y) maps to $(x : y : 1)$ and a projective point $(X : Y : Z)$ maps to $(X/Z, Y/Z)$. This mapping is clearly possible for all projective points except those with $Z = 0$. This however is really only a single point on the curve, the

point at infinity, denoted $\mathcal{O} = (0 : 1 : 0)$. The existence of the point at infinity and its non-representability in affine coordinates is an important detail from an implementation perspective.

To understand the algebraic structure of an elliptic curve, we introduce the set of $\mathbb{K}$-rational points on an elliptic curve:

Definition 2. The *set of* $\mathbb{K}$*-rational points* of an elliptic curve $\mathcal{E}$ is $\mathcal{E}(\mathbb{K}) = \{(x, y) \in \mathbb{K}^2 | y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\} \cup \{\mathcal{O}\}$.

As an elliptic curve is an abelian variety, its set of $\mathbb{K}$-rational points forms an abelian group. The addition law on this group has both a geometric and arithmetic form. Its geometric form is known as the chord-and-tangent rule and is most visible when we fix $a_1 = a_3 = 0$ and $\text{char}(\mathbb{K}) \neq 2$. When adding two distinct points P and Q, first draw a chord through them, this line intersects the curve at a third point, the sum of the two points is the reflection of this point about the x-axis (see Figure 1.1a). The doubling of a point is done by first drawing a tangent to the curve at the point P, which intersects the curve at a third point, the result of doubling is the reflection of this point about the x-axis (see Figure 1.1b) [13]. The point at infinity serves as the neutral element of the group. In the geometric form of the addition law, this simply signifies it is a single point that is at infinity, above and below all of the points.



(a) Point addition. (b) Point doubling.

Figure 1.1: Group law on an elliptic curve over $\mathbb{R}$, with $a_1 = a_3 = 0$.

The arithmetic form of the addition law follows from the geometric one. The slope of the chord or tangent is computed a then the line is intersected with the curve equation giving, for $P = (x_1, y_1)$ and $Q = (x_2, y_2)$:

$$P + Q = (\lambda^2 + a_1\lambda - a_2 - x_1 - x_2, \lambda(x_1 - x_3) - y_1 - a_1x_3 - a_3)$$
$$\lambda = \begin{cases} \dfrac{y_1 - y_2}{x_1 - x_2} & \text{if } P \neq \pm Q \\ \dfrac{3x_1^2 + 2a_2x_1 + a_4 - a_1y_1}{2y_1 + a_1x_1 + a_3} & \text{if } P = Q \end{cases}$$

The negation of point P is computed as $-P = (x_1, -y_1 - a_1x_1 - a_3)$. These formulas can be generalized to use projective coordinates, in which case the divisions necessary for computing the slope are replaced by multiplications, which is a significant performance improvement in implementations. It is also important to note that since the point at infinity does not have an affine representation in the Weierstrass model, the above formulas need exceptional cases when one or both of the input points are the point at infinity.

As we now have an addition law, we can define the multiplication of a point by a scalar, often called *scalar multiplication*, for any $k \in \mathbb{N}$:

$$\begin{aligned} [k] : \mathcal{E}(\mathbb{K}) &\to \mathcal{E}(\mathbb{K}) \\ P &\mapsto [k]P = \underbrace{P + P + \ldots + P}_{k \text{ times}} \end{aligned}$$

1.1.1 Elliptic curves over $\mathbb{F}_p$

Elliptic curves used in cryptography are always defined over a finite field $\mathbb{F}_q$ which is usually either a prime field $\mathbb{F}_p$ or a binary extension field $\mathbb{F}_{2^m}$. In the rest of this work, we will only work with elliptic curves over a prime field $\mathbb{F}_p$ of large characteristic (specifically $p \notin \{2, 3\}$), as those see the most use in modern cryptographic protocols.

The size of the group of $\mathbb{F}_p$-rational points on an elliptic curve over a finite field $\mathbb{F}_p$ is bound by the Hasse interval, for $n = |\mathcal{E}(\mathbb{F}_p)|$ [12]:

$$|n - (p + 1)| \leq 2\sqrt{p}$$

The structure of the group of $\mathbb{F}_p$-rational points on an elliptic curve over a finite field $\mathbb{F}_p$ is well-known and rather simple. This group is either cyclic or isomorphic to a product of two cyclic groups:

$$\mathcal{E}(\mathbb{F}_p) \simeq \mathbb{Z}_{d_1} \times \mathbb{Z}_{d_2}$$

with $d_1 | d_2$ and $d_1 | p - 1$ [12].

As elliptic curves used in cryptography are defined over large finite fields and have a large number of points, the scalars used in scalar multiplication are also

large. Thus a scalar multiplication algorithm cannot just add k times, instead most scalar multiplication algorithms scan the scalar bit by bit and produce the result in number of operations linear in the bit-size [13]. A basic example of such an algorithm is the double-and-add algorithm:

Algorithm 1 Left-to-right double-and-add (complete) [13]

```
function LTR(G, k = (k_l, ..., k_0)_2)
    R = ∞
    for i = l downto 0 do
        R = dbl(R)
        if k_i = 1 then
            R = add(R, G)
    return R    // = [k]G
```

1.1.2 Elliptic curve models

Over a field $\mathbb{F}_p$ with characteristic not 2 or 3, the full Weierstrass model of an elliptic curve as given in Equation (1.1) can be simplified into the short-Weierstrass model [12]:

$$\mathcal{E}/\mathbb{F}_p : \qquad y^2 = x^3 + ax + b$$

This means that for every elliptic curve defined by the full Weierstrass model over a suitable prime field, there exists a curve isomorphism, over said prime field, between it and a curve in the short-Weierstrass model.

A curve isomorphism between two elliptic curves over a finite field $\mathbb{F}_p$ is a rational map ψ with coefficients in $\mathbb{F}_p$ between the curves that is bijective and has $ker(\psi) = \{\mathcal{O}\}$.

Because of the existence of this shorter model curves in the full Weierstrass model are not used in cryptography. Instead, short-Weierstrass curves are used. Other models exist for certain classes of curves, such as the Montgomery model [14], Edwards model [15], Twisted Edwards model [16] or Hessian model [17]. These models often provide performance or security advantages and are thus used being used more in modern cryptography. We discuss these in Subsection 1.3.1.

The notion of birational equivalence is important in understanding the different curve models, as some of them are only birationally equivalent but not isomorphic to an elliptic curve. Two elliptic curves E and E' over a finite field $\mathbb{F}_p$ are birationally equivalent iff there exists a rational map $\psi : \mathcal{E} \to \mathcal{E}'$ for which there exists a rational map $\psi' : \mathcal{E}' \to \mathcal{E}$ such that $\psi \circ \psi'$ is equivalent to identity

on $\mathcal{E}'$ and $\psi' \circ \psi$ is equivalent to identity on $\mathcal{E}$. A birational equivalence can be thought of as a curve isomorphism with exceptional points.

Short-Weierstrass curves

The short-Weierstrass model of an elliptic curve is the oldest one of those presented in this work, it is also the only one directly isomorphic to an elliptic curve. The isomorphism is given by:

$$\mathcal{E}_W \to \mathcal{E}_{SW} : (x, y) \mapsto \left(\frac{x - 3a_1^2 - 12a_2}{36}, \frac{y - 3a_1 x}{216} - \frac{a_1^3 + 4a_1 a_2 - 12a_3}{24} \right)$$

and the curve equation is given by:

$$\mathcal{E}_{SW}/\mathbb{F}_p : \qquad y^2 = x^3 + ax + b$$

with a and b in $\mathbb{F}_p$ such that $4a^3 + 27b^2 \neq 0$, which is required for nonsingularity.

Montgomery curves

The Montgomery model was introduced by Montgomery [14] for speeding up the ECM integer factorization method, which uses elliptic curves. It allows for the use of a coordinate system that leaves out the y coordinate, thereby representing a class of two points with the one x coordinate. In this coordinate system, there exists a fast formula for computing the addition of two points and a doubling of one of them simultaneously, provided that their difference is known. This operation is called a ladder step and is used in the Montgomery ladder algorithm.

The curve equation is given by:

$$\mathcal{E}_M/\mathbb{F}_p : \qquad By^2 = x^3 + Ax^2 + x$$

with $B \neq 0$ and $A^2 \neq 4$, which is required for nonsingularity.

Montgomery curves represent only a subset of curves representable in the short-Weierstrass model. Namely, all Montgomery curves over $\mathbb{F}_p$ have their number of $\mathbb{F}_p$-rational points divisible by four. This condition on short-Weierstrass curves can be expressed algebraicaly as requiring the existence of α and β in $\mathbb{F}_p$ such that $\alpha^3 + a\alpha + b = 0$ and $3\alpha^2 + a = \beta^2$ [18, 19]. The map is then given as:

$$\mathcal{E}_{SW} \to \mathcal{E}_M : \quad (x, y) \mapsto \left(\frac{x - \alpha}{\beta}, \frac{y}{\beta} \right)$$

where $\mathcal{E}_M$ has the coefficients $A = 3\alpha/\beta$ and $B = 1/\beta$.

Edwards curves

The Edwards model was introduced by Edwards [15] and later extended by Bernstein; Lange [20] due to having complete and unified addition formulas for some parameters. An addition formula is *unified* if it correctly handles the addition of the point to itself, so it also can be used as a doubling formula. The *completeness* property of a formula signifies that it has no exceptions and will correctly compute the result with any points on the curve, specifically also the point at infinity and the addition of a point to itself.

An Edwards curve has the equation:

$$\mathcal{E}_E/\mathbb{F}_p : \qquad x^2 + y^2 = c^2(1 + dx^2y^2)$$

with c and d in $\mathbb{F}_p$ with $cd(1-dc^4) \neq 0$. There exist complete addition formulas if d is not a square in $\mathbb{F}_p$ [20]. Even with d square the addition formulas on Edwards curves are unified. The point at infinity is representable in affine coordinates on Edwards curves as $\mathcal{O} = (0, c)$.

Twisted-Edwards curves

The Twisted Edwards model was introduced by Bernstein et al. [16] as a generalization of Edwards curves. Twisted Edwards curves are twists of Edwards curves. They represent the same isomorphism class as Montgomery curves and thus represent a subset of curves representable in the short-Weierstrass model, yet they represent more curves than Edwards curves.

The curve equation is:

$$\mathcal{E}_{TE}/\mathbb{F}_p : \qquad ax^2 + y^2 = 1 + dx^2y^2$$

with a and d distinct nonzero elements in $\mathbb{F}_p$, which can be seen as a quadratic twist of the Edwards curve $\mathcal{E}_E/\mathbb{F}_p : x^2 + y^2 = 1 + (d/a)x^2y^2$. If a is a square in $\mathbb{F}_p$ and d a non-square in $\mathbb{F}_p$ the Twisted-Edwards addition formulas are complete, as then the curve is isomorphic to an Edwards curve with complete addition formulas. Nevertheless, even without this condition there are unified addition formulas in the Twisted-Edwards model [21, 16]. The point at infinity is representable in affine coordinates on twisted-Edwards curves as $\mathcal{O} = (0, 1)$.

The map from the short-Weierstrass model is more involved, but requires the same setup as that for the Montgomery model. Let α and β in $\mathbb{F}_p$ such that $\alpha^3 + a'\alpha + b' = 0$ and $3\alpha^2 + a' = \beta^2$ for an elliptic curve in the short-Weierstrass model $\mathcal{E}_{SW} : y^2 = x^3 + a'x + b'$. The map, as given below has two exceptions:

$$\mathcal{E}_{SW} \to \mathcal{E}_{TE}: \quad P \mapsto (u,v) = \begin{cases} (0,1), & \text{if } P = \mathcal{O} \\ (0,-1), & \text{if } P = (\alpha, 0) \\ \left(\frac{x-\alpha}{y}, \frac{x-\alpha-\beta}{x-\alpha+\beta}\right), & \text{else } P = (x,y) \end{cases}$$

for the point at infinity (which has an affine representation in the Twisted-Edwards model) and the point of order two $(\alpha, 0) \in \mathcal{E}_{SW}$. The coefficients of the resulting Twisted-Edwards curve are $a = 3\alpha + 2\beta$ and $d = 3\alpha - 2\beta$.

$$\begin{tikzcd}
\mathcal{E}_{SW} & \mathcal{E}_{TE} \arrow[l] \arrow[dl] \\
\mathcal{E}_{M} \arrow[u] \arrow[ur] & \mathcal{E}_{E} \arrow[l] \arrow[ul] \arrow[u]
\end{tikzcd}$$

Figure 1.2: Birational equivalence of curve models. An arrow from a particular model signifies that all curves in that model are birationally equivalent to curves in the target one.

1.2 🔒 Cryptosystems

There are many elliptic curve based cryptosystems that utilize different properties of elliptic curves to construct public-key cryptosystems. Classical elliptic curve discrete-logarithm based systems were created by replacing the multiplicative group of integers modulo a prime by a group of points on an elliptic curve over a finite field, these are the focus of this work. They only utilize the group structure of the elliptic curve and are based on some variation of the discrete-logarithm problem.

Definition 3. The *elliptic curve discrete-logarithm problem* (ECDLP) is the problem of finding the discrete logarithm of a point on an elliptic curve:

Given $G \in \mathcal{E}(\mathbb{K})$ a generator of a sub-group of $\mathcal{E}(\mathbb{K})$ and $P = [x]G$, find x [12].

The ECDLP is conjectured to be hard on certain classes of well-chosen curves. The security conditions have complex reasons. However, they generally require the curve order to be a large prime not equal to the order of the base field and a large embedding degree of the curve. These conditions ensure that the only known algorithms to solve ECDLP are generic discrete-logarithm problem algorithms

with slight speedups. The fastest of which is the Pollard's rho algorithm, with a run-time of $\mathcal{O}(\sqrt{|\mathcal{E}(\mathbb{K})|})$, which is exponential in the bit-size of the curve [22, 23]. This highlights the fact that well-chosen elliptic curves are very close to a generic group, as almost no algebraic structure can be used to speed up the discrete-logarithm search.

Specifically, there are no efficient (sub-exponential) index-calculus algorithms for the ECDLP on well-chosen elliptic curves over prime fields. Due to this, classical elliptic curve cryptosystems use parameters with much smaller bit-sizes than finite field discrete-logarithm based cryptosystems of comparable security levels. The current record for solving an ECDLP on a well-chosen elliptic curve over a prime field is on a 112-bit curve by Bos et al. [24].

Some classical elliptic curve protocols, like ECDH or X25519, rely on the security of related problems, such as the computational Diffie-Hellman problem.

Definition 4. The *elliptic curve computational Diffie-Hellman problem* (ECCDH) is the problem of computing a shared key from two shares in a Diffie-Hellman protocol.

Given $G \in \mathcal{E}(\mathbb{K})$ a generator of a sub-group of $\mathcal{E}(\mathbb{K})$, $P = [x_a]G$ and $Q = [x_b]G$, compute $[x_a][x_b]G$.

The applications of elliptic curves in public-key cryptography extend much farther than the aforementioned problems and cryptosystems, to pairing and isogeny based cryptography. However, we will not delve into these topics in this work.

Pairing based cryptography utilizes the existence of pairings on elliptic curves to build digital signature systems with short signatures (Boneh-Lynn-Shacham), group signature systems or identity-based encryption. A pairing is a bilinear non-degenerate mapping from two prime order cyclic groups to a third group. Elliptic curves offer two such pairings, the Weil pairing and the Tate pairing [25].

Isogeny based cryptography is a relatively new field of cryptography, which utilizes isogenies of elliptic curves, to create post-quantum public-key cryptosystems, including key exchange (SIDH, CSIDH) and digital signatures (CFiSh). The isogeny based cryptosystems have very small keys and signatures but are currently rather slow [26].

1.2.1 Key exchange

The key exchange family of cryptographic primitives is one where two or more parties exchange or derive a shared secret key over a public channel, securely. The first example of such a protocol arose with the seminal work of Diffie; Hellman [27]. Both of the following two primitives borrow heavily from the Diffie-Hellman

primitive. They also both assume proper authentication of all parties and messages and are insecure in the presence of an active man-in-the-middle attacker.

ECDH

The *elliptic curve Diffie-Hellman* cryptosystem (ECDH) is a direct generalization of the Diffie-Hellman cryptosystem which replaces the multiplicative group of integers modulo a prime with the group of rational points on an elliptic curve [28, 29, 30, 31]. The cryptosystem requires the parties to agree on what domain parameters will be used before or during the key exchange. There are various sets of standardized elliptic curves, to ensure interoperability of implementations [32, 33]. Most curves standardized for use in ECDH are given in the short-Weierstrass model, as this is a somewhat older cryptosystem which predates the other curve models. The points are usually exchanged in affine coordinates and are either uncompressed (both coordinates are included) or compressed. Point compression uses the vertical symmetry of short-Weierstrass curves, to only transmit the x-coordinate of the point and one bit to encode which y coordinate to take, out of the two possible values [30, 32].

Algorithm 2 ECDH

function KeyGen(params $= (\mathcal{E}(\mathbb{F}_p), G \in \mathcal{E}(\mathbb{F}_p), n = |G|)$)
  t $\xleftarrow{\$} \mathbb{Z}_n$
  $P \leftarrow [t]G$
  return (private key t, public key P)

function Derive(params, private key $t_a \in \mathbb{Z}_n$, public key $P_b \in \mathcal{E}(\mathbb{F}_p)$)
  Verify that P_b is a point on $\mathcal{E}(\mathbb{F}_p)$ and $P_b \neq \mathcal{O}$
  $Q \leftarrow [t_a]P_b$
  return $x(Q)$

The cryptosystem relies on the commutativity of the scalar multiplication action, where for a and $b \in \mathbb{Z}$ and $G \in \mathcal{E}(\mathbb{F}_p)$ we have $[a][b]G = [b][a]G$.

X25519 and X448

The *X25519* key exchange protocol was specified under the name Curve25519 in a paper by Bernstein [34]; however, now that name refers to the specific curve used in the protocol. The protocol is similar to ECDH in structure and uses the same commutative action of scalar multiplication to build a Diffie-Hellman based key exchange cryptosystem. It aims to be heavily optimized yet simple to

implement in constant-time and without implementation pitfalls. It achieves high performance in several ways:

- Uses a prime $p = 2^{255} - 19$, hence the name, which admits fast modular arithmetic in software, when implemented using floating point registers.
- Uses a Montgomery curve with small coefficients, which allows for very efficient x-coordinate only arithmetic and for fast multiplication by the curve coefficient.
- It does not require public-key validation, as every 32-byte string is a valid public key.

To avoid implementation pitfalls and vulnerabilities, it makes further choices:

- Fixes certain bits of the private key to avoid small-subgroup attacks and to make it fixed length [35].
- Uses a Montgomery curve with x-coordinate only arithmetic to avoid invalid curve attacks by not relying on public-key validation, which is unnecessary in this case [36].
- Ensures the curve has an equally secure twist, as without public-key validation, it would otherwise be vulnerable to invalid curve attacks via the twist [37].

Algorithm 3 Curve25519 [34]

Parameters:
$E_{\text{Curve25519}}$: $y^2 = x^3 + 486662x^2 + x$ over $\mathbb{F}_p$ with $p = 2^{255} - 19$
map $X_0 : E_{\text{Curve25519}} \to E_{\text{Curve25519}}$, s.t. $X_0(\mathcal{O}) = 0$ and $X_0(x, y) = x$

function Curve25519(private key $\underline{n}$, public key $\underline{q} \in \{0, 1, \ldots, 255\}^{32}$)
$Q \in E_{\text{Curve25519}}$ such that $X_0(Q) \equiv q \mod p$
$s \leftarrow X_0([n]Q)$
return $\underline{s}$

function KeyGen
$n \xleftarrow{\$} 2^{254} + 8\{0, 1, 2, 3, \ldots, 2^{251} - 1\}$
$\underline{q} \leftarrow$ Curve25519($\underline{n}, \underline{9}$)
return private key $\underline{n}$, public key $\underline{q}$

function Derive(private key $\underline{n}$, public key $\underline{q} \in \{0, 1, \ldots, 255\}^{32}$)
$\underline{s} \leftarrow$ Curve25519($\underline{n}, \underline{q}$)
return $\underline{s}$

While originally designed to be only used with the Curve25519 Montgomery curve, it has been extended to the Montgomery curve Curve448-Goldilocks by Hamburg [38] as X448 in RFC7748 [39]. This restriction to Montgomery curves is no absolute and there is nothing that would stop an implementation from working in a different model and transforming points on input and output. In fact, the short-Weierstrass versions of Curve25519 are named Wei25519 and are already being used [21].

1.2.2 Signatures

Digital signatures are a public-key cryptographic primitive that provides authenticity, integrity and non-repudiation properties. That is a certification that a party possessing a certain private key had signed a particular message, which was not tampered with, while allowing the verification of this certification using the private key. The following two signature systems represent the two most widely used elliptic curve-based signature systems. In contrast to key exchange, where the primitives rely on ECCDH problem, these signature systems rely on the ECDLP [40].

ECDSA

The *Elliptic Curve Digital Signature Algorithm* (ECDSA) is a straight-forward extension of the Digital Signature Algorithm (DSA) to the elliptic curve setting, replacing the multiplicative group of integers modulo a prime with the group of rational points on an elliptic curve, very much akin to how ECDH is an extension of Diffie-Hellman [41, 42]. It was standardized in the *FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION 186-2 Digital Signature Standard (DSS)* [41] along with a set of short-Weierstrass elliptic curves recommended for use, which are now known as the NIST curves. Like DSA, it is a signature primitive very similar to the Schnorr signature primitive [43], the very close similarity yet not equality is intentional, the Schnorr signature primitive was patented at the time of standardization of both DSA and ECDSA.

While the NIST specified curves for use with ECDSA are prime order and as such cannot be directly used with the other curve models mentioned in Subsection 1.1.2, the primitive only requires that the curve has one point of large prime order and restricts its cofactor below a certain bound [41, Table 1]. The restriction to short-Weierstrass curves in the standard is also only for interoperability reasons and if a suitable curve is used, nothing stops an implementation from using whatever curve model it desires, provided it transforms its inputs and outputs properly.

Like DSA, ECDSA is a randomized signature primitive with appendix [44], which means that it uses randomness during signing and that the message cannot be recovered from the signature. The randomness in ECDSA is in the form of a random nonce $k \in \mathbb{Z}_n^*$ used during signing (see Algorithm 4 Sign) which needs to be only used once and generated uniformly randomly in $\mathbb{Z}_n^*$. These requirements are the source of many issues with practical implementations of ECDSA as well as of many exploitable attacks.

Algorithm 4 ECDSA

function KeyGen(params $= (\mathcal{E}(\mathbb{F}_p), G \in \mathcal{E}(\mathbb{F}_p), n = |G|)$)
  t $\xleftarrow{\$} \mathbb{Z}_n$
  $P \leftarrow [t]G$
  return (private key t, public key P)
function H(params, message $m \in \{0,1\}^*$)
  $h =$ Hash(m)
  $L_n =$ the bit-length of n (the order of G)
  return leftmost L_n bits of h as a big-endian unsigned integer
function Sign(params, private key t, message m)
  $k \xleftarrow{\$} \mathbb{Z}_n^*$
  $r \leftarrow x([k]G) \mod n$
  $s \leftarrow k^{-1}(\mathrm{H}(m) + rt) \mod n$
  if $r = 0$ **or** $s = 0$ **then**
    Go to the first step
  return signature (r, s)
function Verify(params, public key P, message m, signature (r, s))
  Verify that $P \in \mathcal{E}(\mathbb{F}_p)$; $P \neq \mathcal{O}$ and $[n]P = \mathcal{O}$
  Verify that $r, s \in \mathbb{Z}_n^*$
  $u_1 \leftarrow \mathrm{H}(m)s^{-1} \mod n$
  $u_2 \leftarrow rs^{-1} \mod n$
  $Q \leftarrow [u_1]G + [u_2]P$
  Verify that $Q \neq \mathcal{O}$
  return $r \stackrel{?}{\equiv} x(Q) \mod n$

Nonce reuse for different messages leads to trivial private key recovery, take two signatures $(r_1, s_1), (r_2, s_2)$ on messages m_1, m_2, respectively, that used the same nonce k and were made by the same private key t. Clearly $r_1 = r_2 = x([k]G)$

and thus we get the two equations with two unknowns:

$$s_1 = k^{-1}(\mathrm{H}(m_1) + tr) \mod n$$
$$s_2 = k^{-1}(\mathrm{H}(m_2) + tr) \mod n$$

which can be easily solved to get the private key as $t = \frac{s_2\mathrm{H}(m_1) - s_1\mathrm{H}(m_2)}{r(s_1 - s_2)}$. A similar situation arises if the nonce k is known, the private key can be extracted if the message m and signature (r, s) is known as $t = \frac{sk - \mathrm{H}(m)}{r}$. These issues may seem far-fetched, yet nonce reuse broke the software signing mechanism on the Playstation console [45].

Full knowledge of the random nonce or nonce reuse are not necessary for ECDSA to break, even a small bias in the nonce distribution or partial information about the nonce values, such as a few bits, can be enough. Lattice attacks, which transform the key recovery problem given partial knowledge of nonces to an instance of a Shortest Vector Problem or a Closest Vector Problem were used in several practical attacks [2]. For example, the Chromebook H1 chip generated nonces which only contained 64 bits of randomness, instead of the full 256 bits that were required [46].

Deterministic ECDSA, as specified in RFC6979 [47], solves some of the aforementioned issues by generating the nonces deterministically, from the message and the private key, without the need for randomness. However, even this method does not solve issues present when information about the nonce leaks through some side-channel or when even a deterministically generated nonce can be corrupted by a fault attack.

EdDSA

The *Edwards curve Digital Signature Algorithm* (EdDSA) was introduced in Bernstein et al. [48] and extended to more curves and variants in Bernstein et al. [49]. It uses the twisted-Edwards model to construct a high-speed signature primitive. Its most common instance is Ed25519, which use a twisted-Edwards curve that is birationally equivalent to Curve25519, which is a Montgomery curve. It was later also standardized in RFC8032 [50]. It uses deterministic generation of nonces, similar to deterministic ECDSA, and does not break even without collision resistance of the used hash function (in the PureEdDSA variant).

While it uses the twisted-Edwards model, an implementation could use any model if it properly transforms the points to and from the twisted-Edwards form.

Algorithm 5 EdDSA as originally defined in Bernstein et al. [48]

Parameters:
- $b \in \mathbb{N}$, hash function $H : \{0,1\}^* \to \{0,1\}^{2b}$
- prime power q with $q \mod 4 \equiv 3$
- $\mathcal{E}(\mathbb{F}_q)$ of the form: $-x^2 + y^2 = 1 + dx^2y^2$ with non-square d
- $B \in \mathcal{E}(\mathbb{F}_q)$ with $l \in \left[2^{b-4}, 2^{b-3}\right]$ a large prime such that $[l]B = \mathcal{O}$

function KeyGen
- $k \xleftarrow{\$} \{0,1\}^b$
- $(h_0, h_1, \ldots, h_{2b-1}) \leftarrow H(k)$
- $a \leftarrow b^{b-2} + \sum_{3 \le i \le b-3} 2^i h_i$
- $A \leftarrow [a]B$
- **return** private key k, public key $\underline{A}$

function Sign(message M, private key k)
- $(h_0, h_1, \ldots, h_{2b-1}) \leftarrow H(k)$
- $r \leftarrow H(h_b, \ldots, h_{2b-1}, M)$
- $R \leftarrow [r]B$
- $S \leftarrow (r + H(\underline{R}, \underline{A}, M)) \mod l$
- **return** signature $(\underline{R}, \underline{S})$

function Verify(message M, public key $\underline{A}$, signature $(\underline{R}, \underline{S})$)
- **return** $[8S]B \stackrel{?}{\equiv} [8]R + [8H(\underline{R}, \underline{A}, M)]A$

1.3 ⟨/⟩ Implementation details

The previous sections presented elements of elliptic curve cryptography in a more or less abstract way, even algorithm descriptions of cryptosystems operated using abstract notions of an elliptic curve, rational points on it and operations with them. This abstract view of elliptic curve cryptography is often detailed enough to work with it and analyze it but is not nearly detailed enough to study the security of concrete, hardware or software implementations of elliptic curve cryptography, especially when considering side-channel attacks.

This section builds on the abstract view by examining concrete implementation details. To implement any of the cryptosystems specified in Section 1.2 developers need to make a multitude of choices which impact the performance, security and overall design of the implementation. The choices include: what curve model to use, what scalar multiplier to use, what coordinate system or systems to use, what formulas to use or how to implement finite field arithmetic. These choices are highly connected, as the choice of a curve model restricts the set of possible coordinate systems to those applicable to a particular curve model,

which in turn restricts the set of possible formulas to those using said coordinate systems.

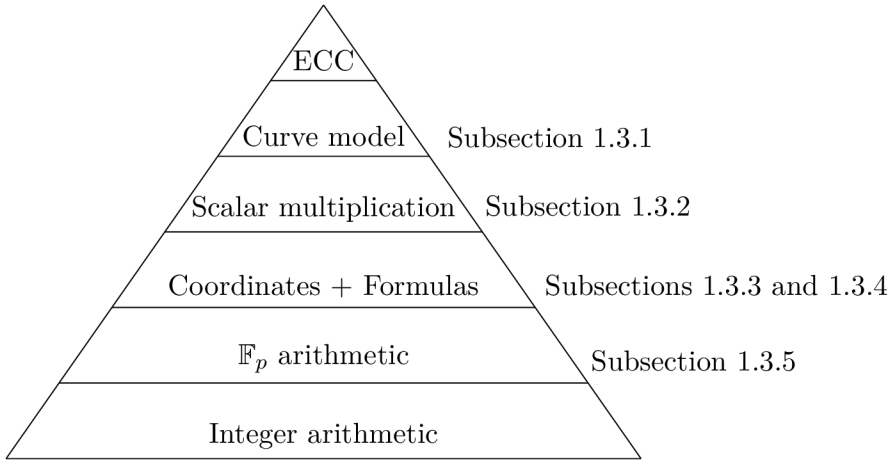


Figure 1.3: The usual layers of an ECC implementation.

1.3.1 Curve model

The first and most high-level choice in an ECC implementation is its choice of curve model. Cryptosystems specified in Section 1.2 do specify the curve models and encodings used for input and output, such that interoperability can be achieved. However, this is a limitation only for the input and output formats used, internally equivalences between curve models can be used to transform points and perform the computation on any compatible curve model. Figure 1.2 displays the unconditional equivalences between curve models, i.e. those mappings that are available for all curves of a certain class. For example, it is always possible to transform an Edwards curve into a Montgomery curve, but the converse is not true.

There are sometimes significant motivations for using a different curve model than the natural one given by the specification of the cryptosystem:

- **Performance.** Several curve models provide much more efficient point arithmetic than the often used short-Weierstrass model.

- **Completeness.** Until recently complete addition formulas were unavailable for the prime order curves in the short-Weierstrass model, they were introduced in 2016 by Renes et al. [51].

- **Platform limitations.** The Montgomery model was first used in a cryptosystem in 2006 [34], the Edwards model was introduced in 2007 [15]. With the somewhat slow-moving field of hardware cryptographic implementations, this means that hardware implementations using these models are still not widely available in products. For example, the JavaCard platform, a popular programmable smartcard platform, still only offers short-Weierstrass curve operations in its API. The general purpose integer arithmetic on most JavaCards is slow enough that using it to reimplement support for Edwards or Montgomery curve models is unthinkable. Thus, when one wants to use these curve models on this platform, the use of their equivalent short-Weierstrass form is unavoidable. Such an implementation already exists in the `jc_curve25519` package [52]. The topic of using limited APIs to perform elliptic curve operations on different models is also described in a draft RFC by Struik [21].

- **Existing stack.** When adding support for a new elliptic curve cryptosystem to a library, it might happen that the library already has elliptic curve support for some other cryptosystem or using some other curve model. When this happens, libraries often choose to reuse the existing code, as adding new code will increase the size and complexity of the library.

A popular example of the use of multiple curve models is Ristretto [53], which is a technique for constructing prime order elliptic curve groups from curves with a cofactor. It is named after a similar technique, Decaf [54], which focused on slightly different curves. Ristretto uses the twisted-Edwards, Montgomery and Jacobi Quartic curve models and transforms points between them using isogenies and isomorphisms.

1.3.2 Scalar multiplication algorithm

Scalar multipliers can be classified using the number and type of inputs they compute on.

- **Single-scalar.** Single-scalar multipliers perform the single operation $(n \in \mathbb{N}, P \in \mathcal{E}(\mathbb{F}_p)) \mapsto [n]P$ and are useful in KeyGen, Derive and Sign. They usually operate either on the fixed generator of a subgroup on the curve, or on public inputs. The scalar they are using to multiply is often secret, so they often use side-channel countermeasures and are intended to be timing and cache-constant.

  - **Fixed-base.** A fixed-base scalar multiplier performs the operation

$$\mathbb{N} \to \mathcal{E}(\mathbb{F}_p) : k \mapsto [k]G$$

where $G \in \mathcal{E}(\mathbb{F}_p)$ is a fixed (base)-point on the curve. As the point remains fixed, the multiplier might perform extensive precomputations with the point that will then lower the cost of individual scalar multiplications. Fixed-base algorithms are most often used in KeyGen and Sign (see Algorithms 2 and 4).

– **Variable-base.** A variable-base scalar multiplier performs the operation

$$\mathbb{N} \times \mathcal{E}(\mathbb{F}_p) \to \mathcal{E}(\mathbb{F}_p) : (k, P) \mapsto [k]P$$

where P is a variable point on the curve. As the point is variable it cannot use expensive precomputations, because their cost will not pay off if the point is used only once. These algorithms are most often used in Derive or sometimes in Verify.

- **Multi-scalar.** Multi-scalar multipliers usually perform the operation

$$\mathbb{N}^n \times \mathcal{E}(\mathbb{F}_p)^n \to \mathcal{E}(\mathbb{F}_p) : ((k_0, \ldots, k_n), (P_0, \ldots, P_n)) \mapsto \sum_{i=0}^{n} [k_i]P_i$$

where n is most often equal to two. Such an operation is often useful in Verify (see Algorithms 4 and 5) where two points are multiplied and added simulatenously. It is of course possible to implement a multi-scalar multiplier naively, by first multiplying the individual points and then adding them together. However, there are methods which interleave this process and achieve better performance, such as the colloquially known Shamir's trick, originally due to Straus [55].

One could also consider the distinction between fixed-scalar and variable-scalar multiplication algorithms, where addition chain precomputation could speed up the fixed-scalar one. However, this is not usually done in practice, and the only application would be in a Derive operation [12].

We further classify scalar multipliers by their type, we are intentionally vague in what 'type' means, as such categorizations are hard and algorithms might combine ideas from several types, for more see [44, 12, 13].

- **Basic.** Basic scalar multipliers scan the scalar bit by bit in either left-to-right or right-to-left fashion and are often called double-and-add scalar multipliers, akin to their square-and-multiply counterpart in multiplicative notation [13].

- **Windows.** Windowed scalar multipliers split the scalar into windows (tuples of digits). These windows can either be fixed, meaning that they divide

the scalar in a regular fashion, or sliding, in which case they skip over zero bits (there is really only one window sliding across the scalar). Windowed algorithms are parameterized by their window size $w \in \mathbb{N}$. The algorithms also do some precomputation, usually to compute all odd multiples of the point smaller than $2^w - 1$. However, this precomputation is cheap so it can also be applied in a variable-base scalar multiplier [13].

- **Combs.** Comb scalar multipliers perform significant precomputation and are thus intended to be used for fixed-base scalar multiplication. They are also parameterized by their width $w \in \mathbb{N}$ as they iterate over the scalar in a comb-like fashion, each iteration processes all bits of the scalar that are w bits apart. Combs are generally the fastest fixed-base scalar multipliers [56, 57].
- **Ladders.** Ladder multipliers are very similar to basic scalar multipliers in that they scan the scalar bit by bit and do not perform any precomputation. However, as introduced in Montgomery [14] they differ by using a ladder formula, by being regular by design and by using two temporary variables, the difference of which is kept constant throughout the algorithm.

Point negation is a very efficient operation in almost all curve models and coordinate systems; as such, it can be used to speed up scalar multiplication. Thus we classify scalar multipliers by their scalar representation.

- **Signed.** A signed scalar multiplier recodes the scalar into some signed representation before it is used in the multiplier. Often this representation is the non-adjacent form (NAF) in which no two consecutive digits are nonzero.
- **Unsigned.** An unsigned scalar multiplier might recode the scalar into some other representation than binary, but it is not signed and does not use point negation.

1.3.3 Coordinate system

Affine points on an elliptic curve in the curve models from Subsection 1.1.2 can always be represented using the affine coordinate system. However, the addition law in affine coordinates is quite ineffective, requiring costly field inversions. Also, the point at infinity might not be representable in affine coordinates, as is the case for the short-Weierstrass and Montgomery models. Luckily, there are other coordinate systems which have more effective formulas and have a representation for the point at infinity. The projective coordinate system for short-Weierstrass curves was already described in Equation (1.2).

Table 1.1 lists the coordinate systems for the selected curve models from the Explicit-Formulas Database, which is a database of formulas for various elliptic curve models by Lange; Bernstein [58]. For each coordinate system, we list its important properties: its form, whether the point at infinity has a representation in it, transformation to affine coordinates and any conditions for applicability. Having a representable point at infinity is a precondition for having complete formulas available, so it is an important property of a coordinate system. Several coordinate systems have conditions on curve parameters, which means that they only work for a subset of curves, however often have other desirable properties, i.e. speed.

Curve	Name	Form	Infinity	Affine	Conditions
$\mathcal{E}_{SW}$	affine	(x, y)	-		
	jacobian	$(X:Y:Z)$	$(1:1:0)$	$x = X/Z^2, y = Y/Z^3$	
	jacobian-0	$(X:Y:Z)$	$(1:1:0)$	$x = X/Z^2, y = Y/Z^3$	$a = 0$
	jacobian-3	$(X:Y:Z)$	$(1:1:0)$	$x = X/Z^2, y = Y/Z^3$	$a = -3$
	modified	$(X:Y:Z:T)$	-	$x = X/Z^2, y = Y/Z^3$	
	projective	$(X:Y:Z)$	$(0:1:0)$	$x = X/Z, y = Y/Z$	
	projective-1	$(X:Y:Z)$	$(0:1:0)$	$x = X/Z, y = Y/Z$	$a = -1$
	projective-3	$(X:Y:Z)$	$(0:1:0)$	$x = X/Z, y = Y/Z$	$a = -3$
	w12-0	$(X:Y:Z)$	-	$x = X/Z, y = Y/Z^2$	$b = 0$
	xyzz	$(X:Y:ZZ:ZZZ)$	-	$x = X/ZZ, y = Y/ZZZ$	
	xyzz-3	$(X:Y:ZZ:ZZZ)$	-	$x = X/ZZ, y = Y/ZZZ$	$a = -3$
	xz	$(X:Z)$	-	$x = X/Z^\dagger$	
$\mathcal{E}_M$	affine	(x, y)	-		
	xz	$(X:Z)$	-	$x = X/Z^\dagger$	
$\mathcal{E}_E$	affine	(x, y)	$(0, c)$		
	inverted	$(X:Y:Z)$	-	$x = Z/X, y = Z/Y$	
	projective	$(X:Y:Z)$	$(0:c:1)$	$x = X/Z, y = Y/Z$	
	yz	$(Y:Z)$	-	$y = Y/(\sqrt{d}Z)^\dagger$	d square, $c = 1$
	yzsquared	$(Y:Z)$	-	$y^2 = Y/(\sqrt{d}Z)^\dagger$	d square, $c = 1$
$\mathcal{E}_{TE}$	affine	(x, y)	$(0, 1)$		
	extended	$(X:Y:Z:T)$	$(0:1:0:1)$	$x = X/Z, y = Y/Z$	
	extended-1	$(X:Y:Z:T)$	$(0:1:0:1)$	$x = X/Z, y = Y/Z$	$a = -1$
	inverted	$(X:Y:Z)$	-	$x = Z/X, y = Z/Y$	
	projective	$(X:Y:Z)$	$(0:1:1)$	$x = X/Z, y = Y/Z$	

† The mapping to affine coordinates is not unique.

Table 1.1: Coordinate systems for the four selected curve models, from EFD [58] and their numerous original sources.

1.3.4 Formulas

While the main operation provided by the group of rational points on an elliptic curve is addition, in practice there are many formulas in the many coordinate systems which compute various operations given input points. We group formulas to categories based on the EFD [58].

- **add** a simple addition formula, for $(P, Q) \in \mathcal{E}(\mathbb{F}_p)^2$ where $P \neq Q$ computes $P + Q$.
  - **Unified.** A unified addition formula also correctly computes the addition of a point to itself, i.e. a doubling.
  - **Complete.** A complete addition formula has no exceptions and correctly adds all pairs of points on the curve, including the point at infinity. A complete addition formulas is necessarily also a unified formula. A necessary condition for the existence of complete addition formulas is the representability of the point at infinity in the particular coordinate system.
- **dbl** a simple doubling formula, for $P \in \mathcal{E}(\mathbb{F}_p)$ computes $[2]P$.
- **tpl** a tripling formula, for $P \in \mathcal{E}(\mathbb{F}_p)$ computes $[3]P$.
- **ladd** a ladder formula, for $(P, Q, P - Q) \in \mathcal{E}(\mathbb{F}_p)^3$ where the difference $P - Q$ is known, it computes $P + Q$ and $[2]Q$ simultaneously. As its name suggest it is used in ladder multipliers, where it provides the regular ladder step executed for each bit.
- **dadd** a differential addition formula, for $(P, Q, P - Q) \in \mathcal{E}(\mathbb{F}_p)^3$ where the difference $P - Q$ is known, it computes $P + Q$. A differential addition formula together with a doubling formula can be used as a ladder step in a ladder multiplier instead of the ladder formula. A ladder step can also be constructed using the less efficient simple addition formula.
- **neg** a simple negation formula, for $P \in \mathcal{E}(\mathbb{F}_p)$ computes $-P$.
- **scl** a scaling formula which scales the projective point back into a form it would have if it was just transformed from affine coordinates. This involves computing the inverses of the projective Z coordinates, multiplying and setting $Z = 1$.

Every formula belongs to one coordinate system but not necessarily just one. Formulas might mix coordinate systems if they have multiple input points, such

as a (differential) addition or ladder formula. Certain formulas are also only applicable given some condition on the input points, such as requiring one of the points to be scaled.

In Table 1.2 we list the number of formulas of each type for all of the coordinate systems in the EFD [58]. The table illustrates the large number of choices a developer has when choosing the formulas to use.

Curve	Name	add	dbl	tpl	ladd	dadd	neg	scl
$\mathcal{E}_{SW}$	jacobian	12	6	3	0	0	1	1
	jacobian-0	12	8	3	0	0	1	1
	jacobian-3	12	10	4	0	0	1	1
	modified	4	3	0	0	0	1	0
	projective	7	5	0	0	0	1	1
	projective-1	7	4	0	0	0	1	1
	projective-3	7	6	0	0	0	1	1
	w12-0	2	1	0	0	0	1	1
	xyzz	3	2	0	0	0	1	1
	xyzz-3	3	4	0	0	0	1	1
	xz	0	5	0	12	10	0	0
$\mathcal{E}_M$	xz	0	4	0	4	4	0	1
$\mathcal{E}_E$	inverted	6	2	2	0	0	1	1
	projective	12	4	4	0	0	1	1
	yz	0	4	0	3	3	0	1
	yzsquared	0	2	0	4	2	0	1
$\mathcal{E}_{TE}$	extended	6	2	1	0	0	1	0
	extended-1	12	2	1	0	0	1	0
	inverted	3	2	0	0	0	1	0
	projective	3	2	1	0	0	1	0

Table 1.2: Formulas for the four selected curve models, from EFD [58] and numerous original sources.

1.3.5 Finite field arithmetic

All of the formulas from the previous section operate on elements of finite fields, in this work we focus on $\mathbb{F}_p$. However, the natively available integer arithmetic instructions use word sizes that are too small to handle the integers used in cryptography and do not offer any modular arithmetic. Thus elliptic curve cryptography developers need to implement efficient multi-precision finite field arithmetic on top of the integer arithmetic available. As we show, this involves several implementation choices.

We discuss the implementation details in the multiplication, modular reduction and modular inversion operations, as those present the largest amount of choices as well as implementation complexity.

Reduction

- **By division.** The simplest, but also the slowest modular reduction algorithm is to perform a division operation with remainder and then simply return the remainder.

- **Barrett.** The Barrett reduction algorithm uses a value precomputed from the modulus to form a good approximation to the division of the element by the modulus, which gives a speedup. It also utilizes binary shifts instead of divisions, which allows further speedup. The precomputation needs to be done once per modulus and thus use of Barrett reduction pays off after just few multiplications [44, 13].

- **Montgomery.** The Montgomery reduction algorithm is not a direct replacement of modular reduction. It requires both precomputation and somewhat costly transformation of input into Montgomery form ($x \mapsto xR \bmod p$ for some precomputed R). However, when multiple multiplications and reductions are performed with the values, such as in scalar multiplication, its speed quickly pays off [44, 12, 13].

- **Fast primes.** There exist several forms of primes that admit much faster modular reduction than any generic methods. The NIST curves were specifically selected to use these primes, for performance reasons. Take the prime $p_{192} = 2^{192} - 2^{64} - 1$, then an integer $0 \leq c < p_{192}^2$ can be reduced if given its representation in 64-bit words $(c_5, c_4, c_3, c_2, c_1, c_0)$ by computing the four integers $s_1 = (c_2, c_1, c_0), s_2 = (0, c_3, c_3), s_3 = (c_4, c_4, 0)$ and $s_4 = (c_5, c_5, c_5)$ via copies and shifts and then computing the result as $s_1 + s_2 + s_3 + s_4 \bmod p_{192}$ [13].

Multiplication

- **Classical.** Classical or schoolbook multiplication is a simple multi-precision multiplication algorithm, which executes the same multiplication algorithm that is taught in most schools, hence the name. It is also sometimes called Long Integer Multiplication (LIM) and has a variant called Comba multiplication which scans the product instead of the operands. The modular variant of this multiplication algorithm simply performs the

multiplication, obtains an integer $< p^2$ and then performs modular reduction [12, 13].

- **Karatsuba.** The Karatsuba multiplication algorithm is a recursive algorithm which splits the numbers to multiply recursively and uses the high speed of binary shifts and additions to perform multi-precision multiplication with asymptotically less single-precision multiplications than schoolbook multiplication. However, its speed might not be better than that of schoolbook multiplication for integer sizes used in elliptic curve cryptography [12, 13].

- **Montgomery.** The Montgomery multiplication algorithm is specific to modular multiplication. It interleaves the multiplication and Montgomery reduction operations to obtain a reduced result. Thus it requires the operands to be in Montgomery form, and the result remains in Montgomery form [12, 13].

Inversion

- **GCD.** The extended Euclidean algorithm for computing the greatest common divisor can be used to compute a modular inverse. When computing $gcd(a, p)$ for a prime p and $a \in \mathbb{F}_p^*$ we obviously get 1 as the result, however during this computation one of the intermediate values has the property $ax \equiv 1 \mod p$ which gives the inverse x [13].

- **Euler.** The modular inverse of an element $a \in \mathbb{F}_p^*$ can also be computed using Euler's theorem, which states that for coprime a and m, it holds that $a^{\phi(m)} \equiv 1 \mod m$ and thus $a^{\phi(m)-1} \equiv a^{-1} \mod m$. For prime $p = m$ this leads to the modular exponentiation of a by $p - 2$ [12].

2 Side-channel attacks

Implementations of cryptographic operations do not exist and run in the abstract way they were described in the previous chapter. They run on physical devices and have manifestations beyond the abstract ones. For example, a digital signature primitive does not just output the signature, it outputs the signature or an error or possibly raises an exception, it also accesses memory, has an effect on the processor caches and runs for a measurable amount of time, among other things. All of these and other manifestations constitute *side-channels* that potentially leak information about the execution of a cryptographic operation to an observing attacker.

The field of *side-channel attacks* (SCA) was able to use leakages in power consumption [1, 59], electromagnetic radiation [60, 61, 62, 63], caches and other micro-architectural structures [64, 65] or even photon emission [66] and sound [67] to extract sensitive information from cryptographic implementations. Side-channel attacks are not limited to passive observation of a cryptographic operation and its manifestations but can also actively influence it by injecting faults into it, for example by injecting short voltage glitches into the device's power supply or inducing a large change in the magnetic field around the device. These attacks fall into the category of *fault injection* and together with passive side-channel attacks give the attacker a powerful set of tools that is hard to defend against. We will, however, only consider passive side-channel attacks in the rest of this work as the active ones often require quite complicated hardware setup and do not directly help achieve the goals of this work.

Reasonable classification of a broad and complex topic like side-channel attacks on elliptic curve cryptography is quite hard, as the main goal of attacks is not to be easily classifiable into sensible categories but to be most efficient and effective. We thus adopt the classification and formalism of Bauer et al. [68], but only consider single-order attacks.

They consider the cryptographic operation to consist of a sequence of elementary calculations $(C_i)_i$ that read, perform computations on and update values in memory. The outputs of these computations are denoted by $O(s, X)$ where s is a secret sub-part and X a public variable that the output depends on. The leakage $\mathbf{L}$ is a multivariate random variable, where its i-th element represent the leakage at time t_i. Certain properties are often assumed about this leakage such as it having a multivariate normal distribution or that its value at index i has the form $\phi_i(O) + B_i$ where the ϕ_i depends only on the computation of O and B_i represents the noise, an independent normally distributed random variable with zero mean and standard deviation σ_i [68, 69].

2.1 Side-channels

In this section, we present the different side-channels that can be used as sources of information about the execution of a cryptographic or otherwise sensitive operation. They range from the purely abstract channels like errors raised by an implementation to physical channels like the power consumption of a device. In the rest of this work, we focus significantly on the power and electromagnetic radiation side-channels, while also considering errors and timing.

2.1.1 Timing

The duration of a cryptographic operation was first considered as an information source by Kocher [70] and was the first side-channel to be used to attack a cryptosystem implementation. Timing is a rough counter of instructions performed by the target device. It is affected by caches and noise unrelated to the cryptographic operation, and as such can usually be used to infer only limited information about the execution of the operation. The utility of a timing side-channel greatly depends on how precisely a particular operation or an interesting part of it can be timed, e.g. how precisely can the attacker detect the start and end of the operation. Timing as a side-channel can be utilized in a large variety of scenarios, it can be used by one device timing the execution on another device (e.g. a smartcard) [2], with both the attacker and victim on one device or even remotely across the network [71].

Making cryptosystem implementations constant-time, that is – eliminating secret dependent timing leakage – is nowadays a very common goal of both implementors and cryptosystem designers.

2.1.2 Errors

Errors, or in general the output of a cryptographic operation can form a side-channel as well. Assertions, exceptions and error values returned by programmers can often contain interesting information about program flow and signal whether certain conditions were hit during an operation. Thus they can be used as a side-channel if this program flow or condition is somehow sensitive. Errors during the validation of padding in RSA encryption form the basis of the famous Bleichenbacher's padding oracle attack [72] which keeps reappearing even in recent work [73]. As is the case of timing, this side-channel can be present with the attacker on the same device as the victim, on separate but local devices or on devices on a network.

2.1.3 Micro-architectural structures

Modern processors are extremely complicated devices with intricate mechanisms to improve performance, such as the use of caches, branch prediction, speculative execution or out-of-order execution. Some of these mechanisms can indeed form a side-channel, as their state changes with the execution of instructions and memory accesses. If this state can be queried or somehow leaked across a security boundary, a side-channel is present. Take the example of caches, which store recently accessed data and instructions from memory for quick lookup. If an attacker can query the state of the cache sets during the execution of a cryptographic operation, the attacker learns valuable information about which parts of memory were accessed and in what order [74].

Recently, there has been a significant amount of work on micro-architectural attacks that do not only leverage caches, but other micro-architectural structures to form side-channels, such as the Meltdown attack [64].

2.1.4 Power consumption

Cryptographic algorithms, while often very abstract in nature, are executed on very concrete and physical devices, containing some form of modern semiconductor circuitry. This circuitry communicates using electrical signals, computes on inputs to produce outputs and consumes power to do so. Apart from its intended inputs and outputs, it has other physical manifestations. One of those being its power consumption, which is not static, but varies with transistors switching and other activities inside of a chip [69].

Instantaneous power consumption of a device during computation usually depends on the instructions and data handled during the computation, as these affect the amount of circuitry switching in each clock cycle. Thus, power consumption forms a side-channel, one that is used very often in side-channel attacks and was at the very beginning of the field with the seminal work of Kocher et al. [1].

Compared to the timing, error or micro-architectural side-channels, power consumption is purely a local side-channel that is often performed using dedicated equipment to mount an attack against a micro-controller. However, there has recently been an example of an attack that performed power analysis on just one micro-controller, with both the attacker and victim running on it [75].

The usual power analysis measurement setup involves an oscilloscope and one or several probes. If voltage probes are used, the current needs to be measured by measuring the voltage drop across a shunt resistor inserted either on the power supply or ground lines of the device (a differential probe is required for power supply line setup). A current probe can also be used to measure the instantaneous

power consumption of a device [69]. The sampling frequencies of the oscilloscopes used during attacks range from kHz to GHz, depending on the requirements of a particular attack. A technique of clock-synchronous sampling (e.g. synchronizing the oscilloscope sampling with a particular point of the device's clock cycle) was demonstrated to reduce the required sampling rate while maintaining the success rates of attacks [76].

2.1.5 Electromagnetic radiation

Electromagnetic emanations are produced by a computing device due to the varying current flows inside of it. These emanations are thus related to the power consumption of the device as well as its circuitry and its geometry. Thus, similarly to power consumption, EM emanations can be used as side-channel and were considered as one in the early work of Agrawal et al. [77]. However, compared to the power consumption side-channel, EM radiation can form a side-channel into only a part of the circuitry of the device, as probes can be positioned to capture only emanations local to that part of circuitry [69]. This allows it to lower noise by targeting the part of circuitry where the sensitive computation takes place and where the leakage is maximized.

2.2 Simple SCA

In a simple side-channel attack, the attacker gains information about the secret s from a trace or traces that all processed the same public input x. The original definition of simple power analysis (SPA) and therefore also simple SCA given by Kocher et al. [1] is that the traces are directly interpreted to yield information about the device's operation and secrets. Simple side-channel attacks are also sometimes characterized as targeting the leakage via secret-dependent differences in code path but ignoring the leakage via secret-dependent values.

The general simple side-channel attack then goes as follows [68]:

1. Choose, fix or learn an input x.
2. Gather N measurements $(\mathbf{l}_j)_j \hookleftarrow (\mathbf{L}|X = x)$, where each measurement consists of M samples.
3. Enumerate hypotheses $\hat{s}$ on the secret sub-part s, and compute $\mathbf{h} = \mathbf{m}(O(\hat{s}, x))$, where $\mathbf{m}$ is a chosen leakage modelling function.
4. Use the distinguisher Δ to obtain a score for the hypothesis $\hat{s}$ as $\Delta[\hat{s}] = \Delta[(\mathbf{l}_j)_j, \mathbf{h}]$.
5. Infer information about s from the scores.

As an example, we will form a simple attack (that is similar to the simple SCA attack from [68]) on the double-and-add scalar multiplication algorithm which leaks the secret scalar.

Example 1. In the following attack, we have the assumption that the double-and-add scalar multiplication algorithm (Algorithm 1) is used with different (not unified) formulas for the double and add operations. We further assume the attacker possesses a trace of the execution of the doubling and addition $\mathbf{p}_1$ and only the addition $\mathbf{p}_0$ on the target device. The attack works from the most significant bit to the least significant, tracking the progress of the scalar multiplier. The attacker chooses the Pearson correlation as a distinguisher and given a key hypothesis $\hat{s}$ on the currently unknown $i-th$ most-significant bit computes

$$\Delta[\hat{s}] = \Delta\left[\frac{1}{N}\sum_j \mathbf{l}_j(t_i,\ldots,t_{i+1}), \mathbf{p}_{\hat{s}}\right] = \rho\left(\frac{1}{N}\sum_j \mathbf{l}_j(t_i,\ldots,t_{i+1}), \mathbf{p}_{\hat{s}}\right)$$

, where the times t_i correspond to the start of the i-th iteration of the loop. The hypothesis $\hat{s}$ that leads to the highest correlation is taken.

In simpler words, the attacker averages the traces to reduce noise and then compares the leakage of a single iteration of scalar multiplication to two patterns, corresponding to situations when $k_i = 0$ and $k_i = 1$, and thus learns the value of the key bit k_i by taking the hypothesis associated with the pattern that better matches.

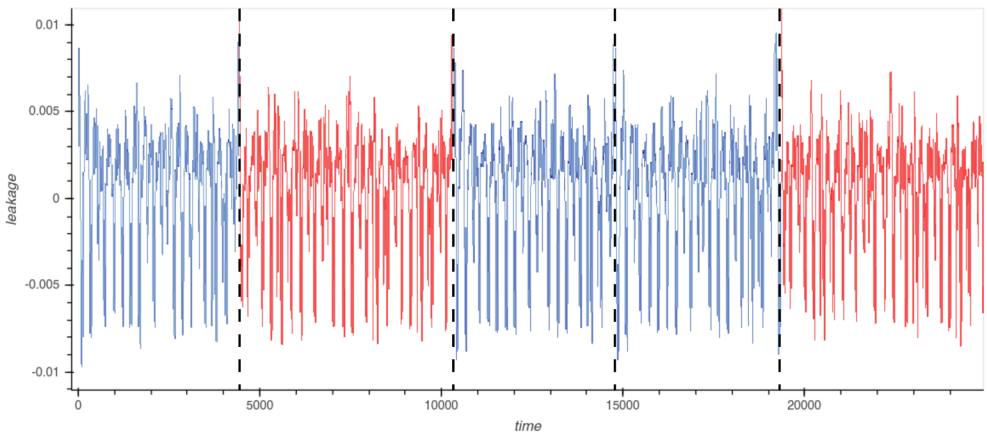


Figure 2.1: Power trace showing three loops of the double-and-add scalar multiplication algorithm operating on a scalar 101.... Note the two consecutive doubles in the middle. (addition, doubling)

The above example attack could also be performed by direct observation of the traces, which fits the simple SCA description by Kocher et al. [1]. Figure 2.1 displays a trace of the scalar multiplication where the pattern of doublings and additions is visible.

2.3 Advanced SCA

Advanced, or sometimes called differential, side-channel attacks utilize traces collected using many, known or chosen, public inputs x_j. Contrary to simple side-channel attacks, which were described as analyzing the secret-dependent differences in code path, the advanced side-channel attacks are much more sensitive and analyze even the secret-dependent differences in values that are being manipulated. For their description we again use the framework of Bauer et al. [68]:

1. Gather N measurements with their inputs $(\mathbf{l}_j, x_j)_j \hookleftarrow (\mathbf{L}|X)$, where each measurement consists of M samples.
2. Enumerate hypotheses $\hat{s}$ on the secret sub-part s, and for each compute predictions $\mathbf{h}$ where $\mathbf{h}_j = \mathbf{m}(O(\hat{s}, x_j))$, where $\mathbf{m}$ is a chosen leakage modelling function.
3. Use the distinguisher Δ to obtain a score for the hypothesis $\hat{s}$ as $\Delta[\hat{s}] = \Delta[(\mathbf{l}_j)_j, (\mathbf{h}_j)_j]$.
4. Infer information about s from the scores.

The distinguishers used in advanced SCA include among others the difference of means [1], correlation [78], mutual information [79] or the maximum likelihood test [80].

The leakage functions that are used are often based on Hamming weight or Hamming distance as the power side-channel leakage of CMOS-based electronics was found to fit this assumption well [69].

2.3.1 Vertical side-channel attacks

In a vertical side-channel attack the N measurements $(\mathbf{l}_j)_j$ are the N collected traces [68]. If one imagines the traces as stacked on top of one another, the name hints at the analysis being performed vertically, analyzing the same part of every trace to extract information on the computation. As advanced side-channel attacks are often performed on random data inputs, this corresponds to analyzing the same operation performed on different inputs in the different traces.

Classical DPA

In the classical *Differential Power Analysis* (DPA) attack, which was the first advanced side-channel attack introduced, the leakage modelling function $\mathbf{m}$ takes the r-th bit from the intermediate value, for some pre-selected r smaller than the bit-size of the intermediate value. Thus $h_j = \mathbf{m}(O(\hat{s}, x_j)) = O(\hat{s}, x_j) \ \& \ 2^r$.

The distinguisher introduced by Kocher et al. [1] on input of $(\mathbf{l}_j)_j, (h_j)_j$ splits the traces into two sets, based on their associated bit value h_j into sets T_0 and T_1. It then computes two mean traces for the sets T_0 and T_i and computes their sample-wise difference, which is why it is often called a difference of means (DoM) distinguisher.

$$\Delta\left[(\mathbf{l}_j)_j, (h_j)_j\right](i) = \frac{\sum_{\mathbf{l} \in T_1} \mathbf{l}(i)}{|T_1|} - \frac{\sum_{\mathbf{l} \in T_0} \mathbf{l}(i)}{|T_0|}$$

The differential trace is however still a trace and the distinguisher should generally output just a scalar score, thus the maximum over the samples of the differential trace is taken:

$$\Delta[\hat{s}] = \Delta\left[(\mathbf{l}_j)_j, (h_j)_j\right] = \max_{i \in [0,M]} \{|\Delta\left[(\mathbf{l}_j)_j, (h_j)_j\right](i)|\}$$

The intuition behind the distinguisher is that if the hypothesis $\hat{s}$ is correct, the sets T_0 and T_1 contain traces that have the selected intermediate value bit equal to 0 and 1, respectively. Thus, under the assumption that this bit is somehow correlated to the leakage, the difference of means should be large in samples where the bit is processed and where it leaks. Conversely, if the hypothesis is incorrect, the sets T_0 and T_1 contain mostly randomly sampled traces, and thus their difference of means should be small [1, 69].

CPA

Correlation Power Analysis (CPA) uses a distinguisher based on the Pearson's correlation coefficient and allows for the use of a wide range of leakage modelling functions apart from the most used Hamming weight and Hamming distance. It can be seen as an extension of the above classical DPA attack which took a single bit Hamming weight leakage model and a simpler distinguisher [78].

Mutual Information Analysis

The idea of partitioning leakage traces based on a key hypothesis and then measuring the statistical properties of the partitions (as used in classical DPA) is also used in the *Mutual Information Analysis* (MIA) attack. MIA introduces a

distinguisher based on the mutual information between the estimated distributions of the modelled leakages and the observed leakages. A key hypothesis that leads to the highest overall mutual information is then likely correct, as it best describes the observed leakages. Mutual information analysis is a very general distinguisher that does not introduce additional assumptions about the leakage and allows the use of almost any leakage modeling function [79]. A comprehensive study of mutual information analysis is available by Batina et al. [81].

RPA/ZPA/EPA

Goubin's *Refined Power Analysis* (RPA) attack on elliptic curve cryptography takes an approach different to aforementioned DPA-like attacks. It is specific to elliptic curve cryptography and only applicable to key exchange primitives like ECDH or X25519 as it requires the attacker to be able to select the point that is multiplied by the static secret that the attacker is trying to recover. The attack requires the existence of "special" points on the curve, i.e. a point $P_0 \neq \mathcal{O}$ that has one of the coordinates equal to zero. It is assumed that operation with such a point, whether it is addition or doubling will produce, over many traces, a mean trace that is discernible from a mean trace obtained using random non-special points.

The strategy of the attack is to use a key hypothesis to compute a scalar c such that the point $[c]P_1 = P_0$ is computed by the scalar multiplier if and only if the key hypothesis is correct. The point $P_1 = [c^{-1} \mod n]P_0$ is given to the scalar multiplier as input, i.e. it is used as a public key during key exchange. The attacker is then able to confirm the key hypothesis if the mean trace from measurements with P_1 as input is different to a mean trace using random points. The attack proceeds iteratively in the same direction as the scalar multiplier, forming key hypotheses in the bits of the secret used in one iteration of the algorithm [82].

The assumption that a point with a zero coordinate would be detectable on a side-channel trace is supported if one analyzes the execution of finite field multiplication algorithms with zero input. These algorithms execute many single-word integer multiplications with zero input, which are likely to leak in one way or another [82]. As multiplications are the likely source of leakage, it is possible to execute the attack even if some intermediate value computed in point addition or doubling is zero, which is the basis of the *Zero-value Point Attack* (ZPA) by Akishita; Takagi [83].

The general idea of creating points that have a detectable side-channel leakage if some key hypothesis is correct is also used in the *Exceptional Procedure Attack* by Izu; Takagi [84]. It uses an error side-channel to detect when the implemen-

tation attempts an impossible finite field inversion 0^{-1}, which the attacker can force under a key hypothesis using a chosen point. It is, however, able to recover only a few bits of the scalar, as the construction of the chosen points requires the computation of division polynomials, which is computationally prohibitive for large degrees.

Template attacks

Template attacks are a subset of profiled attacks, which assume the attacker has a copy of the target device under their control, and can create templates that capture the leakage of the device given the different sub-secrets. In the framework used, this amounts to the leakage model being the estimated PDF $f_{\hat{s},x_j}(\cdot)$ and the distinguisher being a maximum likelihood test $\prod_j f_{\hat{s},x_j}(\mathbf{l}_j)$ [80, 85, 86].

Leakage assessment

In security evaluations, such as those performed during the certification process of a hardware security element or a smartcard, it is often unnecessary to execute full attacks such as DPA or CPA, as the goal is not to leak some secret key but to evaluate whether the implementation is secure or whether it leaks. This is the goal of *leakage assessment*, of which test-vector leakage assessment (TVLA) is the most popular method. Test vector leakage assessment works similarly to a DPA attack, however, it assumes the secret is known and uses a different statistical test, usually Welch's t-test for the difference of means. It aims to use this test to determine whether two groups of traces with different secret inputs are statistically different, i.e. there is leakage [87].

The two categories of TVLA tests are:

- **Specific.** A specific test uses random inputs and partitions the traces into two groups based on a chosen intermediate value, much like a DPA attack.
- **Non-specific.** A non-specific test uses either fixed and random or fixed and different fixed inputs and partitions the traces into two groups based on which input group the trace is associated with.

2.3.2 Horizontal side-channel attacks

Horizontal side-channel attacks form the N measurements $(\mathbf{l}_j)_j$ from the traces horizontally, i.e. a limited number of traces is used that is split into blocks forming the measurements $(\mathbf{l}_j)_j$. The attacks then analyze relations and collisions between these measurements and the intermediate values computed in them.

Doubling attack

The *doubling attack* of Fouque; Valette [88] is a chosen message attack, applicable to key exchange primitives like ECDH using the double-and-add-always scalar multiplication algorithm. The points P and $[2]P$, for an arbitrary point P on the curve, are given to the implementation, producing the traces $\mathbf{l}_P$ and $\mathbf{l}_{[2]P}$ respectively. The attack works based on the fact that the inputs to the doubling operation in iteration i and $i+1$ of the scalar multiplications in $\mathbf{l}_{[2]P}$ and $\mathbf{l}_P$ are the same if and only if the i-th bit of the scalar is zero. Attacker recovers the whole secret scalar from the comparison of two traces.

Yen et al. [89] extend the doubling attack to apply to the Montgomery ladder scalar multiplication algorithm by observing that a collision in the i-th and $i+1$-th iterations of scalar multiplication of $[2]P$ and P happens if and only if the i-th and $i+1$-th bits of the scalar are identical.

Collision attack

The horizontal *collision attack* of Bauer et al. [68] against unified and complete addition formulas detects cases where the inputs to an addition formula are equal by using collisions of inputs in multiplications in the formula, which only happen if the input points are equal. When these applications of the addition formula can be detected, and a vulnerable scalar multiplier is used, the scalar can be reconstructed.

2.4 Countermeasures

The amount of countermeasures protecting elliptic curve cryptography implementations from side-channel attacks rivals that of the attacks themselves. The use of regular scalar multipliers which have no dependency of the code path taken on the secret values is commonplace. Complete addition formulas, which enable easy implementation of regular scalar multipliers are also used extensively. In this section, we discuss some countermeasures based on randomization (sometimes referred to as blinding or masking in secret-key cryptography). A more thorough analysis of known countermeasures, their impact and cost can be found in [90, 91]. In this section we list randomization countermeasures specifically.

2.4.1 Scalar randomization

Randomizing the scalar is an important countermeasure, as the scalar is *the* secret, and its processing in scalar multiplication is critical. Figure 2.2 shows the several approaches which rely on some identity modulo the order of the curve.

Group scalar randomization

function $\textsc{Mult}(G, k)$
  $r \xleftarrow{\$} \{0, 1, \ldots, 2^{32}\}$
  return $[k + rn]G$

Additive splitting

function $\textsc{Mult}(G, k)$
  $r \xleftarrow{\$} \mathbb{Z}_n^*$
  return $[k - r]G + [r]G$

Euclidean splitting

function $\textsc{Mult}(G, k)$
  $r \xleftarrow{\$} \{0, 1, \ldots, 2^{\lfloor \log_2(n)/2 \rfloor}\}$
  $S \leftarrow [r]G$
  $k_1 \leftarrow k \mod r$
  $k_2 \leftarrow \lfloor \frac{k}{r} \rfloor$
  return $[k_1]G + [k_2]S$

Multiplicative splitting

function $\textsc{Mult}(G, k)$
  $r \xleftarrow{\$} \{0, 1, \ldots, 2^{32}\}$
  $S \leftarrow [r]G$
  return $[kr^{-1} \mod n]S$

Figure 2.2: An overview of scalar randomization countermeasures [91].

2.4.2 Point randomization

Point randomization via projective coordinates is possible if a coordinate system is used where a point can be represented by a class of concrete coordinate values. Point randomization effectively randomizes the intermediate values encountered during scalar multiplication of said point [92].

function $\textsc{Mult}(G = (X : Y : Z), k)$
  $r \xleftarrow{\$} \mathbb{F}_p^*$
  $S \leftarrow (rX : rY : rZ)$
  return $[k]S$

2.4.3 Curve randomization

Randomization of the whole curve and thus of all computations on it is possible by picking a random curve isomorphism, transforming the point using this isomorphism, performing the computation and then transforming it back [93].

function $\textsc{Mult}(G, k)$
  $u \xleftarrow{\$} \mathbb{F}_p^*$ // u defines an isomorphism $\psi_u : (x, y) \mapsto (u^{-2}x, u^{-3}y)$
  $S \leftarrow \psi_u(G)$
  return $\psi_u^{-1}([k]S)$

2.5 Reverse-engineering

The application of side-channel analysis to reverse engineering, i.e. to recover information about the implementation and not any particular key, was first demonstrated by Quisquater; Samyde [94]. They were able to reverse-engineer the sequence of executed instructions on a smartcard processor, using electromagnetic radiation as a side-channel with a classifier based on correlation and a simple neural network. Disassembly via side-channels was later executed against several simple microprocessors (PIC16, ATMega 163 and ARM Cortex-M0) [95, 96, 97, 98].

Vermoen et al. [99] demonstrated disassembly of Java bytecode on several smartcards of the JavaCard platform, which is a very popular platform of programmable smart cards. JavaCards might or might not execute the Java bytecode natively and thus present different challenges than traditional embedded microprocessors. They or the microprocessors they are based on are also often used as secure elements, which means they have more side-channel countermeasures than their general-purpose counterparts.

Reverse-engineering of a cryptographic primitive from side-channels was first presented by Clavier [100], which showed how the substitution tables of the GSM A3/A8 ciphers, the design of which was secret at the time, could be recovered. The ideas of SCARE (Side-Channel Analysis for Reverse Engineering) are further developed by Daudigny et al. [101] which attacks a software implementation of DES to leak constants and implementation choices made in the algorithm.

Public-key cryptography does not lend itself to secret designs of ciphers in a way that secret key cryptography does, as there is a limited number of problems to base the design of primitives on. However, implementation details of concrete implementations of public-key cryptography are often secret and thus warrant reverse-engineering. Correlation power analysis was successfully used to reverse-engineer the details of a Montgomery multiplier in an implementation of the RSA cryptosystem [102], but its application to reverse-engineering of elliptic curve cryptography was only briefly mentioned.

As we have shown in Section 1.3, the space of possible implementations of elliptic-curve cryptography is large. The attacks mentioned in Sections 2.2 and 2.3 all require that the attacker has precise knowledge of the target implementation. These two facts show that reverse-engineering of elliptic-curve cryptography implementations is an overlooked problem.

3 PYECSCA: Python Elliptic Curve Side Channel Analysis toolkit

The **pyecsca** toolkit[1] aims to be a comprehensive toolkit for side-channel analysis of elliptic curve cryptography implementations.

In this chapter, we first present the architecture and functionality of the toolkit, then perform source code analysis of open-source cryptographic software libraries to gain information about their ECC implementations. Finally we describe several new methods for reverse-engineering elliptic curve cryptography implementations via side-channels and evaluate a subset of these methods on several targets.

3.1 Architecture and functionality

To implement **pyecsca** we chose the Python programming language, as is evident from the name. We did so because of the large basis of libraries and tools for scientific computing available in Python, such as numpy [103] or scipy [104], both of which we use. The toolkit consists of three parts, the `core`, `codegen` and `notebook` parts (see Figure 3.1 for an overview).

The `core` part of the toolkit provides the major part of the functionality, consisting of the `ec` and `sca` packages. The `ec` package focuses on elliptic curve functionality. It uses data from the Explicit-Formulas Database (EFD) [58] to provide a dynamic way of computing with elliptic curves, utilizing any curve model, coordinate system and formulas supported by the EFD. It provides classes to work with elliptic curves, points on them, finite field elements as well as classes for meta-objects like a curve model, coordinate system or an addition formula. The `sca` package focuses on side-channel attacks, providing functionality to communicate with targets, record traces from oscilloscopes, process and analyze them.

The `codegen` part of the toolkit implements functionality for automatically generating C implementations of elliptic curve cryptography for micro-controllers on top of the core part.

The main way of working with the toolkit is intended to be via Jupyter notebooks [105], which are web-application based interactive notebooks, with embedded code and documentation. The `notebook` part of the toolkit provides a base set of notebooks to work with. It is the part where specific workflows are created using the other parts of the toolkit, which in the end, is just a toolkit.

1. Pronounced [pɪɛtska], named to sound like *piecka* in Slovak which is a diminutive word for stove, hence the logo .

In this section, we discuss the generic functionality of the toolkit, which underpins its reverse-engineering abilities.

The toolkit is statically typed, has a code coverage of 94%, extensive documentation and is released under an open-source (MIT) license.

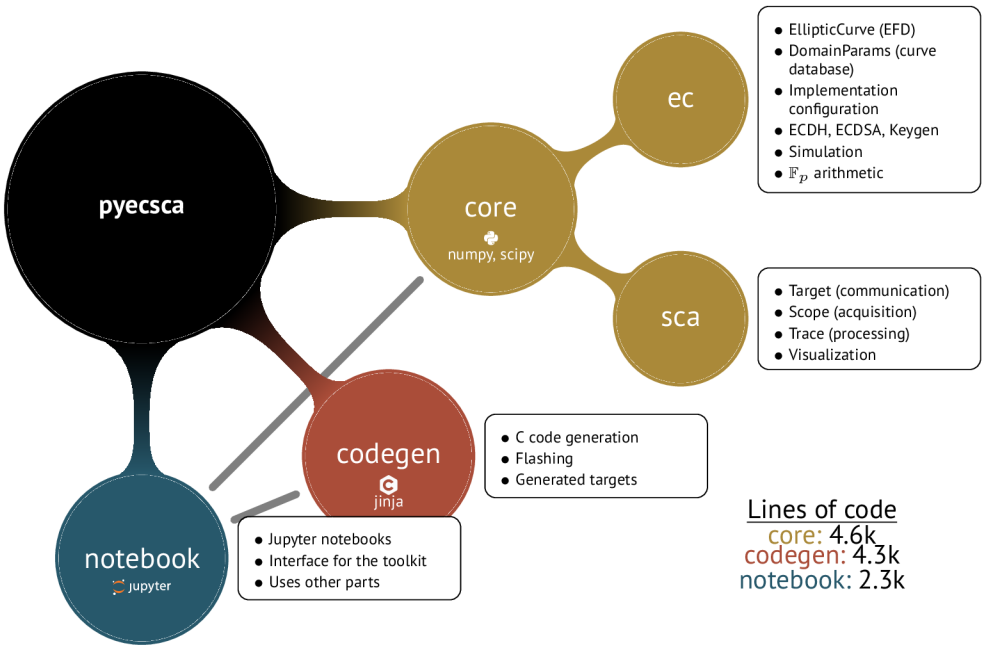


Figure 3.1: Overview of the **pyecsca** toolkit architecture.

3.1.1 Implementation configuration

An ECC implementation configuration is a central concept in **pyecsca** as it represents the implementation details that the toolkit aims to reverse-engineer (see Section 1.3 for an overview). We use data from the Explicit-Formulas Database, which is a database of curve models, coordinate systems and formulas curated by Lange; Bernstein [58]. Although the EFD is a website, it also contains data in raw text-files with undocumented but stable format (see the Appendix for a sample of the data). We use data from the EFD to build the first three components of an implementation configuration: the curve model, the coordinate system and the formulas. The remaining seven components – the scalar multiplier, hash algorithm, random sampling method and big number multiplication, squaring and modular reduction and inversion – are implemented manually. The core part of the toolkit, however, does not implement the different random sampling methods and big number arithmetic, those are only available in the generated C imple-

mentations for micro-controllers. This is because we place the level of detail in the toolkit just above the level of finite field arithmetic and do not simulate these different options. We do implement these in the generated implementations, as these options obviously affect the side-channel leakage of the implementation.

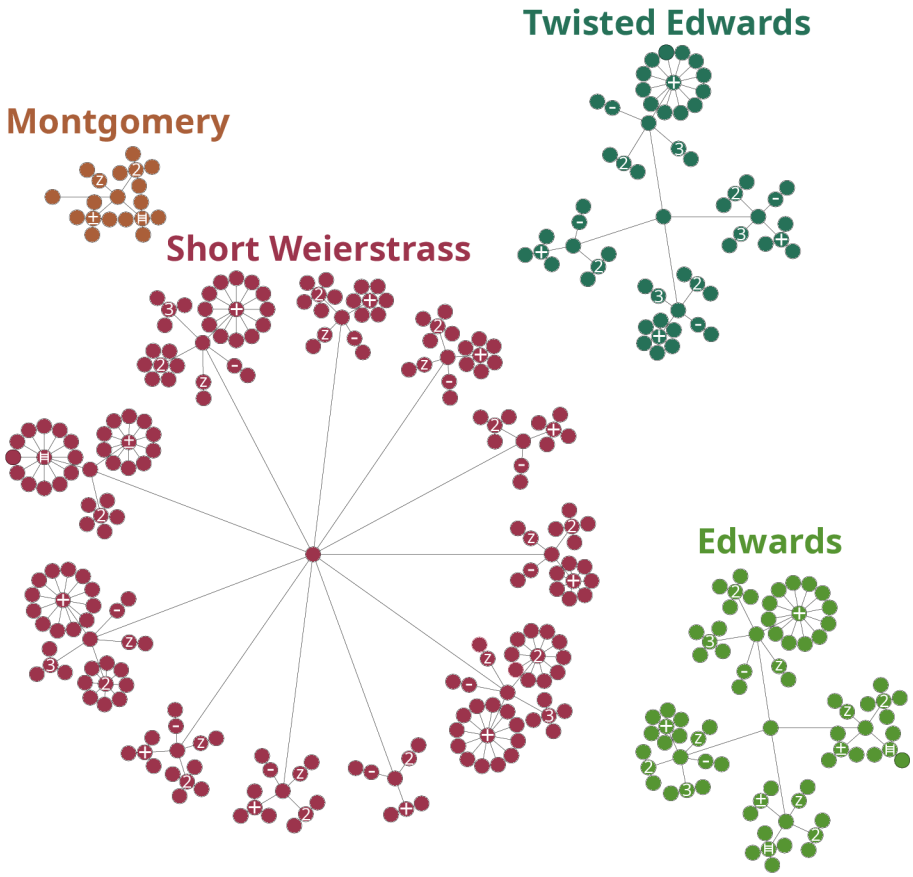


Figure 3.2: Graph of the four selected curve models from EFD, their coordinate systems and formulas using them. The root of each tree is a curve model, nodes with depth one are coordinate systems, nodes below them represent types of formulas (➕ Addition, 2️⃣ Doubling, 3️⃣ Tripling, ➖ Negation, Z Scaling, ➗ Differential addition, 🇱 Ladder) and the leaves represent individual formulas.

The implementation configuration, as defined by the toolkit thus has nine components, described below:

- **Curve model**. The curve model used by the implementation. One of *Short-Weierstrass*, *Montgomery*, *Edwards*, *Twisted-Edwards*. Note that some of these models have restrictions in which classes of curves are representable in them (see Subsection 1.3.1). These four models are only the most popular selection of those available from the EFD.
- **Coordinate system**. The coordinate system on the curve model used by the implementation, different models have different coordinate systems (see Subsection 1.3.3).
- **Scalar multiplier + Formulas**. The scalar multiplication algorithm, used in Key generation, ECDH and ECDSA, along with the formulas used in it (see Subsections 1.3.2 and 1.3.4). Currently, there are eight multipliers implemented: *LTR*, *RTL*, *Coron*, *Ladder*, *SimpleLadder*, *DifferentialLadder*, *BinaryNAF* and *WindowNAF*. These are hand implemented in Python, as EFD only provides formulas. The different scalar multipliers require different types of formulas, for example the *Ladder* multiplier requires the **ladd** formula, while the *DifferentialLadder* multiplier requires the **dadd** and **dbl** formulas. The multipliers are also parameterized, all of them accept a boolean parameter `complete` which specifies whether they should start the scalar multiplication loop at a fixed bound (e.g. whether the formulas passed to them are complete). The *WindowNAF* multiplier is also parameterized by the width of the window, which can be any integer.
- **Hash algorithm**. The hash algorithm used to compute the shared secret in ECDH and to hash the messages in ECDSA. One of *None* (identity function on the data), *SHA1*, *SHA224*, *SHA256*, *SHA384*, *SHA512*.
- **Random sampling**. The random sampling technique used to generate uniformly random numbers modulo n. Can be *Sample* or *Reduce*. In the *Sample* case numbers up to $\lceil \log_2(n) \rceil$ bits are sampled uniformly until one is less than n. In the *Reduce* case a number up to $\lceil \log_2(n) \rceil + 40$ bits is sampled uniformly and reduced modulo n.
- **Big number multiplication**. The big number multiplication algorithm. One of *Base*, *Comba*, *Karatsuba*, *Toom-Cook*.
- **Big number squaring**. The big number squaring algorithm. One of *Base*, *Comba*, *Karatsuba*, *Toom-Cook*.
- **Modular reduction**. The modular reduction algorithm. One of *Base* (remainder after schoolbook division), *Barrett*, *Montgomery*.
- **Modular inversion**. The modular inversion algorithm. One of *GCD* or *Euler*.

Out of the ten components of the configuration, four clearly have strong dependencies, the curve model, coordinate system, formulas and scalar multiplier, while the other six can vary largely independently. It is, however, unlikely for an implementation to use one algorithm for big number multiplication and a different one for big number squaring, due to the additional complexity. Similarly, we assume that the implementation uses the same hash algorithm in ECDH and ECDSA, i.e. we always think of an implementation as only implementing one variant of the algorithms, while for example, a smartcard might provide variants with many hash algorithms. While formulas used in scalar multiplier are also independent, provided that they are from the same coordinate system, curve model and fit the scalar multiplier, they are usually published in papers together, as in a doubling and an addition formula. Thus, it is likely that an implementation will use formulas from one publication.

We performed an analysis of the space of configurations, given the above definition of an implementation configuration. We found that there are more than 30 million configurations in total. The configurations are not distributed across the four considered curve models uniformly, much more configurations use the *Short-Weierstrass* model, which is due to the EFD having many more coordinate systems and formulas for that model. For the purposes of this analysis, we enumerated all possible parametrizations of the scalar multipliers, but had to limit the domain of the integer *width* parameter of the *WindowNAF* multiplier to $\{3, 5\}$. For detailed breakdowns of the number of configurations for the different curve models see Table 3.1b, for the different coordinate systems see Table 3.1a and for the different scalar multipliers see Table 3.1c.

3.1.2 Domain parameters

Elliptic curves used as public domain parameters in cryptosystems are very often standardized, as there are several pitfalls in creating an elliptic curve that is safe for use in a cryptosystem, as well as for interoperability and performance reasons. There are certain optimizations that can be made when using particular curves, and some curves are only representable in some models. Thus there has been work on creating new fast curves [34, 48, 38]. There is thus a significant amount of standard curves scattered across standards (often not freely accessible), publications and source code.

We have collected standard curves into an extensive database (`std` ■ [106]), which also has a web front-end based on the GatsbyJS static page generator [107]. The database currently contains curves from more than ten sources and contains more than a hundred curves. The curves and metadata are stored in JSON, for interoperability. The web interface provides the JSON files for download as well

Curve	Coords	All	/opts
$\mathcal{E}_{SW}$	jacobian	4644864	4032
	jacobian-0	6193152	5376
	jacobian-3	7741440	6720
	modified	387072	336
	projective	2257920	1960
	projective-1	1806336	1568
	projective-3	2709504	2352
	w12-0	129024	112
	xyzz	387072	336
	xyzz-3	774144	672
	xz	534528	464
$\mathcal{E}_M$	xz	313344	272
$\mathcal{E}_E$	inverted	774144	672
	projective	3096576	2688
	yz	235008	204
	yzsquared	129024	112
$\mathcal{E}_{TE}$	extended	387072	336
	extended-1	774144	672
	inverted	193536	168
	projective	193536	168

(a) Number of configurations with a particular coordinate system.

Curve	All	/opts
$\mathcal{E}_{SW}$	27565056	23928
$\mathcal{E}_M$	313344	272
$\mathcal{E}_E$	4234752	3676
$\mathcal{E}_{TE}$	1548288	1344
Total	33661440	29220

(b) Number of configurations with a particular curve model.

Scalarmult	All	/opts
LTR	9271296	8048
RTL	4635648	4024
Coron	2317824	2012
Ladder	686592	596
SimpleLadder	4635648	4024
DiffLadder	525312	456
BinaryNAF	2317824	2012
WindowNAF	9271296	8048

(c) Number of configurations using the different scalar multipliers.

Table 3.1: Number of configurations by their components. **/opts** denotes the number without the independent options like hash-algorithm, random sampling or finite field arithmetic.

as generated code which constructs the curve in the SageMath [108] computer algebra language.

The database is part of the toolkit, and as such can be queried from Python to lookup a curve by name, which is constructed and returned in the coordinate system requested.

3.1.3 Simulation

While enumerating the possible implementation configurations shows the size of the configuration space, there is a much larger potential for the use of the EFD data. Due to the way we built the configuration components, we can actually run computations using the curve model, coordinate system, formulas and the rest of the configuration. Curve model files (see Listing 4.1 for an example) are parsed

to create model classes, which then parse the coordinate system and formula files (see Listings 4.2 and 4.3) recursively. Formulas from the EFD are parsed to create executable Python code, as their three opcode variant (see Listing 4.4) is valid Python syntax.

Apart from simply computing with the configurations, we built a strong tracing system, which is able to capture all of the operations performed during execution, in-order, into an execution tree, down to the level of individual finite field operations. We call these recorded events *actions*, each action can have children and additional metadata. Children form an ordered sequence of sub-actions that were executed during the parent action. One can imagine that the ECDH action has two children: scalar multiplication action by the private key and a hashing action (if the variant of ECDH hashes the point coordinate to produce the shared secret). The metadata in the action records important values associated with the action, usually, the inputs and output of the action. The toolkit uses the following types of actions:

- `KeygenAction`. An action performed during key generation, its metadata consists of the domain parameters over which a keypair was generated.
- `ECDSAAction` (abstract). An abstract action, encompassing two ECDSA actions, its metadata contains the domain parameters used, the hash algorithm and the message.
  - `ECDSASignAction`. Creation of an ECDSA signature, contains the private key.
  - `ECDSAVerifyAction`. Verification of an ECDSA signature, contains the signature and public key.
- `ECDHAction`. An ECDH key agreement execution, its metadata consists of the domain parameters used, hash algorithm, public and private keys.
- `RandomModAction`. An action performed during random sampling (of a private key or random nonce) modulo some number, contains the modulus.
- `ScalarMultiplicationAction`. A multiplication of a point by a scalar, its metadata contains the multiplied point and scalar.
- `CoordinateMappingAction`. An action performed during mapping of a point from one coordinate system to another, its metadata consists of the two coordinate systems as well as the point being mapped.
- `FormulaAction`. Execution of a formula. Its metadata is the most extensive out of the actions, it contains the input point(s), the output point(s),

the formula itself, any values of domain parameters passed into it as well as all intermediate values computed.

The recording of the execution is done by the *context*. There is always one context that is recording the current execution, it is stored as a thread-local variable, such that new actions can be recorded and past actions queried.

There are two context classes implemented, `NullContext` which simply ignores all actions reported to it, and `DefaultContext` which records the actions into an execution tree . The `NullContext` is used when tracing of actions is undesirable, as computations are run for other reasons and performance is important.

The execution tree which is created by the `DefaultContext` is useful in many attacks which use intermediate values in some way. For each trace, which has some associated public inputs, and some key hypothesis, the execution can be simulated, resulting in an execution tree. This tree can then be easily queried for intermediate values, such as "take the second intermediate value computed in the fifth doubling in scalar multiplication".

3.1.4 Code-generation

The **pyecsca** toolkit is capable of fully automatic generation of C implementations of ECC given any implementation configuration, for a set of supported micro-controllers and CPU architectures. The target implementation supports ECDH, ECDSA and key generation accessible through a simple serial interface. We currently target the `STM32F0` and `STM32F3` chips, which are based on ARM Cortex-M0 and ARM Cortex-M4F, respectively. We also target the host device, such that implementations can be generated, built and run on the host device for testing and development purposes.

The ability to generate ECC implementations for micro-controllers is very useful for a multitude of reasons. These implementations can serve as evaluation targets of newly developed side-channel attacks so that researchers do not need to implement their own micro-controller implementations from scratch or cross-compile a (modified) library implementation to a micro-controller. Attacking custom-developed implementations, with potentially custom countermeasures is a common occurrence in SCA research. Using the toolkit, one only needs to modify the sources to add countermeasures or other custom features to the code and can then quickly build, flash and trace implementations with many different configurations.

We use the generated implementations as a target to test the reverse-engineering abilities of the toolkit, i.e., to have a non-black-box implementation that is given to the toolkit as a black-box implementation to reverse-engineer.

The whole workflow to create an implementation has four steps, each fully automated: generation of sources, building of the application, flashing and execution. Figure 3.3 displays an overview of the workflow.

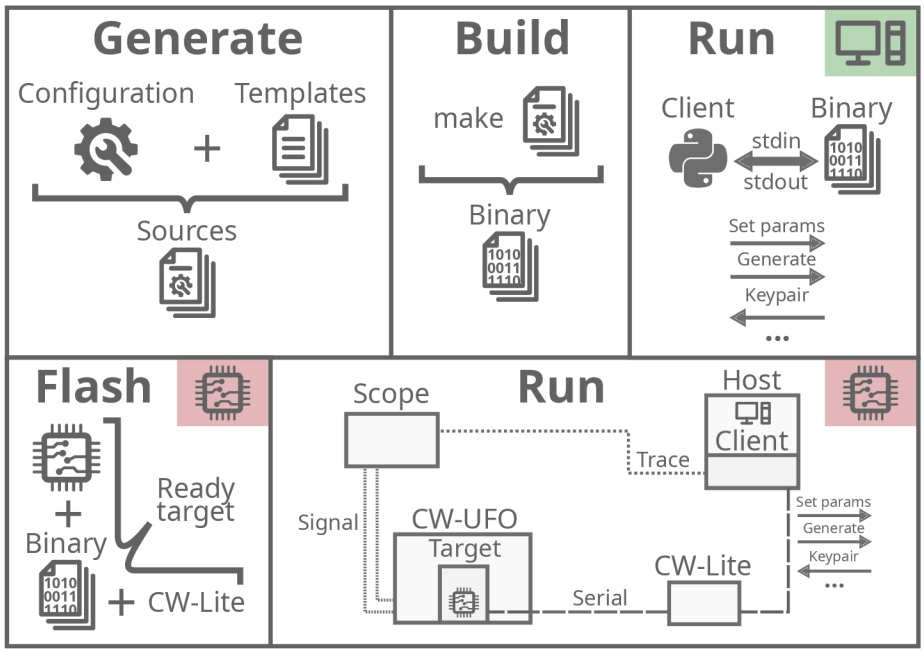


Figure 3.3: Visualization of the whole implementation generation workflow. 1) Generate 2) Build 3) Flash (embedded only) 4) Run

Generating

We use the Jinja2 [109] templating language, to automatically generate implementation C sources from template files (Jinja templates, not C++ templates) and the implementation configuration. The source generation technique we use is similar to server-side rendering of web pages in web application development, where HTML templates written in a template language are filled with user data and rendered. In our case, the C templates are filled with implementation configuration data. See Listing 3.1 for an example of a template and Listing 3.2 for an example of rendered source code from the template.

During the development of the toolkit, the `ECCKiila` tool [110] was released independently by a team at Tampere university which also uses data from the EFD to generate ECC implementations. This shows that our approach is viable.

The implementation configuration contains enough information to generate the full implementation on top of some code which is common to all implemen-

```
void point_set(const point_t *from, point_t *out) {
    {% for variable in variables %}
    bn_copy(&from->{{ variable }}, &out->{{ variable }});
    {% endfor %}
    out->infinity = from->infinity;
}
```

Listing 3.1: Example of a Jinja2 template in C.

```
void point_set(const point_t *from, point_t *out) {
    bn_copy(&from->X, &out->X);
    bn_copy(&from->Y, &out->Y);
    bn_copy(&from->Z, &out->Z);
    out->infinity = from->infinity;
}
```

Listing 3.2: Example of a generated function from a Jinja2 template in Listing 3.1, using a configuration with the projective coordinate system (X, Y, Z).

tations. The common code consists of all of the configuration agnostic parts of the implementation, the ECDH, ECDSA and key generation functionality, input/output functionality, PRNG, ASN.1 and HAL (Hardware Abstraction Layer) code. The full configuration in **pyecsca** has the following parts:

- **Curve model**. The curve model affects the domain parameter structure used in the implementation. Thus it is only used in the domain parameter input functionality.
- **Coordinate system**. The coordinate system affects the number of coordinates as well as formulas for converting points to and from the affine coordinate system, which is the input/output format.
- **Formulas**. Rendering the formulas correctly is the most involved out of the whole configuration, as one needs to generate C code from a list of Python statements with well-defined inputs and outputs. Luckily, one does not need to implement a whole transpiler or any optimizations here, just rewrite the statements in C. We parse the list of Python statements to gather the list of variables, which we can then allocate. The formulas often contain constants which need to be initialized into the big number structure, which is performed on the same pass. The finally the individual

operations are extracted from the statements and generated. In this step, the statement `t5 = X1 * Y2` becomes the function call `bn_mod_mul(&X1, &Y2, &t5, &curve->p);`

- **Scalar multiplier**. The scalar multipliers are implemented in C manually, the correct one is then selected by the configuration and only the options are to the template.

- **Hash algorithm**. The hash algorithms are all implemented in C, the intended one is chosen by the configuration.

- **Random sampling**. Both of the random sampling techniques are implemented manually, the correct one is chosen by the configuration.

- **Big number arithmetic + Modular arithmetic**. The big number arithmetic used is from the libtommath library [111], which implements all of the configurable multiplication, squaring and reduction algorithms as well as many other necessary big number functions. We wrap this library with custom functions and type aliases, such that the rest of the implementation is decoupled from the choice of the big number library. The configuration for the big number arithmetic and modular arithmetic then simply changes some build time and runtime settings in the libtommath library.

Building

After source code generation, the implementation is in the form of a basic makefile-based C project, residing in a temporary directory, with all of the common and generated code together. The toolkit currently supports three targets: `STM32F0`, `STM32F3` and `HOST`, with the first two being embedded micro-controllers from STMicroelectronics and the last being the host device on which the code generation is taking place. Building this implementation for the embedded targets requires setting the architecture-specific compiler prefixes for cross-compilation and options (e.g. `arm-none-eabi` for the ARM boards) and running the make build.

All of this compiler configuration and building is performed automatically by a command-line tool (called `builder`) that is a part of the toolkit. The tool generates the sources and builds them, providing the user with the final runnable implementation in just a few moments (see Listing 3.3 for an example).

```
$ builder build --platform HOST shortw jacobian add-2007-bl
>                               dbl-2007-bl ltr() .
[ ] Rendering...
[*] Rendered.
[ ] Building...
[*] Built.
pyecsca-codegen-HOST.elf
pyecsca-codegen-HOST.hex
```

Listing 3.3: Example of a builder command that produces an implementation for the host device, using Jacobian coordinates on the short-Weierstrass curve model, formulas from a paper in 2007 and a left-to-right double and add scalar multiplier.

Flashing

When targeting the embedded devices, one needs to load the built implementation on them (i.e. *flash* them) before running. As we use the ChipWhisperer target boards, we can use the programmer functionality which the ChipWhisperer toolkit provides. It uses the bootloader provided on the `STM32` chips by the manufacturer to load code. It also requires the ChipWhisperer-Lite board to communicate with the chips.

Running

The process of running an implementation obviously differs between the embedded devices and the host device. However, the interface the implementation provides remains mostly the same. It is a text-based command-response based interface, where the implementation waits for commands to execute, then executes them and returns the results, while maintaining internal state. The commands allow for actions like setting of domain parameters, generation of keypairs, key exchange or signing.

The implementation on the host device uses the standard input/output streams to communicate. The embedded targets communicate using a serial interface (USART), which is exposed on the `STM32` target boards and routed through the ChipWhisperer-UFO board to the ChipWhisperer-Lite board (see Figure 3.3 for a diagram). We then use the ChipWhisperer library to communicate with the ChipWhisperer-Lite board, which forwards the communication to the target board.

```
$ client --platform HOST --fw pyecsca-codegen-HOST.elf shortw
>        jacobian gen secg/secp192r1
(3335971873836879180263929578908692834772062925139859222256,
Point(
[x=2654177139295206431444678278920583929836118551779554742053,
 y=2046145692848826796754295446423869958317523275155794805131]
in AffineCoordinateModel("affine" on short Weierstrass curves)))
```

Listing 3.4: Example of a client command that generates a keypair on the `secp192r1` curve using an implementation on the host device which uses the short-Weierstrass curve model and Jacobian coordinate system.

Both flashing and running a built implementation, on both the host and embedded targets, is possible with a command-line tool that is part of the toolkit (called `client`). Listing 3.4 shows an example output of running an implementation on the host. Even though this tool provides a way to run a built implementation, the intended use of the toolkit is to program this directly, using the toolkit library, into any attack/collection workflow the user might have.

The implementation on embedded targets provides one more capability than the host device one, it allows to generate a trigger signal on an exposed GPIO pin when certain actions are performed. The actions on which the target can trigger are exactly the actions specified in Subsection 3.1.3, which the toolkit traces in its software simulation of ECC. The actions that produce a trigger are configurable at runtime via a command. This dynamic triggering allows one to align the traces produced by an implementation precisely, to study the structure of the leakage trace based on the actions performed and is very versatile.

3.1.5 Acquisition

In power or EM-based side-channel attacks, one communicates with a *target* while using a *scope* to measure its power or EM radiation, for an example of this see Figure 3.5. For flexibility, the **pyecsca** toolkit supports several options for both communication with a target and measurement scope.

On the measurement side, the toolkit supports EM radiation and power-consumption acquisition using various digital sampling oscilloscopes. These scopes connect to the host usually via USB or an Ethernet connection and communicate with it using some (often proprietary) protocols, which allow the host PC to instruct the scope to start a measurement, return measured traces or perform

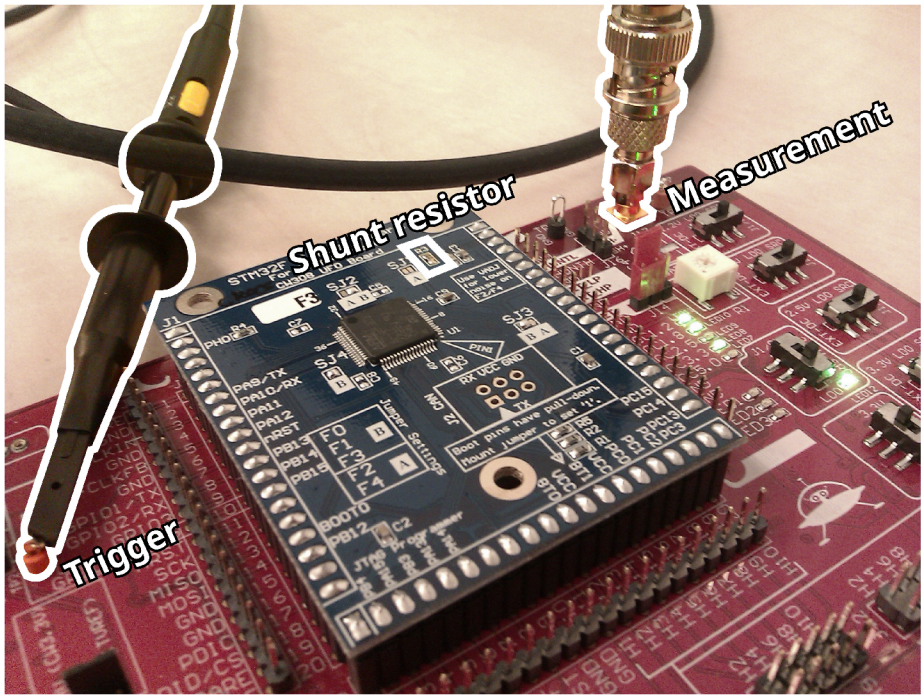


Figure 3.4: Image of the ChipWhisperer UFO board with the ChipWhisperer STM32F3 target board.

other configuration. The protocols, configuration options and properties of the scopes vary heavily between manufacturers and even models. For this reason, we selected a subset of digital sampling oscilloscopes to support which are similar enough that an abstraction layer can be created in the toolkit and all the scope specific details can be hidden. The toolkit supports all PicoScope brand oscilloscopes using two popular Python libraries to communicate with them, the official Python SDK wrappers by Pico Technology [112] and a community library picopython that better abstracts the scopes [113]. The ChipWhisperer-Lite built-in oscilloscope is also supported. However, it offers capture of up to only 24 000 samples at a time, which makes it hard to use it to capture the often long elliptic curve cryptography operations. This scope is a clock-synchronous sampling oscilloscope [76], which has been shown to reduce the number of traces required for attacks. Finally, the toolkit allows to easily implement support for other oscilloscopes, which can then be used without changing the code performing the acquisition, as a common API is used.

The diversity of possible targets and their communication protocols makes it hard to support any significant number of them in the toolkit out of the box.

We thus choose to keep the API very general, to allow for anything to be a target (with user-defined communication methods), but implement two main target protocols that are used in the rest of the work, the ChipWhisperer SimpleSerial protocol and the ISO7816-4 smartcard protocol. The ChipWhisperer SimpleSerial protocol uses the ChipWhisperer python library and the ChipWhisperer-Lite board to communicate with an embedded target using a serial protocol, it is also the communication protocol used in the implementations of ECC generated by the toolkit (see Subsection 3.1.4). The support for ISO7816-4 smartcard targets is useful for analysis of commercial smartcards with ECC support as well as any hardware that communicates using ISO7816-4 APDUs, like some hardware security tokens or cryptocurrency wallets. As ISO7816-4 is only the wire protocol for communication of the reader with the card, the PCSC (Personal Computer/Smart Card) framework is used to send commands to the reader, usually via USB, using the pyscard library [114].

The toolkit contains a class that uses the ISO7816-4 support to create a target out of the ECTester applet running on a JavaCard. ECTester [115] is a tool for testing correctness and security of ECC implementations in smartcards of the JavaCard platform and in software cryptographic libraries. The ECTester applet uses the ECC functionality provided by the card using the JavaCard APIs, this allows it to be used to perform side-channel analysis of the hardware cryptography implementations in JavaCard-based smartcards.

3.1.6 Processing

Processing of collected side-channel traces is an important part of many side-channel attacks and analysis tasks. To this end, the toolkit aims to provide a comprehensive suite of functions for trace alignment, filtering, pattern matching, signal-processing and statistical tests. This is all possible with relatively little code on the part of the toolkit, as Python offers a strong scientific computing basis of libraries like numpy and scipy [103, 104].

Alignment Trace alignment is a key element of vertical side-channel attacks. The toolkit implements several methods which can be used to align several traces to a part of a reference trace. Most of the methods only differ in the function used to compute the distance of parts of two traces. These methods all search for a shift of the aligned part of a trace which results in the lowest distance to the reference part of a trace. The used distance functions are cross-correlation, sum-of-absolute differences and peak to peak alignment.

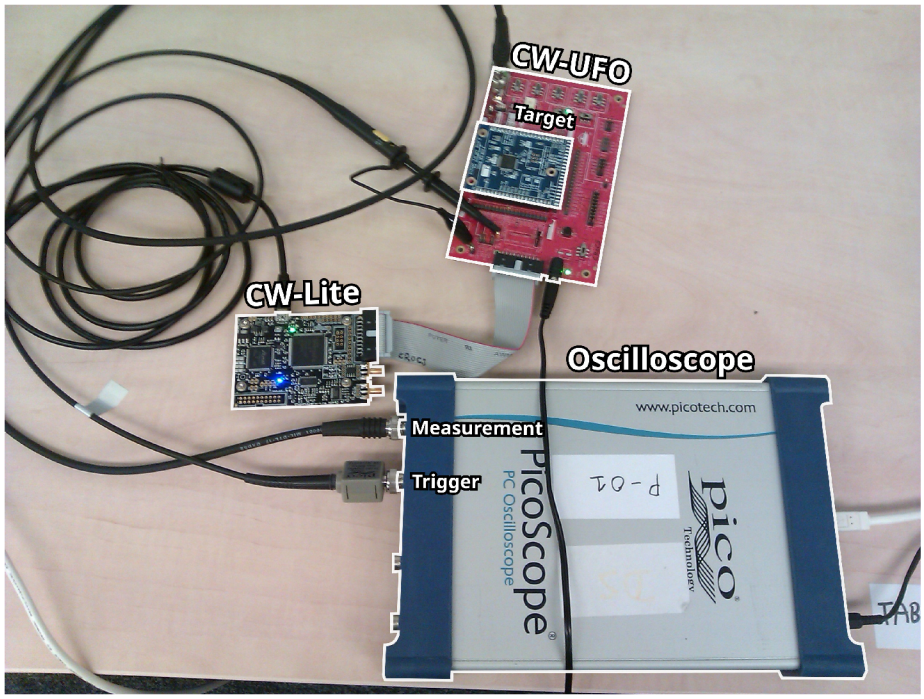


Figure 3.5: Image of the measurement setup for the ChipWhisperer UFO targets.

A different approach to alignment of time series exists in the *Dynamic Time Warp algorithm* (DTW), which warps the whole target trace using a warp path constructed such that the distance of the two traces is minimized [116]. Figure 3.6 displays an example of this warping. The toolkit implements two DTW-based alignment methods, one based on FastDTW [117], which is an approximate algorithm for DTW, and one based on elastic alignment [118], which is an extension of FastDTW that has been shown to work well in side-channel analysis.

Signal processing The toolkit offers digital signal filters (low-pass, high-pass and band-pass), downsampling of traces by averaging, picking n-th samples or decimation as well as generic processing of traces such as normalization, recentering, thresholding, computation of a rolling-mean and others. All of these functions are very useful when working with high-frequency noisy side-channel data. For an example application of digital signal filters see Figure 3.7.

Pattern matching Pattern matching of traces and their parts is a useful tool for simple side-channel attacks and even some timing attacks. In the

toolkit, it is done via cross-correlation of the pattern with the searched trace and subsequent peak-finding in the correlation signal.

Editing The toolkit also provides utility functions for trace editing such as combining traces via averaging, adding or subtracting, as well as padding and cutting of traces.

Statistical tests Statistical tests are heavily used in leakage assessment and other parts of side-channel attacks. Welch's t-test for equality of means (unequal variance), Student's t-test for equality of means and Kolmogorov-Smirnov test for equality of distributions are available in the toolkit.

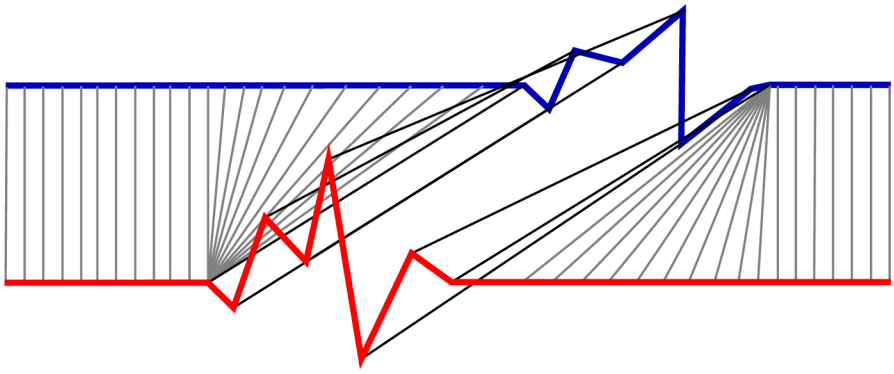


Figure 3.6: Example of trace alignment via the Dynamic Time Warp algorithm.

3.1.7 Visualization

The amounts of data in side-channel attacks can be staggering, with traces that are millions of samples long and with trace sets that contain up to hundreds of thousands of traces, the trace sets can be tens or of gigabytes. Visualizing or plotting even a single trace, which contains several million samples, can be a task that will slow down plotting libraries such as matplotlib [119] to practically unusable speeds.

We thus use the datashader [120] graphics pipeline, which is built to accurately render very large datasets, it internally uses the holoviews and bokeh [121, 122] frameworks. This achieves a sub-second rendering latency when rendering a few traces with millions of samples, without utilizing a dedicated graphics card, which is comparable to commercial side-channel attack toolkits such as Riscure Inspector. Furthermore, the holoviews framework is fully compatible with Jupyter notebooks, which form the main user interface of the toolkit. Two example traces visualized with datashader are displayed in Figure 3.7.

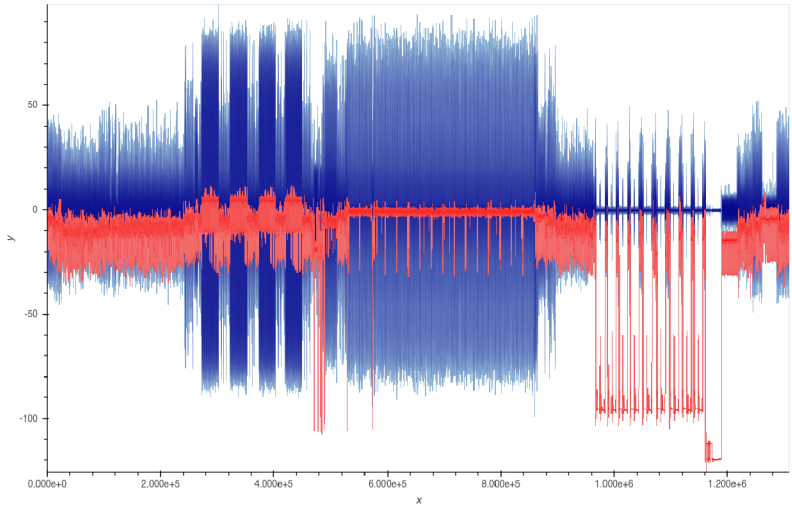


Figure 3.7: Example trace after being passed through two filters (lowpass filter, highpass filter).

3.1.8 IO

Due to the aforementioned large datasets, managing and storing them properly is important. For interoperability with other tools, the toolkit supports several file formats for storage of trace sets. The toolkit has both read and write support for trace sets from the Riscure Inspector side-channel analysis toolkit (`.trs`) as well as read-only support for ChipWhisperer trace sets. A custom Python pickle-based format is used as a basic store of trace sets that are small enough to fit in memory. It also provides a way to store arbitrary metadata with the traces.

The trace sets used in side-channel analysis often do not fit in the memory of the device on which analysis is being performed. This poses a problem for designing algorithms that process the traces in a way that does not require them to be in memory at the same time. Such algorithms exist for many aspects of side-channel attacks, yet they are unusable if the file storage layer does not support such partial loading of traces. To support such large trace sets, we use the HDF5 [123] (Hierarchical Data Format Version 5) file format using the h5py library [124]. This library allows effective storage of very large datasets, with support for compression, and their use without the necessity of loading the whole dataset into memory. This file format also presents some limitations, as storing arbitrary metadata is not straightforward and needs custom encoding and decoding routines.

3.1.9 Notebooks

Providing a usable interface to a toolkit like **pyecsca** can be challenging. Simply releasing it as a bunch of library functions and classes with basic API documentation would not be conducive to users and the toolkit would likely not see use apart from our internal use. However, going the other way and building a rich graphical user interface application is a huge task. Commercial tools, like the Riscure Inspector, have lots of resources to give to this as usability is core to their business. Even then, since novel side-channel attacks require novel techniques, the application needs to support scripting and dynamic introspection in some way and often ends up being very complicated. The open-source ChipWhisperer project originally used a standalone graphical interface as well, but abandoned it during a recent overhaul in favor of Jupyter notebooks [105].

Generating

We will first create a `DeviceConfiguration`, which we will then generate and build.

```
[3]: from pyecsca.ec.model import ShortWeierstrassModel
     from pyecsca.ec.mult import LTRMultiplier
     from pyecsca.ec.configuration import *

     platform = Platform.STM32F3
     hash_type = HashType.SHA1
     mod_rand = RandomMod.REDUCE
     mult = Multiplication.BASE
     sqr = Squaring.BASE
     red = Reduction.BASE

     model = ShortWeierstrassModel()
     coords = model.coordinates["projective"]
     add = coords.formulas["add-1998-cmo"]
     dbl = coords.formulas["dbl-1998-cmo"]
     scl = coords.formulas["z"]
     formulas = [add, dbl, scl]
     scalarmult = LTRMultiplier(add, dbl, scl)

     config = DeviceConfiguration(model, coords, formulas, scalarmult,
                                                  hash_type, mod_rand, mult, sq
                                                  platform, True, True, True)

     config

[3]: DeviceConfiguration(model=ShortWeierstrassModel(), coords=EFDCoordinateModel("proj
```

Now we can render the configuration, which will generate the source files into a randomly created temporary directory, and return the path to the directory as well as names of the elf and hex files which will be built in that directory.

```
[4]: from pyecsca.codegen.builder import render

     directory, elf_name, hex_name = render(config)

     print(directory)

     /tmp/tmplwtoeyhk
```

Figure 3.8: Excerpt from a Jupyter notebook showing the process of generating an ECC implementation.

We also use Jupyter notebooks as an interface to the toolkit. A Jupyter notebook is accessible as a page in a web application that contains live code, visualizations and text. See Figure 3.8 for a screenshot of a part of one of the notebooks available in the toolkit.

3.2 Cryptographic library analysis

The availability of many open-source cryptographic libraries with support for elliptic curve cryptography provides an opportunity to use these libraries to analyze what implementation choices they made. We performed this analysis on a sample of 10 open-source libraries: BouncyCastle, BoringSSL, Botan, Crypto++, libgcrypt, LibreSSL, libsecp256k1, libtomcrypt, Intel Performance Primitives - Crypto and Microsoft CNG. The results of this analysis may be applicable even to closed source implementations and to some degree even to implementations in hardware, as the differences between closed and open-source should not be significant.

We analyzed their source code to determine what curve model, coordinate system and scalar multiplier they are using for ECDH (see Table 3.2), X25519 and X448 (see Table 3.3), ECDSA (see Table 3.4) and EdDSA (see Table 3.5). Furthermore, we make the distinction between the operations KeyGen, Derive, Sign and Verify as different implementation choices can be made in each one. For example, it is reasonable to see a fixed-base scalar multiplier used in KeyGen, then a variable-base one in Derive and Sign and a multi-scalar one perhaps even without side-channel countermeasures in Verify. It might be possible to perform deeper analysis and extract the individual formulas or finite field arithmetic details used in these implementations. However, we did not pursue this as without documentation it would have been extremely tedious.

Recovering implementation details via source code analysis is a complicated process as implementations are often undocumented and very hard to read. They also very often have multiple implementations for the same operation and choose one dynamically based on what curve is being used. We ignore this in our analysis and specify just the generic implementation details, but note that libraries that have curve specific implementations most often have them for the NIST P-256 curve. Our analysis is a best-effort one and might contain misclassifications, especially in determining the scalar multiplier used, as it is an implementation property that does not have clearly defined categories and names.

Several observations can be made from the collected data. It seems that Jacobian coordinates are very popular for short-Weierstrass curves. Also, the prevalence of prime order short-Weierstrass curves (which do not have an equivalent representation in other models) in use for ECDH and ECDSA ensures that libraries also implement said cryptosystems using the short-Weierstrass curve model. Furthermore, in several libraries (BouncyCastle, BoringSSL and LibreSSL) the equivalence between Montgomery and twisted-Edwards curves is used in practice where the X25519 key generation is performed using an equivalent twisted-Edwards curve.

Library	Operation	Curve	Scalar multiplier	Coordinates
BouncyCastle	KeyGen	$\mathcal{E}_{SW}$	Comb	modified
1.65	Derive	$\mathcal{E}_{SW}$	Window NAF	modified
BoringSSL	KeyGen	$\mathcal{E}_{SW}$	Fixed window	jacobian
a810d82	Derive	$\mathcal{E}_{SW}$	Fixed window	jacobian
Botan	KeyGen	$\mathcal{E}_{SW}$	Fixed window	jacobian
2.14	Derive	$\mathcal{E}_{SW}$	Fixed window	jacobian
Crypto++	KeyGen	$\mathcal{E}_{SW}$	-[1]	affine
8.2.0	Derive	$\mathcal{E}_{SW}$	-	affine
libgcrypt	KeyGen	$\mathcal{E}_{SW}$	Basic left-to-right	jacobian
1.8.5	Derive	$\mathcal{E}_{SW}$	Basic left-to-right	jacobian
LibreSSL	KeyGen	$\mathcal{E}_{SW}$	Ladder	jacobian
3.1.3	Derive	$\mathcal{E}_{SW}$	Ladder	jacobian
libtomcrypt	KeyGen	$\mathcal{E}_{SW}$	Sliding window	jacobian
0.18.2	Derive	$\mathcal{E}_{SW}$	Sliding window	jacobian
IPP-crypto	KeyGen	$\mathcal{E}_{SW}$	Window NAF	jacobian
2020	Derive	$\mathcal{E}_{SW}$	Window NAF	jacobian
Microsoft CNG	KeyGen	$\mathcal{E}_{SW}$	Fixed window	jacobian
263e3e6	Derive	$\mathcal{E}_{SW}$	Fixed window	jacobian

[1] We were unable to classify what scalar multiplier is used in Crypto++.

Table 3.2: Library analysis results (ECDH).

Library	Operation	Curve	Scalar multiplier	Coordinates
BouncyCastle	KeyGen	$\mathcal{E}_{TE}$	Signed comb	-
1.65	Derive	$\mathcal{E}_{M}$	Montgomery ladder	xz
BoringSSL[1]	KeyGen	$\mathcal{E}_{TE}$	Comb	yz
a810d82	Derive	$\mathcal{E}_{M}$	Montgomery ladder	xz
Botan[2]	KeyGen	$\mathcal{E}_{M}$	Montgomery ladder	xz
2.14	Derive	$\mathcal{E}_{M}$	Montgomery ladder	xz
Crypto++[2]	KeyGen	$\mathcal{E}_{M}$	Montgomery ladder	xz
8.2.0	Derive	$\mathcal{E}_{M}$	Montgomery ladder	xz
libgcrypt	KeyGen	$\mathcal{E}_{M}$	Montgomery ladder	xz
1.8.5	Derive	$\mathcal{E}_{M}$	Montgomery ladder	xz
LibreSSL[1]	KeyGen	$\mathcal{E}_{TE}$	Comb	yz
3.1.3	Derive	$\mathcal{E}_{M}$	Montgomery ladder	xz

[1] Based on the ref10 Ed25519 implementation.
[2] Uses the curve25519-donna implementation.

Table 3.3: Library analysis results (X25519 and X448).

Library	Operation	Curve	Scalar multiplier	Coordinates
BouncyCastle `1.65`	KeyGen	$\mathcal{E}_{SW}$	Comb	modified
	Sign	$\mathcal{E}_{SW}$	Comb	modified
	Verify	$\mathcal{E}_{SW}$	Window NAF, Shamir's trick	modified
BoringSSL `a810d82`	KeyGen	$\mathcal{E}_{SW}$	Fixed window	jacobian
	Sign	$\mathcal{E}_{SW}$	Fixed window	jacobian
	Verify	$\mathcal{E}_{SW}$	Multi-scalar window NAF	jacobian
Botan `2.14`	KeyGen	$\mathcal{E}_{SW}$	Fixed window	jacobian
	Sign	$\mathcal{E}_{SW}$	Fixed window	jacobian
	Verify	$\mathcal{E}_{SW}$	Multi-scalar	jacobian
Crypto++ `8.2.0`	KeyGen	$\mathcal{E}_{SW}$	-[1]	affine
	Sign	$\mathcal{E}_{SW}$	-	affine
	Verify	$\mathcal{E}_{SW}$	-	affine
libgcrypt `1.8.5`	KeyGen	$\mathcal{E}_{SW}$	Basic left-to-right	jacobian
	Sign	$\mathcal{E}_{SW}$	Basic left-to-right	jacobian
	Verify	$\mathcal{E}_{SW}$	-[1]	jacobian
LibreSSL `3.1.3`	KeyGen	$\mathcal{E}_{SW}$	Ladder	jacobian
	Sign	$\mathcal{E}_{SW}$	Ladder	jacobian
	Verify	$\mathcal{E}_{SW}$	Multi-scalar window NAF	jacobian
libsecp256k1 `a39c2b0`	KeyGen	$\mathcal{E}_{SW}$	Window NAF	(mixed)
	Sign	$\mathcal{E}_{SW}$	Window NAF	(mixed)
	Verify	$\mathcal{E}_{SW}$	Window NAF, Shamir's trick	(mixed)
libtomcrypt `0.18.2`	KeyGen	$\mathcal{E}_{SW}$	Sliding window	jacobian
	Sign	$\mathcal{E}_{SW}$	Sliding window	jacobian
	Verify	$\mathcal{E}_{SW}$	Sliding window	jacobian
IPP-crypto `2020`	KeyGen	$\mathcal{E}_{SW}$	Window NAF	jacobian
	Sign	$\mathcal{E}_{SW}$	Window NAF	jacobian
	Verify	$\mathcal{E}_{SW}$	Window NAF	jacobian
Microsoft CNG `263e3e6`	KeyGen	$\mathcal{E}_{SW}$	Fixed window	jacobian
	Sign	$\mathcal{E}_{SW}$	Fixed window	jacobian
	Verify	$\mathcal{E}_{SW}$	Multi-scalar window NAF	jacobian

[1] We were unable to classify what scalar multiplier is used in parts of Crypto++ and libgcrypt.

Table 3.4: Library analysis results (ECDSA).

Library	Operation	Curve	Scalar multiplier	Coordinates
BouncyCastle `1.65`	KeyGen	$\mathcal{E}_{TE}$	Signed comb	-
	Sign	$\mathcal{E}_{TE}$	Signed comb	-
	Verify	$\mathcal{E}_{TE}$	Window NAF, Shamir's trick	-
BoringSSL[1] `a810d82`	KeyGen	$\mathcal{E}_{TE}$	Comb	yz
	Sign	$\mathcal{E}_{TE}$	Comb	yz
	Verify	$\mathcal{E}_{TE}$	Multi-scalar	(mixed)
Botan[1] `2.14`	KeyGen	$\mathcal{E}_{TE}$	Comb	yz
	Sign	$\mathcal{E}_{TE}$	Comb	yz
	Verify	$\mathcal{E}_{TE}$	Multi-scalar	(mixed)
Crypto++ `8.2.0`	KeyGen	$\mathcal{E}_{TE}$	-[2]	Niels
	Sign	$\mathcal{E}_{TE}$	-	Niels
	Verify	$\mathcal{E}_{TE}$	-	Niels
libgcrypt `1.8.5`	KeyGen	$\mathcal{E}_{TE}$	Basic left-to-right	projective
	Sign	$\mathcal{E}_{TE}$	-[2]	projective
	Verify	$\mathcal{E}_{TE}$	-	projective
LibreSSL[1] `3.1.3`	KeyGen	$\mathcal{E}_{TE}$	Comb	yz
	Sign	$\mathcal{E}_{TE}$	Comb	yz
	Verify	$\mathcal{E}_{TE}$	Multi-scalar	(mixed)

[1] Based on the ref10 Ed25519 implementation.
[2] We were unable to classify what scalar multiplier is used in parts of Crypto++ and libgcrypt.

Table 3.5: Library analysis results (EdDSA).

3.3 Reverse-engineering methods

In this section, we present several novel methods for reverse-engineering elliptic curve cryptography implementation details on black-box devices via side-channel analysis. The general scenario is that the reverse-engineer has a physical device implementing some ECC functionality like ECDH, ECDSA and key generation, yet it is a complete black-box which can only be queried via the mentioned cryptosystems and their functions.

The above scenario is a common one as implementation details are generally not available for commercial products implementing ECC such as smartcards, hardware security modules (HSM), trusted platform modules (TPM) or other hardware accelerators. Namely, the JavaCard platform is a very popular platform for programmable smartcards that provides access to ECC functionality which many cards implement.

Almost all of the attacks described in Section 2.3 require that the attacker has precise knowledge of the implementation under attack (see [91] for an overview). Indeed, most works introducing new side-channel attacks start with this assumption or have it implicitly. For example, Bauer et al. [68] state: "*The algorithm used for the hardware modular multiplication is assumed to be known to the attacker*". The hardness of satisfying assumptions like the one above, for a real-world attacker, was not studied before. Given the large space of possible implementations as enumerated in Subsection 3.1.1 and Tables 3.1a to 3.1c, it may be possible that reverse-engineering the implementation details is the harder part of a side-channel attack.

Our reverse-engineering methods require varying levels of control over the inputs to the target, some work with random inputs and standard domain parameters, some require the ability to set custom domain parameters. The methods are not specifically designed to circumvent countermeasures to side-channel attacks (as discussed in Section 2.4), as that would require significant effort out of the scope of this work. However, we note that the main goal of side-channel countermeasures is to protect the secret keys used by the implementation, not to protect the implementation against reverse-engineering, thus the countermeasures might not be as effective.

3.3.1 Leakage assessment

In our first and simplest reverse-engineering method, we use the test-vector leakage assessment technique, as described in Subsection 2.3.1. We use the specific variant of TVLA in which random inputs are provided to the implementation (for example, random private and public keys in the ECDH Derive operation) and

traces are collected. We then simply perform leakage assessment using all of the plausible implementation configurations, by selecting an intermediate value, partitioning the traces accordingly and performing a Welch's t-test for the difference of means (for each plausible configuration). As the intermediate values computed during the operation clearly depend on the used configuration, the partitioning will be correct only if the intermediate value in the simulated configuration is highly correlated to the real intermediate value processed by the implementation. Thus leakage will be detected only with the correct configuration hypothesis.

This method, while simple, has several drawbacks. It is very computationally heavy, as it requires performing a significant amount of t-tests. However, after the initial collection of traces with random inputs, it is completely parallelizable. It also assumes leakage detectable with TVLA, which side-channel protected implementations do not generally show, or only show with a large amount of traces and possibly by using higher orders. However, this is also a good property, if reverse-engineering succeeds, the assessed leakage can be used to mount an attack.

Capabilities. Curve model, coordinate system, scalar multiplier, formulas.
Type. Offline.
Side-channel. Power or EM radiation.
Target. Key generation, ECDH or ECDSA with random inputs.

3.3.2 RPA

Goubin's Refined Power Analysis [82] constructs specific input points to the ECDH Derive operation that lead to a multiple of the input point having a zero coordinate, which can then be detected using a side-channel. We change this attack to form hypotheses over the implementation configurations and not over the secret scalar and thereby obtain a reverse-engineering method that can recover the scalar multiplier used in an implementation.

Given a known private key, we simulate the execution of the Derive operation using all of the hypothesized configurations and record which multiples of the input point are computed in each configuration. These lists of multiples of the input point will differ between the configurations, as different scalar multipliers take different addition chains to reach the final multiple. We then pick a multiple m that appears in roughly half of the configurations and compute $P_1 = [m^{-1}]P_0$ where P_0 is the point with a zero coordinate on the hypothesized curve model. Next, we query the target with the P_1 point as input to the Derive operation and observe that if the $[m]P_1 = P_0$ multiple is computed, we can detect the zero

coordinate point P_0 from the side-channel trace and thus reject the configurations which do not compute the m-th multiple. The pseudocode of the complete method is in Algorithm 6.

Algorithm 6 RPA-based RE

```
function RPA-RE(E(F_p), C, P_0 ∈ E(F_p))
    d, P ← KeyGen(E(F_p), G, n)
    Q ←$ E(F_p)
    for i = 0 to N do
        l_i ←trace→ Derive_target(d, Q)
    multiples: C → Z^r    // mapping of configurations to addition chains
    for c in C do
        Derive_c(d, Q)
        store computed multiples under multiples[c]
    while |C| > 1 do
        counts: Z → N    // mapping of multiples to their counts
        for c in C do
            for m in multiples[c] do
                counts[m] += 1
        pick m ∈ counts s.t. counts[m] ~ |C|/2
        P_1 = [m^{-1} mod n]P_0
        for i = 0 to N do
            l'_i ←trace→ Derive_target(d, P_1)
        if Δ[(l_i)_i, (l'_i)_i] distinguishes then
            C ← {c ∈ C | m ∈ multiples[c]}    //P_0 appeared
        else
            C ← {c ∈ C | m ∉ multiples[c]}    //P_0 did not appear
    if |C| = 0 then
        return error, the configuration could not be reversed
    else
        return the resulting config ∈ C
```

Capabilities. Curve model, scalar multiplier.

Type. Online.

Side-channel. Power or EM radiation.

Target. ECDH with chosen inputs and known private key or ECDSA signature verification with chosen inputs and public key.

3.3.3 ZPA

Similarly to how the Zero-value Point Attack by Akishita; Takagi [83] is an extension of RPA, we can extend the RPA-based reverse-engineering method as well. When picking $m \in$ counts we will pick only "small" m, as the method requires to compute the m-th division polynomial, the degree of which grows rapidly in m. While Akishita; Takagi [83] extract the zero-value conditions for their target addition and doubling formulas manually, we can not take such a path, as we use many different formulas. We thus propose to automate this by:

1. Considering in what relation are the input points, which depends on the scalar multiplier used. For basic multipliers this is $\mathbf{add}([c]P, P)$ and $\mathbf{dbl}([c]P)$.
2. Expressing all intermediate values in a formula as polynomials in the input coordinates.
3. If necessary (multiple input points to a formula), using a division polynomial to eliminate the variables of all but one input point.
4. Fixing all Z coordinates to 1, as only affine inputs are possible in the targeted cryptosystems.
5. Using the curve equation to eliminate some variables.
6. Finding the roots of the resulting polynomial over $\mathbb{F}_p$.

This method for construction of zero-value points then replaces the computation of P_1 in the RPA-based reverse-engineering method which forms the ZPA-based reverse-engineering method.

Capabilities. Curve model, coordinate system, scalar multiplier, formulas.
Type. Online.
Side-channel. Power or EM radiation.
Target. ECDH with chosen inputs and known or chosen private key.

3.3.4 EPA

We use the ideas from the Exceptional Procedure Attack of Izu; Takagi [84] to form a reverse-engineering technique using errors as a side-channel. In earlier work [125] we found that several commercial JavaCards are able to work with invalid elliptic curve domain parameters in which the parameter p, specifying

the finite field $\mathbb{F}_p$ over which the curve is defined, is not prime. In this case, the implementations compute over $\mathbb{Z}_n$ where elements not co-prime to n do not have multiplicative inverses. It is likely that if an implementation attempts to compute an inverse of a non-invertible element it will raise an error, as both methods of computing inverses (see Subsection 1.3.5) allow for the detection of this case. Computing inverses is generally required when converting from projective to affine coordinates, which is necessary at the end of the ECDH Derive operation which is the target of this method.

If we query the implementation, for example using a random private key and public key in the ECDH Derive operation, and observe an error we can simulate the Derive operation using hypothesized implementation configurations and reject those which do not raise an error during the simulation. Conversely, if for some random inputs to Derive the target does not raise an error, we can reject those configurations that give an error in simulation with the same inputs. Algorithm 7 shows a pseudocode version of this method.

Algorithm 7 EPA-based RE

```
function EPA-RE(ℰ(ℤ_n), C)
    while |C| > 1 do
        P ←$ ℰ(ℤ_n)
        k ←$ ℤ_ord(P)
        R_target, success_target ← DERIVE_target(k, P)
        for c in C do
            R_c, success_c ← DERIVE_c(k, P)
            if success_target ≠ success_c or R_target ≠ R_c then
                C ← C/{c}
    if |C| = 0 then
        return error, the configuration could not be reversed
    else
        return the resulting config ∈ C
```

To evaluate the feasibility of the suggested method, we compute the expected amount of plausible configurations C left after the i-th iteration of the while loop, we denote this $\mathbb{E}[\mathbf{C}_i]$. Clearly $\mathbb{E}[\mathbf{C}_0] = |C|$. Let $p_1 = \frac{\varphi(n)}{n}$ be the probability of a uniformly distributed random element of $\mathbb{Z}_n$ being invertible. Let $p_0 = 1 - p_1$ denote the converse. We assume that only one inversion is performed during the operation, at its end, to obtain the affine x coordinate from the projective point. We further assume that the element Z of the projective point that is inverted is uniformly distributed over $\mathbb{Z}_n$. Under these assumptions, probability of success

of the Derive operation is p_1. Thus, at each iteration Derive succeeds with probability p_1 and we keep p_1 of the configurations or fails with probability p_0 and we keep p_0 of the configurations. We then have:

$$\begin{aligned}\mathbb{E}\left[\mathbf{C}_i\right] &= p_1^2\mathbb{E}\left[\mathbf{C}_{i-1}\right] + p_0^2\mathbb{E}\left[\mathbf{C}_{i-1}\right] \\ &= \mathbb{E}\left[\mathbf{C}_0\right](p_1^2 + p_0^2)^i.\end{aligned}$$

and for the loop to end in expected l iterations we have with $p_r = p_1^2 + p_0^2$:

$$\begin{aligned}\mathbb{E}\left[\mathbf{C}_l\right] &= 1 \\ \mathbb{E}\left[\mathbf{C}_0\right](p_r)^l &= 1 \\ |C|(p_r)^l &= 1 \\ l &= -\log_{p_r}(|C|).\end{aligned}$$

Experimentally, for $|C|$ of one million and n with 262 bits consisting of 25 distinct prime factors, we obtained $p_1 = 0.969$ and $p_r = 0.940$ giving $l = 224$. Thus one can expect to recover the implementation in only 224 Derive_{target} operations.

Capabilities. Curve model, coordinate system, scalar multiplier, formulas.
Type. Online or Offline.
Side-channel. Errors, results, timing.
Target. ECDH or ECDSA signature verification with random inputs but chosen domain parameters.

3.4 Evaluation

We chose to evaluate the leakage assessment method on both implementations generated by the toolkit for the `STM32F3` chip (where we know and can choose the implementation configuration) and on an implementation in a commercial JavaCard (where implementation details are unknown).

3.4.1 Measurement setup

The measurement setup for generated targets on an embedded micro-controller consisted of a PicoScope 4224 oscilloscope, ChipWhisperer-Lite board, ChipWhisperer UFO board and an `STM32F3` ChipWhisperer target board. The signal was amplified using the ChipWhisperer low-noise amplifier and was not filtered in any analog way.

For the JavaCard smartcards the measurement setup consisted of a PicoScope 4224 oscilloscope and a Gemplus GemPC Twin USB card reader. The power consumption was measured in the ground line using a 27 Ω shunt resistor. We note that this setup is not ideal, as the card is powered by the USB power supply of the host machine, which introduces noise in the measurement. Commercial solutions use capacitors, batteries or filtering to deliver a noise-free power supply to the target card.

3.4.2 Experiment 1 - Generated implementations

Our first experiment evaluated the leakage assessment method of reverse engineering on generated implementations on micro-controllers. The whole experiment consisted of 5 parts, data collection, data pre-processing, simulation, partitioning and testing.

Data collection

One thousand traces of the target board performing $[5]P$ with random points P on the `secp128r1` curve were collected. The implementation used a short-Weierstrass curve model, projective coordinates, left-to-right basic scalar multiplier and formulas `add-2007-bl` and `dbl-2007-bl` from the EFD. The scope was running at a sampling rate of 20 MHz and collected 5 million samples each run, which were then trimmed according to a dynamic trigger sent by the implementation, to only keep the scalar multiplication part of the power trace. The resulting trace set was over 13 GB, which highlights the importance of having methods that work with trace sets partially on disk and not all in memory. The full trace collection took around one hour.

Data pre-processing

Due to the fact that the big-number arithmetic library used in the generated implementations is not fully constant-time, the collected traces were unequal length and misaligned. Intentional misalignment by insertion of dummy waits and clock speed variations is a countermeasure used in side-channel protected implementations. So the generated implementations were "accidentally" protected in this way.

Aligning the full traces via DTW or its approximate variant FastDTW would have been prohibitively expensive and also might not have worked if the misalignment was too severe. We thus opted to align individual parts of the traces and then join them together. After using a digital lowpass filter, peaks separating the individual formula executions were apparent (see Figure 3.9). These peaks

were used to split the traces into parts corresponding to the individual applications of addition formulas. The parts were then aligned across the trace set using elastic time warping [118] and joined back together. Pre-processing was the most computationally heavy step and it took almost 10 hours (without parallelization).

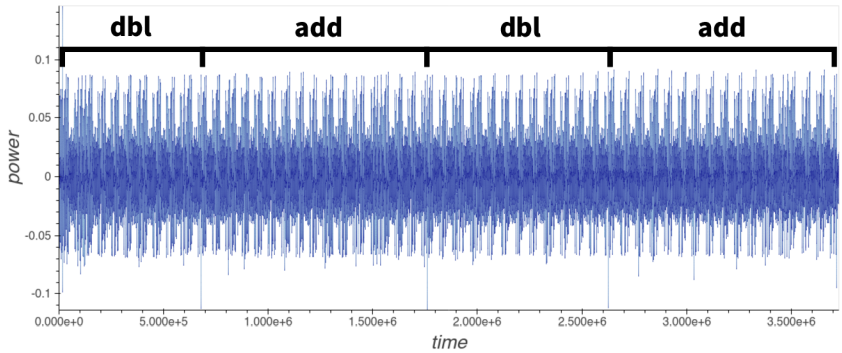


Figure 3.9: A power trace after applying a lowpass filter with a cutoff of 40kHz. The formulas applied (**dbl**, **add**, **dbl**, **add**) are separated by thin negative peaks. Note the the clear finite field multiplications.

Simulation and partitioning

To simulate the intermediate values that appear during the computation of $[5]P$ we used the `DefaultContext` capabilities of the toolkit and chose as the target intermediate value the 5-th least significant bit of the X coordinate of the result of the first addition operation. We simulated this intermediate value using the correct configuration (referred to as `bl` after the formulas used in it), as well as using a wrong configuration with different formulas (`rcb`). We then partitioned the traces according to the resulting intermediate values, for testing we also included one partition where the traces were partitioned randomly (`random`).

Testing

To evaluate the method we computed the values of the Welch's t-statistic for the three partitions induced by the configurations `bl`, `rcb` and `random`, on both the original traces and on the aligned traces. The expected result being that the partitions are deemed different by the test only for the correct `bl` configuration. The usual criterion in TVLA for rejecting the null hypothesis and pronouncing the partitions different is a threshold value of the test statistic $|t| > 4.5$ [87]. We also adopt this criterion, but note that it is dependent on the number of traces and their properties.

Surprisingly, we have detected leakage with the t-value above the 4.5 threshold using all three partitions `bl`, `rcb`, `random` on both the original traces and aligned ones. Thus, a straightforward application of this method to reverse-engineering, where the detection of leakage in one intermediate value using a hypothesis on the configuration confirms the hypothesis, could not be confirmed. However, if a maximum t-value is taken, the correct configuration is chosen in this case.

There are several possible sources of issues that could have affected our results:

- The trace set being too small.
- The presence of significant misalignment that our alignment technique could not handle.
- Collisions between the chosen intermediate value and other internal values computed during the execution.

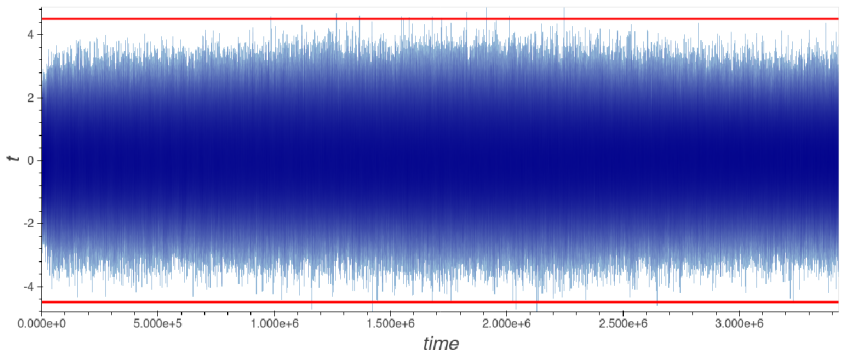


Figure 3.10: Values of the Welch's t-statistic for the correct configuration hypothesis, computed on misaligned traces.

3.4.3 Experiment 2 - JavaCards

Our second experiment evaluated the leakage assessment reverse-engineering method on a commercial JavaCard. Out of the few available cards, we chose an `Infineon CJTOP 80k` smartcard, after cursory analysis of the cards' behavior and power traces. This card showed large power consumption differences and clear patterns in the trace.

Given that the smartcard is a commercial one, it is expected that security-sensitive operations (like ECDH Derive) have side-channel countermeasures in place. Success of the leakage assessment reverse engineering on a security-sensitive operation would directly imply a vulnerability in the card. Thus we do

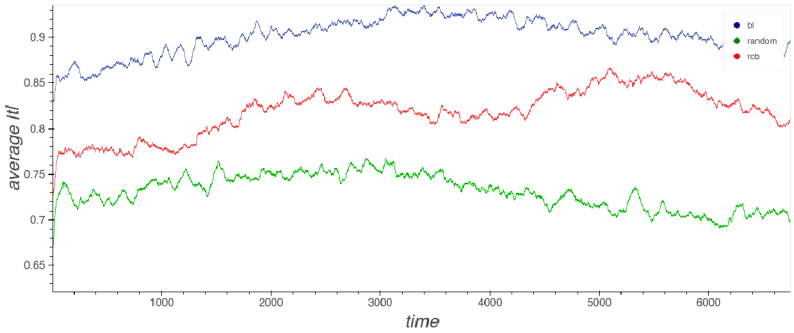


Figure 3.11: Downsampled (by averaging 5000 consecutive samples) absolute value of the Welch's t-statistic, computed on misaligned traces, for the three partitions.

not expect success, but nonetheless, explore the behavior of the method on the ECDH Derive operation and also on the ECDSA Verify operation.

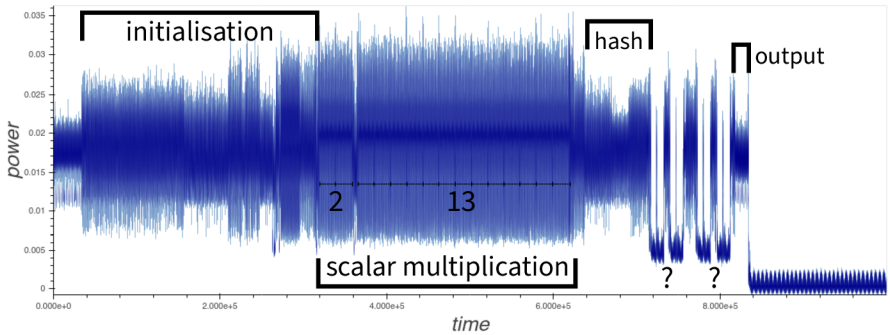
Data collection

We collected 2000 traces of the ECDH Derive operation with private key equal to 5 and random public keys (see Figure 3.12a) and 4000 traces of the ECDSA Verify operation with random public keys, messages and signature values (see Figure 3.12b), over the `secp160r1` curve. The ECDH measurement used a sampling rate of 6.66 MHz and the ECDSA measurement used 10 MHz.

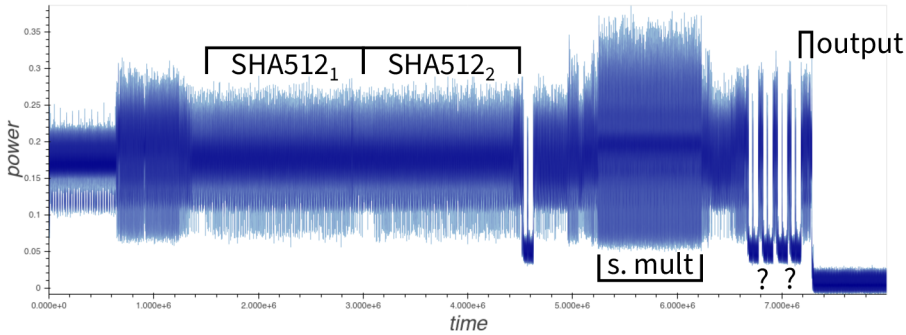
Upon further visual analysis, we determined that the scalar multiplication algorithms used in the two operations are different. The scalar multiplier used in Derive is somehow visually split into two segments, one consisting of two parts, the other of 13 parts. Each of these parts consists of 16 sub-parts and has in either zero, one, two or three of these sub-parts a peak that is visually discernible. The indices of these peaks (in which sub-part out of 16) form a pattern which likely hints at the type of scalar multiplier used. We found two sequences of indices (see also Figure 3.13):

$$[(10), (8), (5), (3), (1, 15), (13), (11), (10), (8), (6), (4), (2), (14), (12), (10)]$$
$$[(5, 11), (9, 15), (4, 10), (14), (), (), (7, 13), (), (), (6, 12), (), (), (), (2, 9, 15), ()]$$

The scalar multiplier in Verify likely does not have side-channel countermeasures in place, as the sequence of operations performed is clearly visible in the trace (see Figure 3.14).



(a) ECDH Derive with 5 as private key.



(b) ECDSA Verify.

Figure 3.12: Traces of operations on the `Infineon CJTOP 80k` smartcard over the `secp160r1` curve. (a) and (b) not to scale.

Data pre-processing

The traces did not show significant misalignment, so we chose to align them statically (i.e. only by shifting them) and not apply a dynamic alignment algorithm like DTW. We aligned the Derive traces by correlation using the part of the traces between the two segments of the scalar multiplication as the aligned pattern. Similarly, we aligned the Verify traces by correlation, using a pattern just before the start of the scalar multiplication.

Simulation and partitioning

For the ECDH Derive operation we simulate the whole execution of the operation using all short-Weierstrass configurations compatible with the curve used and partition on the 5-th least significant bit on an intermediate value in the middle of the scalar multiplication.

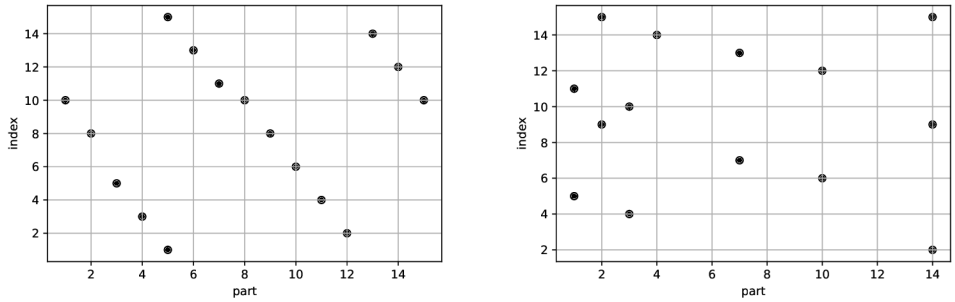


Figure 3.13: Patterns of indices in scalar multiplication in Derive on the `Infineon CJTOP 80k` card.

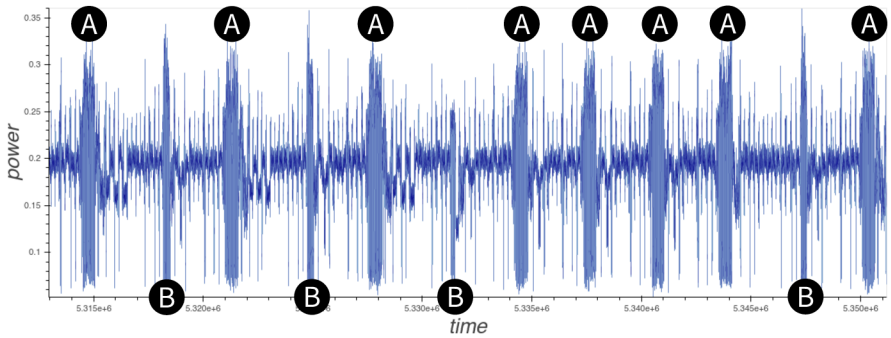


Figure 3.14: Types of operations in Verify on the `Infineon CJTOP 80k`.

For the ECDSA Verify operation, we simulate the ordinary execution of the operation using all short-Weierstrass configurations, which however do not contain multi-scalar algorithms which might be used in the operation. To account for this, we also simulate the execution of $[2]P$ and $G + P$ using all of the applicable addition and doubling formulas, where P is the public key. In all cases, we partition on the 5-th least significant bit of an intermediate value.

Testing

As expected, the t-test values for the partitions induced by the different configurations in the Derive operation are inconclusive, i.e. all show leakage at random points in the traces.

In the case of the Verify operation, the tests are similarly inconclusive, showing only slightly more points of leakage in configurations which use projective coordinates. However, in the presence of inconclusive data with clear false posi-

tives, such a thing cannot be taken as a confirmation that projective coordinates are used in the implementation.

Conclusions and future work

This work presented the **pyecsca** toolkit for side-channel analysis of elliptic curve cryptography implementations, which is currently the most extensive open-source tool aimed at side-channel analysis of elliptic curve cryptography know to us. It offers comprehensive tools for simulation and generation of ECC implementations, acquisition and processing of side-channel traces and reverse-engineering of ECC implementations.

The vast amount of possible implementations of ECC we collected together with the black-box nature of many devices show that there are numerous implementation details worthy of reverse-engineering. Keeping implementation details secret, as is encouraged by several certification schemes and widespread in the secure hardware industry, goes against the well-known Kerckhoff's principle and makes outside security analysis hard. Many secure hardware implementations do rely on security by obscurity, without explicitly stating or advertising so, as their security claim is backed by certifications with secret processes and tests, and their design is secret as well.

Relying on security by obscurity is widespread in the industry, and has lead to several attacks being discovered later than they would have been otherwise and thus having a greater impact [126, 2, 127]. Security by obscurity is widely regarded as unacceptable in the design of cryptosystems, yet somehow remains alive in hardware.

The reverse-engineering methods presented in this work aim to show that keeping implementation details of elliptic curve cryptography secret is likely a futile endeavor and that a determined attacker can gather implementation details from side-channels. These implementation details then apply to all products of a particular model, a series of models or even a manufacturer, and so the effort scales well.

We believe the presented reverse-engineering methods to be theoretically sound, yet hard to apply in practice, in a way that many side-channel attacks are. Practical issues of noise, measurement setup, alignment and many other need to be solved by trial-and-error or guessed by professionals with extensive experience. This is also apparent from the negative practical results we report. We thus propose to apply the methodology of Rioja et al. [128] that systematizes the setup and execution of side-channel attacks, in the application of reverse-engineering methods.

Appendices

Data attachment

- The **pyecsca** toolkit sources, also available at `https://github.com/J08nY/pyecsca`.
- The **pyecsca** codegen package sources, also available at `https://github.com/J08nY/pyecsca-codegen`.
- The **pyecsca** notebook package sources, also available at `https://github.com/J08nY/pyecsca-notebook`.
- The **pyecsca** documentation, also available at `https://neuromancer.sk/pyecsca/`.

Several icons used in this work are from the Noun Project: 3266818, 2473944, 1088239, 1066478.

EFD data

(embedded)

```
name short Weierstrass curves
parameter a
parameter b
coordinate x
coordinate y
satisfying y^2 == x^3 + a*x + b
addition x = (y2-y1)^2/(x2-x1)^2-x1-x2
addition y = (2*x1+x2)*(y2-y1)/(x2-x1)-(y2-y1)^3/(x2-x1)^3-y1
doubling x = (3*x1^2+a)^2/(2*y1)^2-x1-x1
doubling y = (2*x1+x1)*(3*x1^2+a)/(2*y1)-(3*x1^2+a)^3/(2*y1)^3-y1
negation x = x1
negation y = -y1
toweierstrass weierx = x
toweierstrass weiery = y
a0 = 1
a1 = 0
a2 = 0
a3 = 0
a4 = a
a6 = b
fromweierstrass x = weierx
fromweierstrass y = weiery
```

Listing 4.1: Short-Weierstrass model data from the EFD [58].

(embedded)

```
name projective coordinates
variable X
variable Y
variable Z
neutral X = 0
neutral Y = 1
neutral Z = 0
satisfying x = X/Z
satisfying y = Y/Z
```

Listing 4.2: Projective coordinate system data on the short-Weierstrass curve model (from the EFD [58]).

(embedded)

```
source 2007 Bernstein--Lange
unified
compute U1 = X1 Z2
compute U2 = X2 Z1
compute S1 = Y1 Z2
compute S2 = Y2 Z1
compute ZZ = Z1 Z2
compute T = U1+U2
compute TT = T^2
compute M = S1+S2
compute R = TT-U1 U2+a ZZ^2
compute F = ZZ M
compute L = M F
compute LL = L^2
compute G = (T+L)^2-TT-LL
compute W = 2 R^2-G
compute X3 = 2 F W
compute Y3 = R(G-2 W)-2 LL
compute Z3 = 4 F F^2
```

Listing 4.3: Addition formula data in the projective coordinate system on the short-Weierstrass curve model (from the EFD [58]).

(embedded)

```
U1 = X1*Z2
U2 = X2*Z1
S1 = Y1*Z2
S2 = Y2*Z1
ZZ = Z1*Z2
T = U1+U2
TT = T^2
M = S1+S2
t0 = ZZ^2
t1 = a*t0
t2 = U1*U2
t3 = TT-t2
R = t3+t1
F = ZZ*M
L = M*F
LL = L^2
t4 = T+L
t5 = t4^2
t6 = t5-TT
G = t6-LL
t7 = R^2
t8 = 2*t7
W = t8-G
t9 = F*W
X3 = 2*t9
t10 = 2*W
t11 = G-t10
t12 = 2*LL
t13 = R*t11
Y3 = t13-t12
t14 = F^2
t15 = F*t14
Z3 = 4*t15
```

Listing 4.4: Three-operation-code addition formula data in the projective coordinate system on the short-Weierstrass curve model (from the EFD [58]).

Additional figures

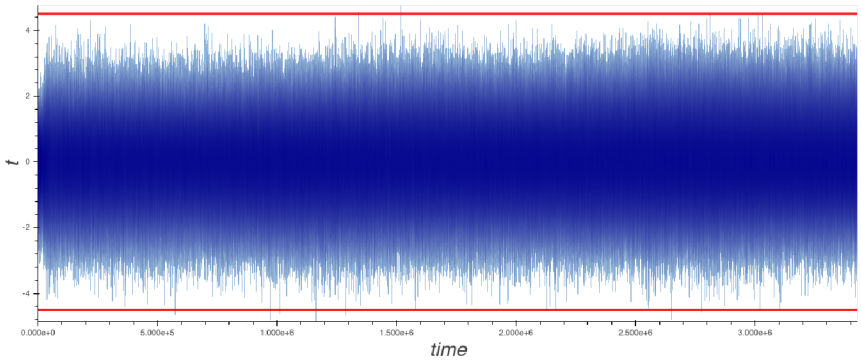


Figure 4.1: Values of the Welch's t-statistic for the `rcb` configuration hypothesis, computed on misaligned traces.

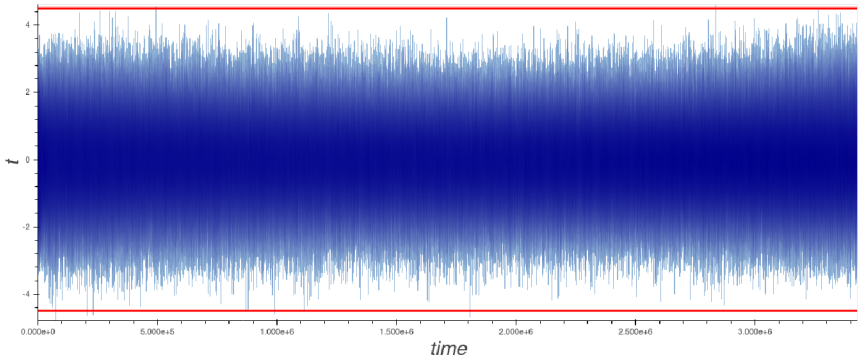


Figure 4.2: Values of the Welch's t-statistic for the a random partition of the traces, computed on misaligned traces.

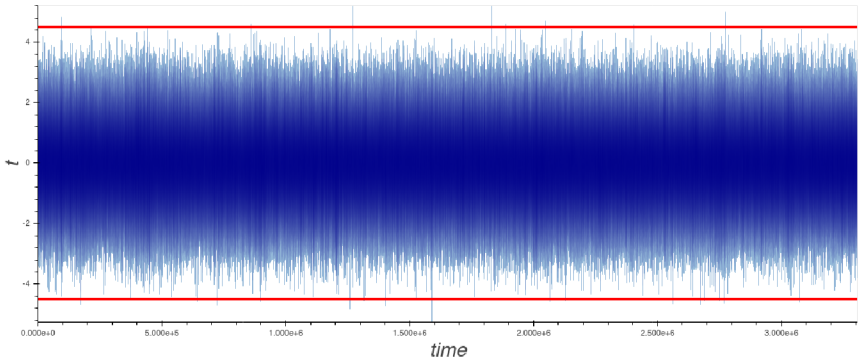


Figure 4.3: Values of the Welch's t-statistic for the `bl` configuration hypothesis, computed on aligned traces.

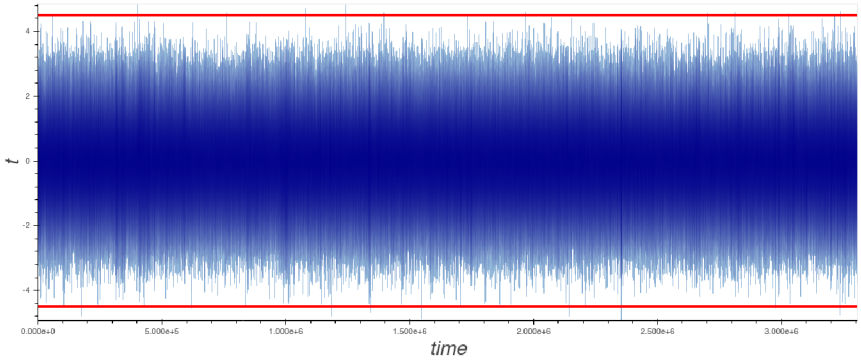


Figure 4.4: Values of the Welch's t-statistic for the `rcb` configuration hypothesis, computed on aligned traces.

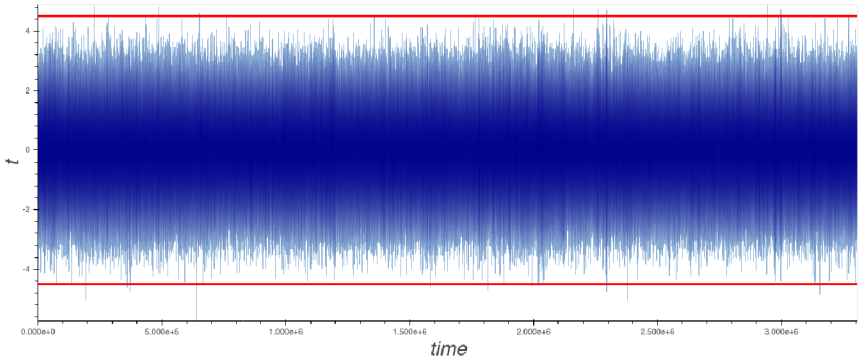


Figure 4.5: Values of the Welch's t-statistic for the a random partition of the traces, computed on aligned traces.

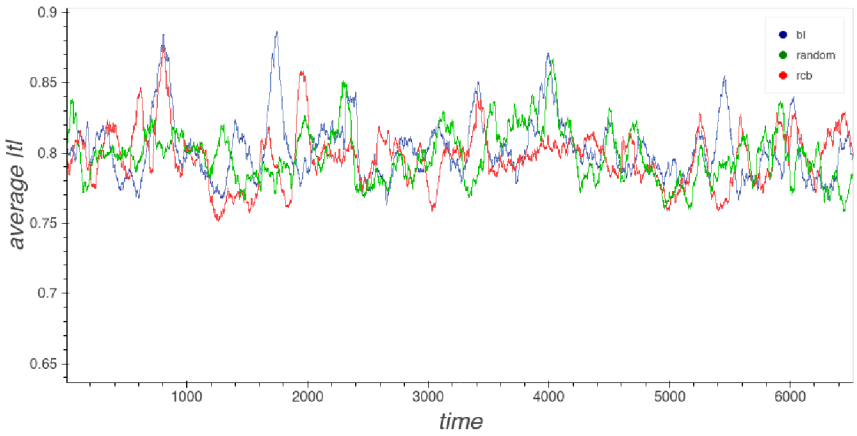


Figure 4.6: Downsampled (by averaging 5000 consecutive samples) absolute value of the Welch's t-statistic, computed on aligned traces, for the three partitions induced.

Bibliography

1. KOCHER, Paul C.; JAFFE, Joshua; JUN, Benjamin. Differential Power Analysis. In: WIENER, Michael J. (ed.). *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*. Springer, 1999, vol. 1666, pp. 388–397. Lecture Notes in Computer Science. Available from DOI: 10.1007/3-540-48405-1_25.
2. JANCAR, Jan; SEDLACEK, Vladimir; SVENDA, Petr; SYS, Marek. Minerva: The curse of ECDSA nonces; Systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces. In: *Conference on Cryptographic Hardware and Embedded Systems (CHES) 2020*. Ruhr-University of Bochum, Transactions on Cryptographic Hardware and Embedded Systems, 2020. ISSN 2569-2925. Available also from: https://minerva.crocs.fi.muni.cz.
3. RISCURE. *Inspector Side Channel Analysis* [online] [visited on 2020-02-26]. Available from: https://www.riscure.com/security-tools/inspector-sca/.
4. RAMBUS. *DPA Workstation Analysis Platform* [online] [visited on 2020-02-26]. Available from: https://www.rambus.com/security/dpa-countermeasures/dpa-workstation-platform/.
5. NEWAE TECHNOLOGY INC. *ChipWhisperer* [online] [visited on 2020-02-26]. Available from: https://chipwhisperer.com.
6. LEDGER DONJON. *lascar* [online] [visited on 2020-02-26]. Available from: https://github.com/Ledger-Donjon/lascar.
7. ESHARD. *scared* [online] [visited on 2020-02-26]. Available from: https://gitlab.com/eshard/scared.
8. RISCURE. *Jlsca* [online] [visited on 2020-02-26]. Available from: https://github.com/Riscure/Jlsca.
9. DPA CONTEST. *DPA contest website* [online] [visited on 2020-02-26]. Available from: http://www.dpacontest.org/home/.
10. PROUFF, Emmanuel; STRULLU, Rémi; BENADJILA, Ryad; CAGLI, Eleonora; DUMAS, Cécile. Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database. *IACR Cryptology ePrint Archive*. 2018, vol. 2018, p. 53. Available also from: http://eprint.iacr.org/2018/053.
11. WILES, Andrew. Modular Elliptic Curves and Fermat's Last Theorem. *Annals of Mathematics*. 1995, vol. 141, no. 3, pp. 443–551. ISSN 0003486X. Available from DOI: 10.2307/2118559.
12. COHEN, Henri; FREY, Gerhard; AVANZI, Roberto M.; DOCHE, Christophe; LANGE, Tanja; NGUYEN, Kim; VERCAUTEREN, Frederik. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2005-07-19. Discrete Mathematics and It's Applications, no. 34. ISBN 9781584885184.
13. HANKERSON, Darrel; MENEZES, Alfred J.; VANSTONE, Scott. *Guide to Elliptic Curve Cryptography*. New York, USA: Springer, 2004. ISBN 9780387218465. Available from DOI: 10.1007/b97644.
14. MONTGOMERY, Peter L. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*. 1987, vol. 48, no. 177, pp. 243–264. Available from DOI: 10.1090/S0025-5718-1987-0866113-7.
15. EDWARDS, Harold. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*. 2007, vol. 44, no. 3, pp. 393–422. Available from DOI: 10.1090/S0273-0979-07-01153-6.

16. BERNSTEIN, Daniel J.; BIRKNER, Peter; JOYE, Marc; LANGE, Tanja; PETERS, Christiane. *Twisted Edwards Curves* [Cryptology ePrint Archive, Report 2008/013]. 2008. Available also from: `http://eprint.iacr.org/2008/013`.
17. SMART, Nigel P. The Hessian Form of an Elliptic Curve. In: KOÇ, Çetin Kaya; NACCACHE, David; PAAR, Christof (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. Springer, 2001, vol. 2162, pp. 118–125. Lecture Notes in Computer Science. Available from DOI: `10.1007/3-540-44709-1_11`.
18. OKEYA, Katsuyuki; KURUMATANI, Hiroyuki; SAKURAI, Kouichi. Elliptic Curves with the Montgomery-Form and Their Cryptographic Applications. In: IMAI, Hideki; ZHENG, Yuliang (eds.). *Public Key Cryptography, Third International Workshop on Practice and Theory in Public Key Cryptography, PKC 2000, Melbourne, Victoria, Australia, January 18-20, 2000, Proceedings*. Springer, 2000, vol. 1751, pp. 238–257. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-540-46588-1\_17`.
19. COSTELLO, Craig; SMITH, Benjamin. Montgomery curves and their arithmetic - The case of large characteristic fields. *J. Cryptographic Engineering*. 2018, vol. 8, no. 3, pp. 227–240. Available from DOI: `10.1007/s13389-017-0157-6`.
20. BERNSTEIN, Daniel J; LANGE, Tanja. Faster Addition and Doubling on Elliptic Curves. In: *Proceedings of the 13th International Conference on the Theory and Application of Cryptology and Information Security*. Berlin, Heidelberg: Springer, 2007-12, pp. 29–50. ASIACRYPT '07. Available from DOI: `10.1007/978-3-540-76900-2_3`.
21. STRUIK, Rene. *Alternative Elliptic Curve Representations* [Internet Requests for Comments - Draft]. RFC Editor, 2020-04. RFC Draft. Available also from: `https://tools.ietf.org/html/draft-ietf-lwig-curve-representations-10`.
22. BERNSTEIN, Daniel J.; LANGE, Tanja. *SafeCurves: choosing safe curves for elliptic-curve cryptography* [online]. 2017-01 [visited on 2018-05-16]. Available from: `https://safecurves.cr.yp.to/`.
23. JANČÁR, Ján. *Security considerations for elliptic curve domain parameters selection* [online]. 2018 [visited on 2020-06-17]. Available from: `https://is.muni.cz/th/maq0p?lang=en`. Bachelor's thesis. Faculty of Informatics, Masaryk University.
24. BOS, Joppe W.; KAIHARA, Marcelo E.; KLEINJUNG, Thorsten; LENSTRA, Arjen K.; MONTGOMERY, Peter L. *PlayStation 3 computing breaks* 2^{60} *barrier 112-bit prime ECDLP solved* [online] [visited on 2020-03-14]. Available from: `https://www.epfl.ch/labs/lacal/articles/112bit_prime/`.
25. MRABET, Nadia El; JOYE, Marc. *Guide to Pairing-Based Cryptography*. 1st ed. Boca Raton, USA: CRC Press, 2016. ISBN 9781498729505.
26. DE FEO, Luca. *Exploring isogeny graphs* [online]. 2018 [visited on 2020-03-14]. Available from: `https://github.com/defeo/hdr/releases/download/defended/hdr.pdf`. Habilitation. Université de Versailles Saint-Quentin-en-Yvelines.
27. DIFFIE, Whitfield; HELLMAN, Martin E. New directions in cryptography. *IEEE Trans. Inf. Theory*. 1976, vol. 22, no. 6, pp. 644–654. Available from DOI: `10.1109/TIT.1976.1055638`.
28. *Special Publication 800-56A Revision 3: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised)*. 2018-04. Special Publication. National Institute for Standards and Technology. Available also from: `https://doi.org/10.6028/NIST.SP.800-56Ar3`.
29. *American National Standard X9.63-2001, Public key cryptography for the financial services industry: key agreement and key transport using elliptic curve cryptography*. 1999. Preliminary draft. Accredited Standards Committee X9.

30. *SEC 1: Elliptic Curve Cryptography*. 2009-05. Standard. Standards for Efficient Cryptography Group. Available also from: http://www.secg.org/sec1-v2.pdf.
31. *IEEE Standard Specifications for Public-Key Cryptography*. 2000. Standard. IEEE Std 1363-2000 Working Group.
32. *SEC 2: Recommended Elliptic Curve Domain Parameters*. 2010-01. Standard. Standards for Efficient Cryptography Group. Available also from: http://www.secg.org/sec2-v2.pdf.
33. *ECC Brainpool Standard Curves and Curve Generation*. 2005. Standard. ECC Brainpool. Available also from: http://www.ecc-brainpool.org/download/Domain-parameters.pdf.
34. BERNSTEIN, Daniel J. Curve25519: new Diffie-Hellman speed records. In: *Proceedings of the 9th International Conference on Theory and Practice in Public Key Cryptography*. Berlin, Heidelberg: Springer, 2006-04, pp. 207–228. PKC '06. Available from DOI: 10.1007/11745853_14.
35. LIM, Chae Hoon; LEE, Pil Joong. A Key Recovery Attack on Discrete Log-based Schemes Using a Prime Order Subgroupp. In: JR., Burton S. Kaliski (ed.). *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*. Springer, 1997, vol. 1294, pp. 249–263. Lecture Notes in Computer Science. Available from DOI: 10.1007/BFb0052240.
36. JAGER, Tibor; SCHWENK, Jörg; SOMOROVSKY, Juraj. Practical Invalid Curve Attacks on TLS-ECDH. In: PERNUL, Günther; RYAN, Peter Y. A.; WEIPPL, Edgar R. (eds.). *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*. Springer, 2015, vol. 9326, pp. 407–425. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-319-24174-6_21.
37. FOUQUE, Pierre-Alain; LERCIER, Reynald; RÉAL, Denis; VALETTE, Frédéric. Fault attack on elliptic curve Montgomery ladder implementation. In: *Proceedings of the 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 2008, pp. 92–98. FDTC'08. Available from DOI: 10.1109/FDTC.2008.15.
38. HAMBURG, Mike. *Ed448-Goldilocks, a new elliptic curve* [Cryptology ePrint Archive, Report 2015/625]. 2015. Available also from: http://eprint.iacr.org/2015/625.
39. LANGLEY, Adam; HAMBURG, Mike; TURNER, Sean. *RFC 7748: Elliptic Curves for Security* [Internet Requests for Comments]. RFC Editor, 2016-01. RFC, 7748. Available also from: https://tools.ietf.org/html/rfc7748.
40. VAUDENAY, Serge. The Security of DSA and ECDSA. In: DESMEDT, Yvo (ed.). *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*. Springer, 2003, vol. 2567, pp. 309–323. Lecture Notes in Computer Science. Available from DOI: 10.1007/3-540-36288-6_23.
41. *FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION 186-2 Digital Signature Standard (DSS)*. 2000-01. Standard. National Institute for Standards and Technology. Available also from: http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf.
42. *American National Standard X9.62-2005, Public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ECDSA)*. 2005. Standard. Accredited Standards Committee X9.
43. SCHNORR, Claus-Peter. Efficient Identification and Signatures for Smart Cards. In: *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*. 1989, pp. 239–252. Available from DOI: 10.1007/0-387-34805-0_22.

44. MENEZES, Alfred J.; VANSTONE, Scott A.; VAN OORSCHOT, Paul C. *Handbook of Applied Cryptography*. 1st ed. Boca Raton, USA: CRC Press, 1996-10. ISBN 9780849385230. Available also from: http://cacr.uwaterloo.ca/hac/.
45. FAILOVERFLOW. *Console Hacking 2010: PS3 Epic Fail* [online] [visited on 2019-09-12]. Available from: https://www.cs.cmu.edu/~dst/GeoHot/1780_27c3_console_hacking_2010.pdf.
46. CHROMIUM OS. *U2F ECDSA vulnerability - The Chromium Projects* [online] [visited on 2019-09-12]. Available from: https://sites.google.com/a/chromium.org/dev/chromium-os/u2f-ecdsa-vulnerability.
47. PORNIN, Thomas. *RFC 6979: Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)* [Internet Requests for Comments]. RFC Editor, 2013-08. RFC, 6979. Available also from: https://tools.ietf.org/html/rfc6979.
48. BERNSTEIN, Daniel J.; DUIF, Niels; LANGE, Tanja; SCHWABE, Peter; YANG, Bo-Yin. High-speed high-security signatures. *J. Cryptographic Engineering*. 2012, vol. 2, no. 2, pp. 77–89. Available from DOI: 10.1007/s13389-012-0027-1.
49. BERNSTEIN, Daniel J.; JOSEFSSON, Simon; LANGE, Tanja; SCHWABE, Peter; YANG, Bo-Yin. EdDSA for more curves. *IACR Cryptology ePrint Archive*. 2015, vol. 2015, p. 677. Available also from: http://eprint.iacr.org/2015/677.
50. JOSEFSSON, Simon; LIUSVAARA, Ilari. *RFC 8032: Edwards-Curve Digital Signature Algorithm (EdDSA)* [Internet Requests for Comments]. RFC Editor, 2017-01. RFC, 8032. Available also from: https://tools.ietf.org/html/rfc8032.
51. RENES, Joost; COSTELLO, Craig; BATINA, Lejla. Complete Addition Formulas for Prime Order Elliptic Curves. In: FISCHLIN, Marc; CORON, Jean-Sébastien (eds.). *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*. Springer, 2016, vol. 9665, pp. 403–428. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-662-49890-3_16.
52. OSWALD, David. *jc_curve25519* [online] [visited on 2020-06-20]. Available from: https://github.com/david-oswald/jc%7B%5C_%7Dcurve25519.
53. HAMBURG, Mike; VALENCE, Henry de; LOVECRUFT, Isis; ARCIERI, Tony. *The Ristretto Group* [online] [visited on 2020-06-20]. Available from: https://ristretto.group/.
54. HAMBURG, Mike. Decaf: Eliminating Cofactors Through Point Compression. In: GENNARO, Rosario; ROBSHAW, Matthew (eds.). *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*. Springer, 2015, vol. 9215, pp. 705–723. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-662-47989-6_34.
55. STRAUS, Ernst G. Problems and solutions: Addition chains of vectors. In: 1964.
56. PIPPENGER, Nicholas. On the Evaluation of Powers and Related Problems (Preliminary Version). In: *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*. IEEE Computer Society, 1976, pp. 258–263. Available from DOI: 10.1109/SFCS.1976.21.
57. BROWN, Michael; HANKERSON, Darrel; HERNANDEZ, Julio López; MENEZES, Alfred. Software Implementation of the NIST Elliptic Curves Over Prime Fields. In: NACCACHE, David (ed.). *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*. Springer, 2001, vol. 2020, pp. 250–265. Lecture Notes in Computer Science. Available from DOI: 10.1007/3-540-45353-9_19.

58. LANGE, Tanja; BERNSTEIN, Daniel J. *Explicit-Formulas Database* [online] [visited on 2020-04-25]. Available from: `https://www.hyperelliptic.org/EFD/`.
59. MESSERGES, Thomas S.; DABBISH, Ezzy A. Investigations of Power Analysis Attacks on Smartcards. In: GUTHERY, Scott B.; HONEYMAN, Peter (eds.). *Proceedings of the 1st Workshop on Smartcard Technology, Smartcard 1999, Chicago, Illinois, USA, May 10-11, 1999*. USENIX Association, 1999.
60. GENKIN, Daniel; PIPMAN, Itamar; TROMER, Eran. Get your hands off my laptop: physical side-channel key-extraction attacks on PCs - Extended version. *J. Cryptographic Engineering*. 2015, vol. 5, no. 2, pp. 95–112. Available from DOI: `10.1007/s13389-015-0100-7`.
61. GENKIN, Daniel; PACHMANOV, Lev; PIPMAN, Itamar; TROMER, Eran. Stealing Keys from PCs Using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation. In: GÜNEYSU, Tim; HANDSCHUH, Helena (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*. Springer, 2015, vol. 9293, pp. 207–228. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-662-48324-4_11`.
62. GENKIN, Daniel; PACHMANOV, Lev; PIPMAN, Itamar; TROMER, Eran. ECDH Key-Extraction via Low-Bandwidth Electromagnetic Attacks on PCs. In: SAKO, Kazue (ed.). *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*. Springer, 2016, vol. 9610, pp. 219–235. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-319-29485-8_13`.
63. GENKIN, Daniel; PACHMANOV, Lev; PIPMAN, Itamar; TROMER, Eran; YAROM, Yuval. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In: WEIPPL, Edgar R.; KATZENBEISSER, Stefan; KRUEGEL, Christopher; MYERS, Andrew C.; HALEVI, Shai (eds.). *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016, pp. 1626–1638. Available from DOI: `10.1145/2976749.2978353`.
64. LIPP, Moritz; SCHWARZ, Michael; GRUSS, Daniel; PRESCHER, Thomas; HAAS, Werner; MANGARD, Stefan; KOCHER, Paul; GENKIN, Daniel; YAROM, Yuval; HAMBURG, Mike. Meltdown. *meltdownattack.com*. 2018. Available also from: `https://meltdownattack.com/meltdown.pdf`.
65. RYAN, Keegan. Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone. In: CAVALLARO, Lorenzo; KINDER, Johannes; WANG, XiaoFeng; KATZ, Jonathan (eds.). *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 2019, pp. 181–194. Available from DOI: `10.1145/3319535.3354197`.
66. CARMON, Elad; SEIFERT, Jean-Pierre; WOOL, Avishai. Photonic side channel attacks against RSA. In: *2017 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2017, McLean, VA, USA, May 1-5, 2017*. IEEE Computer Society, 2017, pp. 74–78. Available from DOI: `10.1109/HST.2017.7951801`.
67. GENKIN, Daniel; SHAMIR, Adi; TROMER, Eran. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In: GARAY, Juan A.; GENNARO, Rosario (eds.). *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*. Springer, 2014, vol. 8616, pp. 444–461. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-662-44371-2_25`.
68. BAUER, Aurélie; JAULMES, Éliane; PROUFF, Emmanuel; REINHARD, Jean-René; WILD, Justine. Horizontal collision correlation attack on elliptic curves - - Extended

Version -. *Cryptography and Communications*. 2015, vol. 7, no. 1, pp. 91–119. Available from DOI: 10.1007/s12095-014-0111-8.

69. MANGARD, Stefan; OSWALD, Elisabeth; POPP, Thomas. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. New York, USA: Springer, 2007. ISBN 978-0-387-30857-9. Available from DOI: 10.1007/978-0-387-38162-6.
70. KOCHER, Paul C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: KOBLITZ, Neal (ed.). *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. Springer, 1996, vol. 1109, pp. 104–113. Lecture Notes in Computer Science. Available from DOI: 10.1007/3-540-68697-5_9.
71. BRUMLEY, David; BONEH, Dan. Remote Timing Attacks Are Practical. In: *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003. Available also from: https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical.
72. BLEICHENBACHER, Daniel. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In: KRAWCZYK, Hugo (ed.). *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*. Springer, 1998, vol. 1462, pp. 1–12. Lecture Notes in Computer Science. Available from DOI: 10.1007/BFb0055716.
73. BÖCK, Hanno; SOMOROVSKY, Juraj; YOUNG, Craig. Return Of Bleichenbacher's Oracle Threat (ROBOT). In: ENCK, William; FELT, Adrienne Porter (eds.). *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. USENIX Association, 2018, pp. 817–849. Available also from: https://www.usenix.org/conference/usenixsecurity18/presentation/bock.
74. BERNSTEIN, Daniel J. *Cache-timing attacks on AES* [online]. 2005 [visited on 2020-04-22]. Available from: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.
75. O'FLYNN, Colin; DEWAR, Alex. On-Device Power Analysis Across Hardware Security Domains. Stop Hitting Yourself. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, vol. 2019, no. 4, pp. 126–153. Available from DOI: 10.13154/tches.v2019.i4.126-153.
76. O'FLYNN, Colin; CHEN, Zhizhang. Synchronous sampling and clock recovery of internal oscillators for side channel analysis and fault injection. *J. Cryptographic Engineering*. 2015, vol. 5, no. 1, pp. 53–69. Available from DOI: 10.1007/s13389-014-0087-5.
77. AGRAWAL, Dakshi; ARCHAMBEAULT, Bruce; RAO, Josyula R.; ROHATGI, Pankaj. The EM Side-Channel(s). In: JR., Burton S. Kaliski; KOÇ, Çetin Kaya; PAAR, Christof (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. Springer, 2002, vol. 2523, pp. 29–45. Lecture Notes in Computer Science. Available from DOI: 10.1007/3-540-36400-5_4.
78. BRIER, Eric; CLAVIER, Christophe; OLIVIER, Francis. Correlation Power Analysis with a Leakage Model. In: JOYE, Marc; QUISQUATER, Jean-Jacques (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Springer, 2004, vol. 3156, pp. 16–29. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-540-28632-5_2.
79. GIERLICHS, Benedikt; BATINA, Lejla; TUYLS, Pim; PRENEEL, Bart. Mutual Information Analysis. In: OSWALD, Elisabeth; ROHATGI, Pankaj (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*. Springer, 2008, vol. 5154, pp. 426–442. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-540-85053-3_27.

80. CHARI, Suresh; RAO, Josyula R.; ROHATGI, Pankaj. Template Attacks. In: JR., Burton S. Kaliski; KOÇ, Çetin Kaya; PAAR, Christof (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. Springer, 2002, vol. 2523, pp. 13–28. Lecture Notes in Computer Science. Available from DOI: 10.1007/3-540-36400-5_3.
81. BATINA, Lejla; GIERLICHS, Benedikt; PROUFF, Emmanuel; RIVAIN, Matthieu; STANDAERT, François-Xavier; VEYRAT-CHARVILLON, Nicolas. Mutual Information Analysis: a Comprehensive Study. *J. Cryptology*. 2011, vol. 24, no. 2, pp. 269–291. Available from DOI: 10.1007/s00145-010-9084-8.
82. GOUBIN, Louis. A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems. In: DESMEDT, Yvo (ed.). *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*. Springer, 2003, vol. 2567, pp. 199–210. Lecture Notes in Computer Science. Available from DOI: 10.1007/3-540-36288-6_15.
83. AKISHITA, Toru; TAKAGI, Tsuyoshi. Zero-Value Point Attacks on Elliptic Curve Cryptosystem. In: BOYD, Colin; MAO, Wenbo (eds.). *Information Security, 6th International Conference, ISC 2003, Bristol, UK, October 1-3, 2003, Proceedings*. Springer, 2003, vol. 2851, pp. 218–233. Lecture Notes in Computer Science. Available from DOI: 10.1007/10958513_17.
84. IZU, Tetsuya; TAKAGI, Tsuyoshi. Exceptional Procedure Attack on Elliptic Curve Cryptosystems. In: DESMEDT, Yvo (ed.). *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*. Springer, 2003, vol. 2567, pp. 224–239. Lecture Notes in Computer Science. Available from DOI: 10.1007/3-540-36288-6_17.
85. MEDWED, Marcel; OSWALD, Elisabeth. Template Attacks on ECDSA. In: CHUNG, Kyo-Il; SOHN, Kiwook; YUNG, Moti (eds.). *Information Security Applications, 9th International Workshop, WISA 2008, Jeju Island, Korea, September 23-25, 2008, Revised Selected Papers*. Springer, 2008, vol. 5379, pp. 14–27. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-642-00306-6_2.
86. BATINA, Lejla; CHMIELEWSKI, Lukasz; PAPACHRISTODOULOU, Louiza; SCHWABE, Peter; TUNSTALL, Michael. Online template attacks. *J. Cryptographic Engineering*. 2019, vol. 9, no. 1, pp. 21–36. Available from DOI: 10.1007/s13389-017-0171-8.
87. SCHNEIDER, Tobias; MORADI, Amir. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In: GÜNEYSU, Tim; HANDSCHUH, Helena (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*. Springer, 2015, vol. 9293, pp. 495–513. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-662-48324-4_25.
88. FOUQUE, Pierre-Alain; VALETTE, Frédéric. The Doubling Attack - *Why Upwards Is Better than Downwards*. In: WALTER, Colin D.; KOÇ, Çetin Kaya; PAAR, Christof (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*. Springer, 2003, vol. 2779, pp. 269–280. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-540-45238-6_22.
89. YEN, Sung-Ming; KO, Lee-Chun; MOON, Sang-Jae; HA, JaeCheol. Relative Doubling Attack Against Montgomery Ladder. In: WON, Dongho; KIM, Seungjoo (eds.). *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*. Springer, 2005, vol. 3935, pp. 117–128. Lecture Notes in Computer Science. Available from DOI: 10.1007/11734727_11.

90. FAN, Junfeng; VERBAUWHEDE, Ingrid. An Updated Survey on Secure ECC Implementations: Attacks, Countermeasures and Cost. In: NACCACHE, David (ed.). *Cryptography and Security: From Theory to Applications - Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday*. Springer, 2012, vol. 6805, pp. 265–282. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-642-28368-0_18.
91. DANGER, Jean-Luc; GUILLEY, Sylvain; HOOGVORST, Philippe; MURDICA, Cédric; NACCACHE, David. A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards. *J. Cryptographic Engineering*. 2013, vol. 3, no. 4, pp. 241–265. Available from DOI: 10.1007/s13389-013-0062-6.
92. CORON, Jean-Sébastien. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In: KOÇ, Çetin Kaya; PAAR, Christof (eds.). *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*. Springer, 1999, vol. 1717, pp. 292–302. Lecture Notes in Computer Science. Available from DOI: 10.1007/3-540-48059-5_25.
93. JOYE, Marc; TYMEN, Christophe. Protections against Differential Analysis for Elliptic Curve Cryptography. In: KOÇ, Çetin Kaya; NACCACHE, David; PAAR, Christof (eds.). *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*. Springer, 2001, vol. 2162, pp. 377–390. Lecture Notes in Computer Science, no. Generators. Available from DOI: 10.1007/3-540-44709-1_31.
94. QUISQUATER, Jean-Jacques; SAMYDE, David. Automatic Code Recognition for Smart Cards Using a Kohonen Neural Network. In: *Proceedings of the 5th Conference on Smart Card Research and Advanced Application Conference - Volume 5*. San Jose, CA: USENIX Association, 2002, p. 6. CARDIS'02. Available from DOI: 10.5555/1250988.1250994.
95. EISENBARTH, Thomas; PAAR, Christof; WEGHENKEL, Björn. Building a Side Channel Based Disassembler. *Trans. Comput. Sci.* 2010, vol. 10, pp. 78–99. Available from DOI: 10.1007/978-3-642-17499-5_4.
96. MSGNA, Mehari; MARKANTONAKIS, Konstantinos; MAYES, Keith. Precise Instruction-Level Side Channel Profiling of Embedded Processors. In: HUANG, Xinyi; ZHOU, Jianying (eds.). *Information Security Practice and Experience - 10th International Conference, ISPEC 2014, Fuzhou, China, May 5-8, 2014. Proceedings*. Springer, 2014, vol. 8434, pp. 129–143. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-319-06320-1_11.
97. STROBEL, Daehyun; BACHE, Florian; OSWALD, David; SCHELLENBERG, Falk; PAAR, Christof. Scandalee: a side-channel-based disassembler using local electromagnetic emanations. In: NEBEL, Wolfgang; ATIENZA, David (eds.). *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*. ACM, 2015, pp. 139–144. Available also from: http://dl.acm.org/citation.cfm?id=2755784.
98. PARK, Jungmin; XU, Xiaolin; JIN, Yier; FORTE, Domenic; TEHRANIPOOR, Mark. Power-based side-channel instruction-level disassembler. In: *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. ACM, 2018, 119:1–119:6. Available from DOI: 10.1145/3195970.3196094.
99. VERMOEN, Dennis; WITTEMAN, Marc F.; GAYDADJIEV, Georgi. Reverse Engineering Java Card Applets Using Power Analysis. In: SAUVERON, Damien; MARKANTONAKIS, Constantinos; BILAS, Angelos; QUISQUATER, Jean-Jacques (eds.). *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems, First IFIP TC6 / WG 8.8 / WG 11.2 International Workshop, WISTP 2007, Heraklion, Crete, Greece, May 9-11, 2007, Proceedings*. Springer, 2007, vol. 4462, pp. 138–149. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-540-72354-7_12.

100. CLAVIER, Christophe. Side Channel Analysis for Reverse Engineering (SCARE) - An Improved Attack Against a Secret A3/A8 GSM Algorithm. *IACR Cryptology ePrint Archive*. 2004, vol. 2004, p. 49. Available also from: http://eprint.iacr.org/2004/049.
101. DAUDIGNY, Rémy; LEDIG, Hervé; MULLER, Frédéric; VALETTE, Frédéric. SCARE of the DES. In: IOANNIDIS, John; KEROMYTIS, Angelos D.; YUNG, Moti (eds.). *Applied Cryptography and Network Security, Third International Conference, ACNS 2005, New York, NY, USA, June 7-10, 2005, Proceedings*. 2005, vol. 3531, pp. 393–406. Lecture Notes in Computer Science. Available from DOI: 10.1007/11496137_27.
102. AMIEL, Frédéric; FEIX, Benoit; VILLEGAS, Karine. Power Analysis for Secret Recovering and Reverse Engineering of Public Key Algorithms. In: ADAMS, Carlisle M.; MIRI, Ali; WIENER, Michael J. (eds.). *Selected Areas in Cryptography, 14th International Workshop, SAC 2007, Ottawa, Canada, August 16-17, 2007, Revised Selected Papers*. Springer, 2007, vol. 4876, pp. 110–125. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-540-77360-3_8.
103. OLIPHANT, Travis E. *A guide to NumPy*. Trelgol Publishing USA, 2006.
104. VIRTANEN, Pauli; GOMMERS, Ralf; OLIPHANT, Travis E.; HABERLAND, Matt; REDDY, Tyler; COURNAPEAU, David; BUROVSKI, Evgeni; PETERSON, Pearu; WECKESSER, Warren; BRIGHT, Jonathan; VAN DER WALT, Stéfan J.; BRETT, Matthew; WILSON, Joshua; JARROD MILLMAN, K.; MAYOROV, Nikolay; NELSON, Andrew R. J.; JONES, Eric; KERN, Robert; LARSON, Eric; CAREY, CJ; POLAT, İlhan; FENG, Yu; MOORE, Eric W.; VAND ERPLAS, Jake; LAXALDE, Denis; PERKTOLD, Josef; CIMRMAN, Robert; HENRIKSEN, Ian; QUINTERO, E. A.; HARRIS, Charles R; ARCHIBALD, Anne M.; RIBEIRO, Antônio H.; PEDREGOSA, Fabian; VAN MULBREGT, Paul; CONTRIBUTORS. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*. 2020, vol. 17, pp. 261–272. Available from DOI: 10.1038/s41592-019-0686-2.
105. JUPYTER/IPYTHON PROJECT. *Jupyter* [online] [visited on 2020-04-25]. Available from: https://jupyter.org/index.html.
106. JANČÁR, Ján; SEDLÁČEK, Vladimír. *std: Standard curves database* [online] [visited on 2020-04-24]. Available from: https://neuromancer.sk/std/.
107. GATSBYJS. *GatsbyJS* [online] [visited on 2020-04-25]. Available from: https://www.gatsbyjs.org/.
108. THE SAGE DEVELOPERS. *SageMath, the Sage Mathematics Software System (Version 9.0)*. 2020. Available also from: https://www.sagemath.org.
109. THE PALLETS PROJECTS. *Jinja* [online] [visited on 2020-04-23]. Available from: https://jinja.palletsprojects.com/en/2.11.x/.
110. RIVERA-ZAMARRIPA, Luis; CHI-DOMÍNGUEZ, Jesús-Javier; BRUMLEY, Billy Bob. *ECCKiila* [online] [visited on 2020-07-03]. Available from: https://gitlab.com/nisec/ecckiila/.
111. DENIS, Tom St; CONTRIBUTORS. *libtommath* [online] [visited on 2020-04-26]. Available from: https://www.libtom.net/LibTomMath/.
112. PICOTECH. *picotech / picosdk-python-wrappers* [online] [visited on 2020-06-08]. Available from: https://github.com/picotech/picosdk-python-wrappers.
113. O'FLYNN, Colin. *colinoflynn / pico-python* [online] [visited on 2020-06-08]. Available from: https://github.com/colinoflynn/pico-python.
114. ROUSSEAU, Ludovic; CONTRIBUTORS. *pyscard* [online] [visited on 2020-06-13]. Available from: https://pyscard.sourceforge.io/.
115. ŠVENDA, Petr; JANČÁR, Ján. *ECTester* [online]. 2018 [visited on 2018-05-16]. Available from: https://github.com/crocs-muni/ECTester.

116. KRUSKAL, Joseph B; LIBERMAN, Mark; KRUSKAL, J. The symmetric time-warping problem: from continuous to discrete. 1999.
117. SALVADOR, Stan; CHAN, Philip. Toward accurate dynamic time warping in linear time and space. *Intell. Data Anal.* 2007, vol. 11, no. 5, pp. 561–580.
118. WOUDENBERG, Jasper G. J. van; WITTEMAN, Marc F.; BAKKER, Bram. Improving Differential Power Analysis by Elastic Alignment. In: KIAYIAS, Aggelos (ed.). *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings.* Springer, 2011, vol. 6558, pp. 104–119. Lecture Notes in Computer Science. Available from DOI: `10.1007/978-3-642-19074-2\_8`.
119. HUNTER, J. D. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering.* 2007, vol. 9, no. 3, pp. 90–95. Available from DOI: `10.1109/MCSE.2007.55`.
120. DATASHADER CONTRIBUTORS. *Datashader* [online] [visited on 2020-06-13]. Available from: `https://datashader.org`.
121. HOLOVIEWS CONTRIBUTORS. *Holoviews* [online] [visited on 2020-06-13]. Available from: `https://holoviews.org`.
122. BOKEH DEVELOPMENT TEAM. *Bokeh: Python library for interactive visualization.* 2020. Available also from: `https://bokeh.org/`.
123. THE HDF GROUP. *THE HDF5® LIBRARY & FILE FORMAT* [online] [visited on 2020-06-13]. Available from: `https://www.hdfgroup.org/solutions/hdf5`.
124. H5PY CONTRIBUTORS. *HDF5 for Python* [online] [visited on 2020-06-13]. Available from: `https://www.h5py.org/`.
125. SEDLACEK, Vladimir; JANCAR, Jan; SVENDA, Petr. Fooling primality tests on smartcards. In: *25th European Symposium on Research in Computer Security (ESORICS) 2020.* Springer, 2020. Available also from: `https://crocs.fi.muni.cz/papers/primality_esorics20`.
126. NEMEC, Matús; SÝS, Marek; SVENDA, Petr; KLINEC, Dusan; MATYAS, Vashek. The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli. In: THURAISINGHAM, Bhavani M.; EVANS, David; MALKIN, Tal; XU, Dongyan (eds.). *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.* ACM, 2017, pp. 1631–1648. Available from DOI: `10.1145/3133956.3133969`.
127. MOGHIMI, Daniel; SUNAR, Berk; EISENBARTH, Thomas; HENINGER, Nadia. TPM-FAIL: TPM meets Timing and Lattice Attacks. In: *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, 2020-08. Available also from: `https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-tpm`.
128. RIOJA, Unai; PAGUADA, Servio; BATINA, Lejla; ARMENDARIZ, Igor. *The uncertainty of Side-Channel Analysis: A way to leverage from heuristics.* 2020-06. Available from arXiv: `2006.12810 [cs.CR]`.