UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Corso di Laurea Magistrale in Informatica

Neural network attacks using side-channel leakages to break ECC implementations on consumer electronics

Relatore:
Prof. Danilo BRUSCHI
Correlatore:
Ing. Guido BERTONI
Controrelatore:
Prof. Vincenzo PIURI

Autore:
Michele Corrias
808746

Anno Accademico 2020/2021

Going in one more round when you don't think you can,
that's what makes all the difference in your life.
— R. B.

Dedicated to my mom, my sister, Olimpia and Rocky.

Contents

List of Figures

List of Tables

List of Algorithms and Listings

Chapter 1

Introduction

In the pervasive world we live in today, privacy and security are sensitive and important issues. The use of embedded systems, IoT devices and smart cards, becomes more and more ingrained and widespread in the evolution towards *Web 3.0* (the *Semantic Web*, a web of data). with all the advantages but also all the security risks that come with it. An attacker could take control over a service, system or network. He could simply intercept the victim's private data, but also steal it and even impersonate the victim himself.
During cryptographic operations performed by a device, physical information might be leaked as timing differences, power consumption or electromagnetic emissions in a specific time period. In the literature a well known set of attacks exploits these unintended leakages to obtain sensitive information (generally a cryptographic secret key). They are called *Side-Channel Attacks* (SCA), since *side channel* convey the idea that the channel used to transfer information is not the intended channel (i.e., plaintext, ciphertext), but a lateral (*side*) channel that was not supposed to exist by the developer.
Even if the cryptographic algorithm is proved to be secure in the classical theoretical sense, SCAs could work effectively on it, because the physical leakages are usually caused by the insecure algorithm implementation. The scientific community in recent years has developed countermeasures to defend against these types of attacks, but also on the other hand the development of new attack strategies and exploit methods has continued.
Recent advances in *Machine Learning* (ML) have made it possible to train a *Neural Network* (NN) to allow advanced attacks to be easier to perform and more effective. Thus, a wider spectrum of attacks can be reached.

The ultimate goal of this work is to provide all the requirements needed to mount a specific key-recovery attack (i.e., a *lattice* attack via *LLL* algorithm [FGR12; Ara+20; Ben+14; Roc+21]),

in order to retrieve a private key used by a consumer electronics device during a cryptographic computation. To achieve this goal, in this study we proceed to mount, with a *Long Short-Term Memory* (LSTM) network, a specific SCA based on power consumption leakage of a modern board, running an *Elliptic Curve Cryptography* (ECC) implementation.
We apply different techniques, which will be detailed in this document. We focus on a *STM32* microcontroller and on the *micro-ecc* software library, a widely used implementation of *Elliptic Curve Digital Signature Algorithm* (ECDSA). The acquired power consumption traces of the target device performing sensitive ECC operations will be used to develop a supervised training of the LSTM. This is in order to exploit the leakage model related to the ECDSA implementation considered. The LSTM performing the SCA allows to retrieve a few bits of the ephemeral key used in the ECDSA implementation on each digital signature. Such retrieved bits make it possible to mount a LLL attack to recover the ECC secret key in a few hundreds signatures.
Some of the most advanced works regarding the use of NNs in the SCA field are: [WPB19; Wei+20; WPP21; Per+21; Pic+21] (they will be explored in subsection 2.3.3). We introduce an innovating part in the application of the *Human Activity Recognition* methodology [Ang+13; OR16]), and of the LSTM, on off-the-shelf microcontrollers performing sensitive ECC operations. To the best of our knowledge, our approach constitutes the first application of using this technique in the state of the art.
This study continues the work started by the author of [Nav20] with his thesis, which stems from the work of [Rya19] presented in the conference on *Cryptographic Hardware and Embedded Systems* (CHES) of 2019. The study was intended to show how a NN can be trained to break the security of ECC when implemented on a modern microcontroller. Despite the results achieved, unfortunately there were some technical details and limitations to consider, that made it difficult in [Nav20] to mount a complete attack. In particular, two important variables were identified over which there was not the sufficient control:

- the real leakage model of the device
- the acquisition system of the power consumption traces

In addition to these things, another problem in [Nav20] was on the dataset organization, that was unbalanced and required more work to make it fit the need of the NN.
So, our approach in this study is to start from the results achieved in the previous work and to reduce the number of variables, monitoring and managing the leakage model and the acquisition system with a purpose-built simulation environment. Thus, we aim to focus only on the NN, updating and evaluating it. Once we have obtained perfect results with the NN in the simulation phase, we can move on and use it in the real use case, where there are more noise and difficulties to be tackled.

With the improvements obtained in this work, we are able to achieve a NN accuracy of more than 97%. These results allow to use the trained LSTM to find a few bits of the ephemeral key on each digital signature. Thus, an attacker could mount a LLL attack and recover the ECC private key in a few hundreds signatures collected.

The remainder of the document is organised as follows. In chapter 2, we present the general background needed to understand this work, with particular attention to ECC (section 2.1), NNs (section 2.2), SCAs (section 2.3) and a few countermeasures section 2.4. In chapter 3, we discuss some problems addressed by the previous work [Nav20]. We also present our methodology, which consists of a simulation environment used to better manage the dataset organization and to improve the design of our LSTM in charge of mounting a specific SCA. In chapter 4, we exploit all the information learned in the simulation phase to propose an application in a real use case. We train the LSTM on the real power consumption traces, and we use it to mount a SCA based on power consumption leakage of an off-the-shelf microcontroller running a widely used ECC implementation. In chapter 5, we present the successful results obtained from our NN on the real traces of the target device. We also provide different metrics to evaluate the results achieved. Finally in chapter 6, we summarize the conclusions of this work, we discuss the impact of our solution and we suggest the future works.

Chapter 2

Background

In the following chapter, we present the main background topics of this work. In particular, ECC section 2.1), NNs (section 2.2), SCAs (section 2.3) and a few of their countermeasures section 2.4. The elliptic curves, presented in the following section, are a key component for many widely-used standard asymmetric cryptographic algorithms, of which there are several implementations for embedded systems. Thus, the security of these devices is paramount to user security.

2.1 Elliptic Curve Cryptosystems

Elliptic Curve Cryptography (ECC) is an approach to public-key cryptography built on the algebraic structure of elliptic curves over finite fields.
An elliptic curve is a plane non-singular cubic curve [BHW11], that means with non-singular points and of third order. An elliptic curve $\mathcal{E}$ over a field $\mathbb{K}$ is defined by the following equation [HMV06]:

$$\mathcal{E} : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \qquad a_1, a_2, a_3, a_4, a_6 \in \mathbb{K} \tag{2.1}$$

where the discriminant $\Delta \neq 0$ is defined as follows:

$$\left.\begin{array}{l} \Delta = -d_2^2d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\ d_2 = a_1^2 + 4a_2 \\ d_4 = 2a_4 + a_1a_3 \\ d_6 = a_3^2 + 4a_6 \\ d_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \end{array}\right\} \tag{2.2}$$

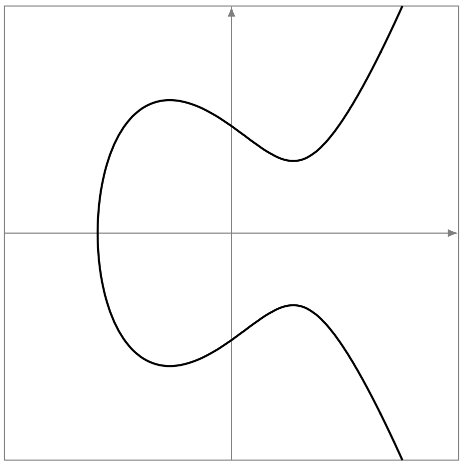


Figure 2.1: Weierstrass elliptic curve in $\mathbb{R}$

As defined by Koblitz [Kob87], an elliptic curve $\mathcal{E}_{\mathbb{K}}$ defined over a field $\mathbb{K}$ of characteristic $\neq 2, 3$ is the set of solutions $(x, y) \in \mathbb{K}^2$ to the Equation 2.3:

$$\mathcal{E} : y^2 = x^3 + ax + b \qquad a, b \in \mathbb{K} \tag{2.3}$$

This is the *Weierstrass* form of an elliptic curve, as depicted in Figure 2.1.
An elliptic curve $\mathcal{E}_{\mathbb{K}}$ over a field $\mathbb{K}$ is said to be non-singular if in every point on the curve at least one partial derivative of the curve is non-zero. If no singular points exists on $\mathcal{E}_{\mathbb{K}}$, then it's said to be a non-singular curve. Geometrically the non-singularity property of the elliptic curve means that there is a unique tangent line to the elliptic curve at every point $P \in \mathcal{E}_{\mathbb{K}}$ [BHW11].
The value of the discriminant of the curve Δ, when different from zero, ensures the non-singularity property and in the Weierstrass form is [HMV06]:

$$\Delta = 4a^3 + 27b^2 \neq 0 \tag{2.4}$$

2.1.1 Group Law

The set of points on an elliptic curve, augmented with a particular point $\mathcal{O}$ called *point at infinity*, forms an *additive abelian group* [DP97].
Let $\mathcal{E}_{\mathbb{K}} : y^2 = x^3 + ax + b$ defined over $\mathbb{K}$ where $(a, b) \in \mathbb{K}$:

$$\mathcal{E}(\mathbb{K}) = \{a, b \in \mathcal{E}_{\mathbb{K}} : (a, b) \in \mathbb{K}^2\} \cup \{\mathcal{O}\} \tag{2.5}$$

is an abelian group, which means it is commutative one. The point at infinity is the symbolic point at which all lines having same direction intersect. So, geometrically, it can be thought as the intersection of two parallel lines. In particular in Equation 2.5, $\mathcal{O}$ is the point at infinity of the y-axis, so, for instance, any line through $\mathcal{O}$ and a point of the curve will be the vertical line in that point.
So, let $\mathcal{E}_{\mathbb{K}}$ be an elliptic curve defined over the field $\mathbb{K}$, then we consider $P, Q, \mathcal{O}$ elements of $\mathcal{E}(\mathbb{K})$, where $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ are two points of the curve, while $\mathcal{O}$ is the point at infinity. We define $-P$ as the point with the same x-coordinate but negative y-coordinate of P. The following properties are true:

- $\mathcal{O} + \mathcal{O} = \mathcal{O}$
- $\mathcal{O} = -\mathcal{O}$
- $P + \mathcal{O} = \mathcal{O} + P = P \qquad \forall P \in \mathcal{E}(\mathbb{K})$
- $P + (-P) = -P + P = \mathcal{O}$
- $P + Q = (x_3, y_3)$

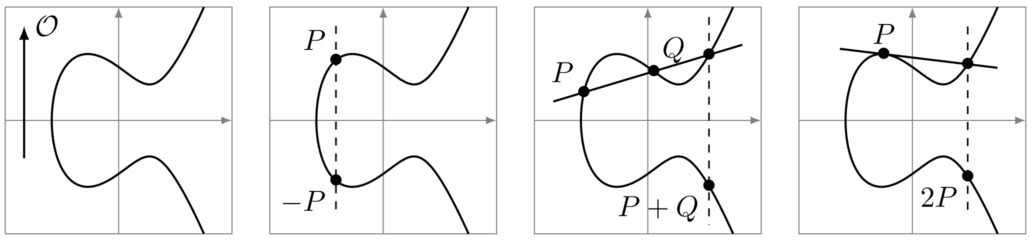
The last property is called the *addition law* [TW05]:

$$\begin{aligned}
x_3 &= \lambda^2 - x_1 - x_2 \\
y_3 &= \lambda(x_1 - x_3) - y_1 \\
\text{where } \lambda &= \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & \text{if } P \neq Q \\ \dfrac{3x_1^2 + a}{2y_1} & \text{if } P = Q : \text{here the addition is reduced to a doubling} \end{cases}
\end{aligned} \tag{2.6}$$

It can be shown that the addition law is associative and also commutative [TW05]:

$$\begin{aligned} (P+Q)+R &= P+(Q+R) \\ P+Q &= Q+P \end{aligned} \qquad \forall P, Q, R \in \mathcal{E}(\mathbb{K}) \tag{2.7}$$

The elliptic curve group operations described above are depicted in Figure 2.2.
The group defined by an elliptic curve over a finite field is finite [Kob87]. This is one of the reasons why the elliptic curves considered in this work, and more generally in cryptography, are defined over finite fields. Working in a finite field also provides some useful properties for cryptography. So in this work, from now on, we consider elliptic curves defined over finite fields $\mathbb{F}_p$, with p prime number.



Neutral element $\mathcal{O}$ Inverse element $-P$ Addition $P+Q$ Doubling $P+P$

Figure 2.2: Elliptic Curve Group Operations

The order of a finite field is its number of elements. Thus, the number of points in an elliptic curve group $\mathcal{E}(\mathbb{F}_p)$, denoted as $\#\mathcal{E}(\mathbb{F}_p)$, is defined as the *order of the curve* [LD00].

2.1.2 Affine vs Projective Coordinates

There are different methods to represent a point in an elliptic curve: one is that of *affine* coordinates, shown in Equation 2.6, in two dimensions. When two points are represented in affine coordinates, adding them requires an expensive field inversion, which is much slower than multiplication [Hit+02].
Another method of representing a point on an elliptic curve is by *projective* coordinates. This coordinates system is faster because it is possible to eliminate the inversion cost for the intermediate point additions in a scalar multiplication [Riv11]. In the projective coordinates system, there are different possible representations of a point P on an elliptic curve, as by a triplet (X, Y, Z), such that:

$$P = (x, y) \in \mathcal{E}(\mathbb{F}_p) \quad \Longrightarrow \quad P = (X, Y, Z) \mid (X/Z^c, Y/Z^d) = (x, y) \qquad c, d \in \mathbb{Z} \tag{2.8}$$

A point has several projective coordinates, as many as different Z. The equivalence class containing (X, Y, Z) and all the other projective representations of the affine point $(X/Z^c, Y/Z^d)$ is denoted $(X : Y : Z)$.
The most employed projective coordinates are the *Jacobian* ones, for which $c = 2$ and $d = 3$: they allow for fast point doubling, which is usually the most frequent operation in a scalar multiplication algorithm [Riv11].

2.1.3 Scalar Multiplication

The main operation of cryptographic implementations based on ECC is the *elliptic scalar multiplication* [LD00]. Let k be an integer and P a point of an elliptic curve, the scalar multiplication

kP is the result of adding P to itself k times.

$$kP = \underbrace{P + P + \cdots + P}_{k \text{ TIMES}} \tag{2.9}$$

The scalar multiplication is the core operation of elliptic curve cryptosystems, in which the scalar is usually secret [Riv11].
Since the multiplication can be realized as repeated point addition of the point with itself, scalar multiplication on elliptic curves, thus described, may seem intuitive because similar to basic algebra. But there are some technical details experts must include in their practical implementation, which will be further explored in section 2.4.
Having defined scalar multiplication, the definition of *order of a point* P on an elliptic curve is the smallest positive integer n such that $nP = \mathcal{O}$ [LD00].

2.1.4 Elliptic Curve Discrete Logarithm Problem

The security of all ECC algorithms is based on the complexity of the *Elliptic Curve Discrete Logarithm Problem* (ECDLP) [HMV06].
Given an elliptic curve $\mathcal{E}_{\mathbb{F}_p}$ defined over a finite field $\mathbb{F}_p$ and two points G and Q of the curve, ECDLP is about finding a positive integer k, if there exists, such that:

$$Q = kG \tag{2.10}$$

where kG is the result of scalar multiplication between k and G. Given the curve and the points, the challenge of solving the equation, and then finding the scalar k, is a difficult computational problem [LS08].

According to the literature, the best algorithm for solving ECDLP requires exponential time. Conversely, the best algorithms known for solving other important hard mathematical problems as the *integer factorization* problem (on which *Rivest–Shamir–Adleman* (RSA) [RSA19; Riv+83] is based) and the *discrete logarithm* problem (on which *Digital Signature Algorithm* (DSA) [KG13; Nis92] is based) require sub-exponential computational cost. As a result, ECC can use smaller parameters than other public-key cryptosystems such as RSA and DSA, while still providing equivalent levels of security. This is a reason that makes ECC advantageous because it allows requirement reductions in computing power, storage space, transmission bandwith and energy consumption [Age16; HMV06; LD00; Mag16; LKT13]. In Table 2.1, we summarize the key lengths comparison of different cryptosystems with a comparable security level, according to *National Institute of Standards and Technology* (NIST) [Gir20].

Table 2.1: The minimum key size requirement for security by NIST recommendations (2020, [Gir20]). All key lengths are provided in bits.
Triple Data Encryption Algorithm (TDEA, [BM17; Dwo])
Advanced Encryption Standard (AES, [DR99; Dwo+01])
Secure Hash Algorithms (SHA: SHA-1 [Sta95], SHA-2 [PW08; Dan15], SHA-3 [Ber+13; Dwo15])
Hash(A): Digital signatures and other applications requiring collision resistance
Hash(B): *Hash-based Message Authentication Code* (HMAC, [BCK96; Tur08]), *KECCAK Message Authentication Code* (KMAC, [KCP16]), key derivation functions and random bit generation

Date	*Security Strength*	*Symmetric Algorithms*	*Factoring Modulus*	*Discrete Logarithm Key*	*Discrete Logarithm Group*	*ECC*	*Hash(A)*	*Hash(B)*
Legacy	80	2TDEA	1024	160	1024	160	SHA-1	
2019 - 2030	112	3TDEA AES-128	2048	224	2048	224	SHA-224 SHA-512/224 SHA3-224	
2019 - 2030 & beyond	128	AES-128	3072	256	3072	256	SHA-256 SHA-512/256 SHA3-256	SHA-1 KMAC128
2019 - 2030 & beyond	192	AES-192	7680	384	7680	384	SHA-384 SHA3-384	SHA-224 SHA-512/224 SHA3-224
2019 - 2030 & beyond	256	AES-256	15360	512	15360	512	SHA-512 SHA3-512	SHA-256 SHA-512/256 SHA-384 SHA-512 SHA3-256 SHA3-384 SHA3-512 KMAC256

There are finite fields more suitable for security features, randomness, speed than others: in 1999 NIST recommended fifteen elliptic curves. Specifically it has recommended five finite fields $\mathbb{F}_p$ for certain primes p of sizes 192, 224, 256, 384, and 521 bits (for each of the prime fields, one elliptic curve is recommended) [KSD13].

2.1.5 ECC Algorithms

ECC has three basic applications, which will be explained below: key establishment, encryption and digital signature.

Elliptic Curve Diffie-Hellman

The *Elliptic Curve Diffie-Hellman* (ECDH) [Bar+18; Law+03] is a key exchange protocol. This is the elliptic curve counterpart of the *Diffie-Hellman* (DH) [DH76] key exchange that allows

two parties, without prior knowledge of each other, to establish a shared secret over an insecure channel [LD00]. ECDH is different from the general DH protocol, because it is based on ECDLP rather than the *Discrete Logarithm Problem* (DLP) [McC90; HL15]. ECDH is a key agreement protocol because all participating parties contribute information which is used to derive the shared secret key [HMV06], where each of the entities has an elliptic curve public-private key pair.
Let Alice and Bob be the parties involved, ECDH is presented in Algorithm 2.1:

Algorithm 2.1: ECDH

```
input: $\mathcal{E}_{\mathbb{F}_p}$ over a finite field $\mathbb{F}_p$, $G$ base point of order $n : nG = \mathcal{O}$
output: $S$
begin
    Alice selects $d_A$ random integer
    Bob selects $d_B$ random integer
    Alice sends $Q_A \leftarrow d_A G$ to Bob
    Bob sends $Q_B \leftarrow d_B G$ to Alice
    Alice computes $S \leftarrow d_A Q_B$
    Bob computes $S \leftarrow d_B Q_A$
end
```

Note that, both Alice and Bob had to agree on domain parameters. Each party involved has its own private key d, to be kept secret, and its own public key Q, made public.
The shared secret key S computed by both parties is equal because:

$$S = d_A Q_B = d_A d_B G = d_B d_A G = d_B Q_A = S \tag{2.11}$$

Because the keys exchange is performed over an insecure channel, an eavesdropper (Eve) could listen on it. Although Eve can listen to all the data passing through the channel, from such data it is not possible to retrieve Alice and Bob's secrets or the shared secret.
ECDH is hard to break because can be reduced to solving ECDLP [BL96; HL15].

Elliptic Curve Digital Signature Algorithm

Digital signatures aim to provide the digital counterpart to handwritten signatures and stamped seals. A digital signature depends on the signer's private key and, additionally, on the message being signed; signatures must be verifiable [MOP].
A digital signature scheme should provide the following basic cryptographic properties [JMV01; NRR17; ZL99]:

- *data integrity* (i.e., ensuring information has not been altered by unauthorized or unknown means [MOP])

- *authentication* (i.e., corroborating the source of information, also known as *data origin authentication* [MOP])
- *non repudiation* (i.e., preventing the denial of previous commitments or actions [MOP])

The *Elliptic Curve Digital Signature Algorithm* (ECDSA) [JMV01; KG13; Van92] is an algorithm for digital signatures: this is the elliptic curve counterpart of the *Digital Signature Algorithm* (DSA) [KG13; Nis92]). The private key is used to compute a signature, which is sent together with the message, while the public key is made known. The receiver can, then, use the public key to verify the validity of the signature, thus confirming that the message was sent by the sender.
ECDSA belongs to the asymmetric digital signatures schemes with appendix [JMV01]:

- asymmetric because each entity chooses a public - private key pair. The entity maintains the private key secret, which is used for signing messages, and makes its public key, that will be used to verify signatures, known and accessible to third parties
- appendix because the message digest is generated using a cryptographic hash function, then the signature is applied to the digest rather than the message itself

Let Alice and Bob the parties involved, ECDSA is presented in Algorithm 2.2:

Algorithm 2.2: ECDSA

```
input: $\mathcal{E}_{\mathbb{F}_p}$ over a finite field $\mathbb{F}_p$, $G$ base point $\in \mathcal{E}(\mathbb{F}_p)$ of order $n : nG = \mathcal{O}$, $m$ message, $H$ hash function
output: $(r, s)$ signature of $m$
begin
    Key Generation (by both):
        select $d \in [1, n-1]$ random integer
        $Q \leftarrow dG$
    Signing (by Alice):
        select $k \in [1, n-1]$ random integer
        $X = (x_1, y_1) \leftarrow kG$
        $r \leftarrow x_1 \mod n$
        if $r = 0$ then
            goto 8
        $e \leftarrow H(m)$
        $s \leftarrow k^{-1}(e + dr) \mod n$
        if $s = 0$ then
            goto 8
    Signature verification (by Bob):
        verify $r, s \in [1, n-1]$
        $e \leftarrow H(m)$
        $w \leftarrow s^{-1} \mod n$
        $u_1 \leftarrow ew \mod n$
        $u_2 \leftarrow rw \mod n$
        $X = (x_1, y_1) \leftarrow u_1G + u_2Q$
        if $X = \mathcal{O}$ then
            return invalid signature
        $v \leftarrow x_1 \mod n$
        if $v = r$ then
            return valid signature
end
```

Note that, both the parties had to agree on domain parameters: the elliptic curve and the hash function to be used on the message. The signer needs its own private key d, to be kept secret, and its own public key Q.
Given (r, s) the signature of the message m, the proof that signature verification works is the following:

$$\begin{gathered} s = k^{-1}(e + dr) \mod n \\ k \equiv s^{-1}(e + dr) \equiv s^{-1}e + s^{-1}rd \equiv we + wrd \equiv u_1 + u_2d \mod n \\ u_1G + u_2Q = (u_1 + u_2d)G = kG \mod n \longrightarrow v = r \end{gathered} \tag{2.12}$$

As with ECDH (subsubsection 2.1.5), relying on the hardness of ECDLP also applies to ECDSA [JM97; Ara+20]. But some attacks on the implementation of this algorithm, known in literature, prove this assumption is not always correct.
Under the assumption that implicit information on the *nonce* (ephemeral key) k is known, supposing the nonces share a certain amount of bits without knowing the value of the shared bits, an attack of polynomial complexity exists in the literature [FGR12]. Then, a more recent attack proves that it is possible to break ECDSA with less-than-one bit of nonce leakage [Ara+20]. These and other attacks on ECDSA implementation will be explored in subsubsection 2.3.3.

Elliptic Curve Integrated Encryption Scheme

The *Elliptic Curve Integrated Encryption Scheme* (ECIES) [GHS10; Bro09; 04] is a public-key encryption scheme based on ECC. It is a variant of the *ElGamal* scheme [ABR99; ABR01]. There are lightly different implementations of ECIES, depending on the standard version [ME+10]. The algorithm uses the following functions [GHS10]:

- *key agreement* (KA), used for the generation of a shared secret by two parties
- *key derivation function* (KDF), produces a set of cryptographic keys from a shared secret
- *encryption* (ENC), symmetric encryption algorithm
- *message authentication code* (MAC), data used to authenticate messages

- *hash* (H) digest function, used within the KDF and the MAC functions

Let Alice and Bob be the parties involved, ECIES is presented in Algorithm 2.3:

Algorithm 2.3: ECIES

```
input: $\mathcal{E}_{\mathbb{F}_p}$ over a finite field $\mathbb{F}_p$, $G$ base point $\in \mathcal{E}(\mathbb{F}_p)$ of order $n : nG = \mathcal{O}$, $m$ message, $KA$, $KDF$, $ENC$, $MAC$,
    $H$, optional shared information $si_1, si_2$
output: $R \parallel tag \parallel c$
begin
    Key Generation (by both):
        select $d \in [1, n-1]$ random integer
        $Q \leftarrow dG$
    Encrypt (by Alice):
        select $r \in [1, n-1]$ random integer
        $R \leftarrow rG$
        $S = (x_S, y_S) \leftarrow KA(rQ_B)$
        if $S = \mathcal{O}$ then
            fail
        $k_{ENC} \parallel k_{MAC} \leftarrow KDF(x_S \parallel si_1)$
        $c \leftarrow ENC(k_{ENC}, m)$
        $tag \leftarrow MAC(k_{MAC}, c \parallel si_2)$
        send $(R \parallel tag \parallel c)$
    Decrypt (by Bob):
        receive $(R \parallel tag \parallel c)$
        $S = (x_S, y_S) \leftarrow KA(d_B R)$
        if $S = \mathcal{O}$ then
            fail
        $k_{ENC} \parallel k_{MAC} \leftarrow KDF(x_S \parallel si_1)$
        $tag^* \leftarrow MAC(k_{MAC}, c \parallel si_2)$
        if $tag \neq tag^*$ then
            fail and exit
        $m \leftarrow ENC^{-1}(k_{ENC}, c)$
end
```

Note that both Alice and Bob had to agree on domain parameters: the elliptic curve, the optional shared information and the functions to be used (about these ones, there are details in [Bar+18]. Each party involved has its own private key d, to be kept secret, and its own public key Q, made known.

Given x_S the shared secret, the proof that it is the same for both Bob and Alice is:

$$S = d_B R = d_B r G = r d_B G = r Q_B \tag{2.13}$$

2.2 Neural Networks

In this section, we introduce a generic overview of the artificial intelligence (subsection 2.2.1) and machine learning (subsection 2.2.2) fields. Then we focus on neural networks (subsection 2.2.3), with special attention to that of our interest (subsection 2.2.5).

2.2.1 Artificial Intelligence

Thinking machines have long been the dream of scientists and engineers. Even before programmable computers were built, there were questions about whether they might become intelligent. The beginning of the *Artificial Intelligence* (AI) debate is owed to Alan Turing [Tur09] in 1950: he wondered if machines could think. He proposed the *Turing test*, where a human examiner tries to distinguish a human textual answer from a computer one.
John McCarthy defines AI as:

> *the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.* [McC07]

The AI field today has a wide range of practical applications and active research subjects: automatic speech recognition, customer service with online virtual agents and computer vision are just a few of these. Computers and machines are being used to simulate the human mind's problem-solving and decision-making abilities. It is well-known and iconic the event in which *International Business Machines* (IBM)'s *Deep Blue* chess-playing system defeated the world chess champion Garry Kasparov in a chess match in 1997: the victory is considered a milestone in the AI history [CHH02].
Since the beginning, AI quickly approached to find a solution to those problems described by a list of strict rules, challenging for humans but relatively easy for computers. The AI aims to solve simple-to-perform tasks, but hard-to-describe-formally ones, which for example are issues solved intuitively by human beings, such as recognizing faces or spoken words [GBC16].

2.2.2 Machine Learning

Machine Learning (ML) is a large branch of the AI scientific area. A ML algorithm is able to learn and acquire its own knowledge by extracting patterns from raw data. ML is a form of applied statistics focused on the use of computers to statistically estimate complex functions rather than proving confidence intervals around these functions [GBC16]. Most ML algorithms have

settings, known as *hyperparameters*, which must be determined out of the learning algorithm itself.
ML allowed computers to solve problems concerning real-world knowledge and make decisions that appear subjective [GBC16]: for example, recommending a cesarean section [Kha+20], biometrically recognizing faces [Ora14], detecting spam in incoming e-mails [Cra+15], or implementing intrusion detection systems [BG15]. The ML algorithms performance is highly dependent on the representation of the provided data. Each portion of information included in the data representation is called a *feature*.
According to Tom Mitchell:

> *the field of ML is concerned with the question of how to construct computer programs that automatically improve with experience.* [Mit+97]

A widely used operational definition of ML is the training of a model from data that generalizes a decision against a performance measure [Bro22]. A computer program learns from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E [Mit+97].

Task

A *task* in ML is typically described by considering how the system should process an *example*, that is, a set of features measured quantitatively by an object or event that the system itself must process. [GBC16]. Note that, the learning process itself is not the task: through learning, a ML algorithm gains the required ability to complete the task. Thus, tasks are the decisions that a ML algorithm needs to take.
ML can be used to solve a wide range of tasks. The following ones are just a few examples: *classification*, in which the model must establish which of k categories the input belongs to (e.g., object recognition), *regression*, in which the model must predict a numerical value, given some input (e.g., trading and financial predictions), ...This is just a summary, however to see more about these or other types of tasks, refer to [GBC16].

Experience

The *experience* leads the algorithm to improvement by learning, and is strictly dependent on what data to collect. Most of the ML algorithms experience an entire *dataset*, a collection of many examples [GBC16]. Almost all ML algorithms can be generally divided in two approaches: *unsupervised* or *supervised*, based on the experience type during the learning process.

Unsupervised Learning Unsupervised learning algorithms experience a dataset containing a lot of features and learn relevant information about the dataset structure [GBC16]. Unsupervised learning examines and clusters unlabeled datasets, finding data groupings or hidden patterns without requiring human interaction: hence, *unsupervised*. Common tasks of unsupervised approach are clustering, association and dimensionality reduction. Some applications where it is widely used are, for instance, computer vision, anomaly detection, and recommendation engines. Unsupervised learning will be not considered in this work.

Supervised Learning Supervised learning algorithms experience a dataset containing features, but each example is also associated with a label. In the literature, this is also known as *labeling* [GBC16]. The labeled datasets, containing inputs and correct outputs, are intended to guide the algorithm through the process of accurately classifying data or forecasting output: hence, *supervised*. The model can test its accuracy and is allowed to learn over time, thanks to labeled inputs and outputs. Common tasks of supervised approach are regression, classification and structured output problems. Some applications where it is widely used are, for example, image and object recognition, spam detection and predictive analytics.

Classification Classification [Fis36; Fis38] is a predictive modeling task, in which a given input data is associated with a class label. An algorithm which implements classification is known as a *classifier*. Some typical classification problems, as already mentioned, are spam detection, object recognition and face recognition [GBC16].
Binary classification includes those classification tasks with two classes. *Multi-class classification*, on the other hand, involves those classification tasks with more than two classes. There are mainly two types of multi-class classification techniques, which allow using binary classification algorithms for multi-class classification:

- *One-vs-Rest* (OvR), a method that involves splitting the multi-class dataset into multiple binary classification problems. A single binary classifier is trained per class, with the samples of that class as positive samples and all other samples as negatives [BN06]

- *One-vs-One* (OvO), another method which consists of splitting the multi-class dataset into multiple binary classification problems. Unlike OvR, it splits the dataset into one dataset for each class versus every other class. This produces significantly more datasets [BN06]

Performance

To evaluate the success of a ML algorithm, a quantitative measure of its *performance* must be designed. Usually this performance measure is specific to the task being carried out by the system

[GBC16]. So, the performance is how the final results of the ML algorithm will be evaluated. A widely measured metric is the *accuracy* of the model, namely the examples proportion for which the model returns the correct output. It is possible to get an equivalent information by measuring the *error rate*, namely the examples proportion for which the model returns an incorrect output [GBC16].
It is interesting how the ML algorithm performs on new data, never seen before, since this establishes how well it will work in the real world. These performance measures are evaluated through a *testing* dataset, different from the *training* dataset used for learning. It is a crucial decision to choose a performance metric that closely matches the ML system's desired behavior [GBC16].

Underfitting vs Overfitting The model fit is important for understanding the model accuracy:

1. *underfitting* occurs when a ML model fails to learn the problem properly and performs poorly on a training dataset. It is not possible to extract significant features from the training dataset, and so, collect the relationship between the input examples and the target values. Underfitting is usually caused by an oversimplification of the problem definition [JK15]

2. *overfitting* happens when a ML model learns optimally from a specific dataset, performing well on the training dataset but its accuracy drastically decreases with new unseen datasets or the addition of statistical noise to the training dataset. It can not generalize properly the features of the statistical distribution

2.2.3 Artificial Neural Network

Artificial Neural Networks (ANNs), hereafter abbreviated simply as *Neural Networks* (NNs), are computational models inspired by biological learning, that is, how learning happens in the human brain [GBC16].

Without going into detail, the structure of the human brain is made up of a huge network of linked neurons, each of which has a cell body, a set of dendrites and an axon (Figure 2.3). On average, the human brain has about 100 billion neurons, each one may be linked to up to 10 thousand other neurons, with up to one thousand trillion synapses connecting them [Zha19]. Oversimplifying, the dendrites of a neuron are the input channels, while the axon is an output channel. So, a neuron receives input signals through its dendrites, which are linked to the axons of neighboring neurons. In this way, an *action potential* (a rapid electrical pulse with enough intensity) can be transmitted along the axon of one neuron to all the other interconnected neurons. This is how a signal is propagated in the human brain. Thus, a neuron is similar to an *all-or-none*

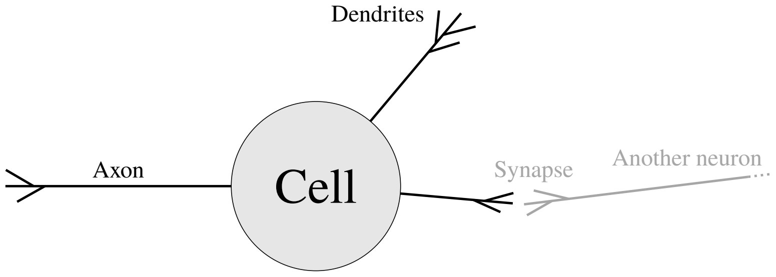


Figure 2.3: Simplified structure of a neuron of the human brain

switch, receiving a set of inputs and fires an action potential or nothing [GBC16].

The *McCulloch-Pitts* neuron (1943) [MP43] was the first computational model of an artificial neuron. It is considered a milestone of NNs history and an early model of brain function. This linear model was a binary classifier. It embedded the all-or-none activity principle, simulating the behavior of a single biological neuron, through a binary threshold with boolean logic [MP43; GBC16].

The *perceptron* (1958) [Ros58], originally proposed on the basis of the M-P neuron, takes the artificial neuron a step further. It was the first artificial neuron model that could learn the weights associated to the inputs [GBC16], in order to classify data. We could compactly abstract the operation performed by a single perceptron in Equation 2.14. The Figure 2.4 shows a simplified depiction of a perceptron with different inputs, the associated weights and a bias term. The perceptron computes a sum of the products of the inputs and their associated weights, then applies that result through an activation function.

$$y = f_{\int}(z) = f_{\int}\left(\sum_{i=0}^{n}(x_i \times w_i)\right) \qquad \begin{cases} [x_1, \ldots, x_n] \text{ inputs} \\ [w_1, \ldots, w_n] \text{ weights} \\ x_0 = 1 \text{ with } w_0 \text{ bias term} \\ f_{\int} \text{ non-linear activation function} \end{cases} \tag{2.14}$$

Neural Network Architecture

Similarly to human mind, a NN architecture is built with interconnected neurons. These ones propagate information across the network, by receiving inputs from neighboring neurons and mapping them to outputs, which will be received by the next neighboring neurons. The input

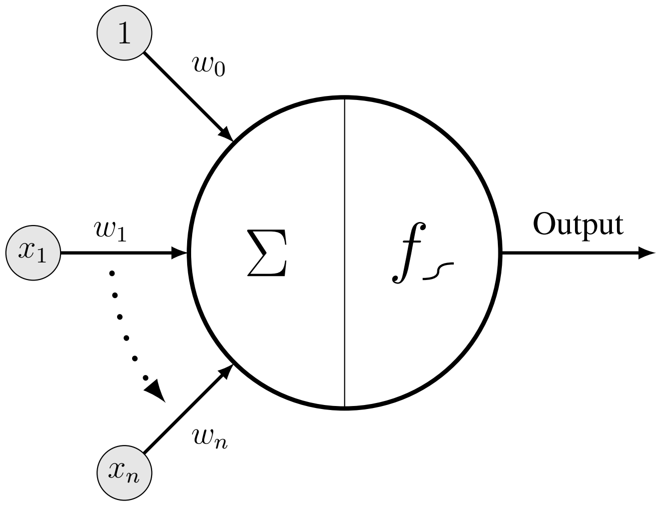


Figure 2.4: Nonlinear function implemented by an artificial neuron

is processed across neurons by functions predefined, known as *activation functions*, returning a numerical output [GPG18].
A NN can be described as a hierarchical directed weighted graph, whose nodes are the neurons [GPG18]. The NN output is defined through the activation functions and the weights of the neurons [Cal20]. A main advantage of a NN is the capability to provide automatic feature extraction directly from the raw data. More, another benefit is the scalability of the computational model because the results get better with larger models and more information [Dea16]: even if it requires more computation, the performance improves by building larger NN and training them with more data.
The main question of NN is about clearly understanding what the model is learning, thus, what function it is approximating [Guo+18]. A NN constraint is the request for a huge amount of data to provide a reliable result.
A NN architecture is generally a sequence of layers, each composed of a set of nodes (Figure 2.5):

- the *input layer* is the first one and it defines what data shape is acceptable to the NN, introducing an input into it
- the *output layer* is the last one and it defines what data shape has the last NN output, choosing the computation results
- the *hidden layers* are the middle ones, and may be more than one. Each hidden layer receives as input the output of the previous layer, and after having processed it, sends the results to the next layer. These layers have to approximate the function that better finds

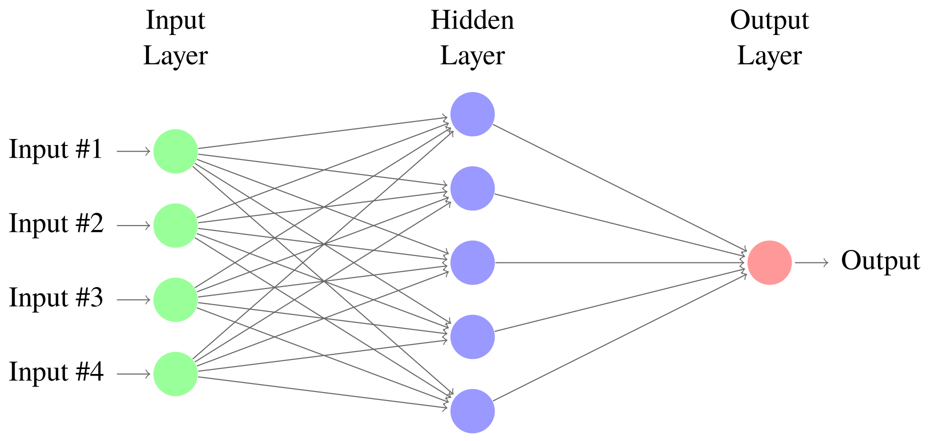


Figure 2.5: Example of a NN architecture

the input data statistical distribution. There are specific hidden layer designs based on the problem addressed

In a NN architecture, the connections between layers are called *edges*. Every edge is characterized by a weight, which affects the relationship between the two nodes involved. The weights are important elements and their management is critical, because it requires many mathematical details to be taken into account for a proper implementation. But it will not be discussed here, for a more extensive exploration refer to [GBC16].

Deep Learning

Deep Learning (DL) is an approach to AI and, specifically, a type of ML that enables computer systems to improve with experience and data [GBC16]. Note that, often, in the literature the terms NN and DL are used as synonyms: DL refer to large and deep neural network architectures [Den12]. A NN composed by several layers can be generally considered as a DL algorithm: in this case, the adjective *deep* in DL refers just to the depth of the network layers [GBC16]. In this work, these two terms can be used interchangeably.
In terms of learning description, DL exploits a hierarchy of representation of concepts: computational models can learn complex concepts building them out of simpler ones [LBH15]. Imagining a hypothetical graph showing this hierarchy between concepts, the graph is deep with many layers [GBC16].
DL algorithms attempt to exploit the unknown structure in the input distribution to find valid representations, with higher-level learned features described in relation to lower-level features [Ben12]. Each level is obtained by combining non-linear modules, evolving the representation at level x, starting from the raw input, into a representation at lightly more abstract level $x + 1$

[LBH15]. Automatically learning features at multiple layers enables a model to learn complex functions by mapping the input to the output directly from data, without totally depending on human-made features [Ben09].

Neural Network Training

The training is the first phase of a NN algorithm. Starting from the received input, the network learns how to compute a valid output. The data organization can affect the performance of the network. A significant training dataset for a NN has the following properties [GBC16]:

- the statistical data distribution, used by the NN during the training to learn and extract features automatically, must be representative of the problem
- the dataset needs to be clear, in reference to both the format and also the structure of the data, which should be expressed as simply as possible
- the dataset must describe the problem efficiently

The learning process occurs during the training phase. The two most common approaches are the supervised learning (paragraph 2.2.2) and the unsupervised learning (paragraph 2.2.2) [GBC16]. The learning process aims to find and manage the weights set that best fit the data patterns, in order to improve the accuracy of the final result. It is a task that requires a lot of computation. Two key functions are employed to accomplish the task [KMD20; Zel03]:

- a *cost* function (or *loss* function), continually evaluated during the learning. It computes an *error rate*, a statistical measure to be minimized
- a *optimization* function that performs the weights update. For example, a widely used optimization function is the *stochastic gradient descent* (SGD) [BN06]

If the problem is properly defined and the learning is effective, the NN finds features linking the input with the output [Cal20].

In the training phase, the *validation* process is important. In the validation, the NN training is assessed and the hyperparameters are tuned to achieve the highest possible accuracy. Parameter tuning is a difficult mathematical topic of the optimization research. The validation can reveal some flaws emerged in the training phase, such as overfitting (paragraph 2.2.2). A specific validation dataset, different from the training dataset, is used to compute this process [RN02].

$$\begin{tikzcd} x_t \arrow[r] & \boxed{h_t} \arrow[loop right, "f"] \end{tikzcd}$$

Figure 2.6: A RNN neuron

Neural Network Testing

The *testing* is another important phase in a NN model. The training phase has concluded and the NN has learned the input data distribution. Thus in this phase, there is the application of what has been learned. A specific testing dataset is used here [RN02], with the same properties, structure, representation and data format of the training dataset. However, the testing dataset needs to differ from the training dataset (or one of its subsets), in order to obtain a complete view of the actual network performances [RN02].

2.2.4 Recurrent Neural Network

The *Recurrent Neural Network* (RNN) [RHW86] focuses on the process of sequential data values. Most RNNs can also process sequences of variable length [GBC16]. The RNN has connections with loops, adding a memory to the network over time: for this reason, it was proposed for modeling time series [PMB13]. The neurons of the hidden layers can be linked back, giving a *back-propagation loop* and allowing previous outputs to be used as current inputs [AA18].
A RNN architecture is quite similar to that of a standard multilayer NN, with possible recurrent connections among hidden neurons associated with a time delay [PMB13]. Graphically, it appears as a direct weighted cyclic graph. Thanks to these connections, the model is able to remember information about the past inputs, allowing it to discover temporal correlations between events far from each other in the data [PMB13]. A RNN neuron has a specific state, storing a local memory for the current computation (as depicted in Figure 2.6): at the time t, x_t represents the input, h_t represents the state and f is the function which computes the new state h_{t+1}.
The RNN model presents several advantages, as the possibility of processing input of any length, but the model size not increasing with the input size [AA18]. The computation considers historical information and the weights are shared across time [AA18]. While in principle the RNN is a strong model with different benefits, in practice it is difficult to train properly [PMB13]. The computation can evolve slowly [AA18]. Applications of RNNs are mostly in the fields of natural language processing and speech recognition [AA18].

2.2.5 Long Short-Term Memory Network

The *Long Short-Term Memory* (LSTM) NNs [HS97] are a subset of the RNNs. It has been extensively demonstrated that LSTMs networks learn long-term dependencies more easily than the simple RNNs [GBC16]. The LSTM has shown to be incredibly effective in a variety of applications: unconstrained handwriting recognition, speech recognition, handwriting generation, machine translation, image captioning and parsing [GBC16]. As in RNNs, there are self-loops in the LSTM networks, but with the important addition of making the self-loop weight conditioned by the context, rather than fixed [GBC16]. Therefore, LSTM networks have the self-loop weights *gated*, that are managed by a dedicated hidden unit. The LSTM *cell* (Figure 2.7) replaces the usual hidden unit of an ordinary RNN [GBC16]. A LSTM cell is smarter than a usual arti-

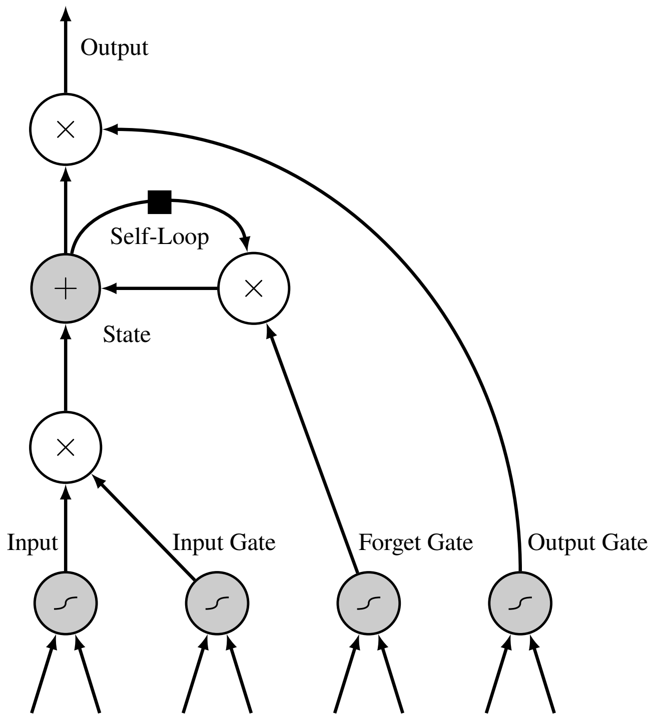


Figure 2.7: Block diagram of a LSTM cell. Cells are connected recurrently to each other. The input gate allows the accumulation of computed input feature values into the state. The state unit has a linear self-loop, whose weight is managed by the forget gate. The state unit can also be used as an extra input to the gating units. The black square represents a single time step delay. The output gate can switch off by the output cell [GBC16].

ficial neuron: it has a memory for recent sequences and contains three types of gates, managing the state and the output of the unit itself [GBC16]:

- the *forget gate* decides which information to discard, because irrelevant to the status evolution of the cell
- the *input gate* determines which input values will be used to update the memory state
- the *output gate* provides the output based on input and the memory of the unit

Thus, a LSTM cell behaves like a small finite state machine, where the internal gates have weights learned during the training.
In order to achieve the highest possible accuracy, there are specialized algorithms for the optimization function, that take into account the time component [Hoc+01]: for example, the *Back Propagation Through Time* (BPTT) [Moz95]. For further details refer to [Cal20; GBC16].

Human Activity Recognition

As mentioned before, LSTM NNs can learn and remember through large input data sequences. Typical applications of LSTM NNs are sequence classification and activity recognition in time series, usually where the data flow in input has no fixed size or duration [GBC16]. The success of LSTM NNs for time series problems suggested to employ it also in related topics, as the *Human Activity Recognition* (HAR) [OR16].

The HAR technique aims to predict what a human is doing from movements recorded with sensors. The goal of the HAR methodology is to determine a person's activities, based on a collection of observations of himself and the surrounding environment [Ang+13].
In this work, we have been inspired by the HAR state of the art. In particular, for the choice of a LSTM NN in order to recognize different activities over time. While these activities are usually performed by a human being, in our study, they are performed by a microcontroller (section 4.1) during a cryptographic operation. So in this work, we will use the LSTM with activity recognition approach against the ECC algorithms. We will present what we are interested in recognizing in chapter 3 (section 3.2, with subsection 3.2.1, and section 3.3), and we will detail the architecture of the LSTM we have built in subsection 3.4.3.

2.3 Side-Channel Attacks

A cryptographic algorithm may be proven secure in some theoretical model. Contrasted with the theoretical security, there is then an implementation security of a cryptographic algorithm. Also from a practical approach, in fact, cryptosystems implementation can present different flaws. In the literature, there are examples for which a cryptosystem is exposed to security risks not only by a theoretical approach.
Attacks that target the actual implementation of a specific algorithm, rather than its theoretical design, are denoted as *physical attacks*. There different kinds of physical attacks: *Fault Attacks* [BDL97], that involve stressing a device at the right time during a computation and exploiting the erroneous result to counter the security provided by the cryptographic algorithm. *Malformed-Input Attacks* [BMM00], similar to the previous one, where the attacker builds a malformed input, specific for the target to attack, and exploits the erroneous result. *Side-Channel Attacks* (SCAs), which are presented in the following section more in detail (unlike the other two attack classes, which are not considered in this work. For an exhaustive explanation, refer to [BDL97; BMM00]).

2.3.1 Overview

SCAs exploit weaknesses in the implementation of cryptographic algorithms, rather than the algorithms themselves [MOP08]. These attacks are based on a *side channel*, namely an unwanted channel where physical information leaks from the device during its sensitive computation.
SCAs can exploit different side channel leakages produced by a device in a cryptographic computation [Riv11], such as timing measurements [Koc96], power consumption [KJJ99], electromagnetic radiations [QS01], and even sound [GST17].

Kocher was the first who demonstrated, in 1996, the possibility to recover the secret key, manipulated by the algorithm implementation, by exploiting the leakage coming from the implementation itself. In one of his first works (i.e., *Timing Attack*, [Koc96]), he used information coming from the execution timing as a side channel leakage. In particular he extracted and analyzed traces made of the execution time of real-world cryptographic algorithms: with a sufficiently accurate timer, many of the algorithms analyzed were vulnerable. The goal of the timing attack is to recover the private key by measuring the elapsed time during the key-dependent operations. In most of the implementations he analyzed, operations took a different amount of time depending on the values of the data to be processed: in the case of an algorithm vulnerable to timing attack, these data would be the private key, plus other input such as the plaintext. An example of

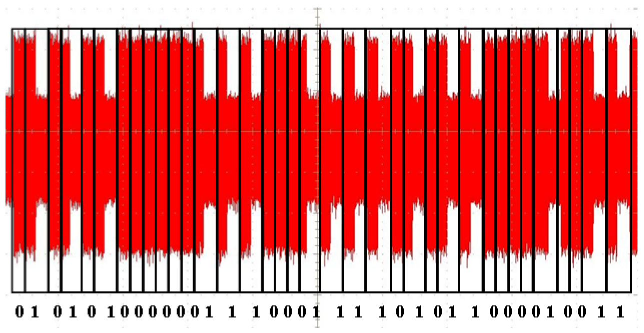


Figure 2.8: SPA leaks from an RSA implementation

two vulnerable cryptographic algorithms has been presented in Algorithm 2.5 and Algorithm 2.6.

Power Analysis (PA) is a method based on the analysis of power consumption leakages collected during (cryptographic) operations. A power consumption trace is a collection of power consumption measurements recorded during a (cryptographic) operation of a chip [KJJ99]. Like in the timing attack, power consumption also depend on the data as well as the running operation.

The *Simple Power Analysis* (SPA) is the first version of PA attacks, and it works with only a single power trace or a few ones. SPA aims to detect data-dependent power variations: it can disclose the sequence of instructions executed, so it can break cryptographic implementations where the execution path depends on the running data [KJJ99]. For example, Figure 2.8 shows a portion of a power trace of a modular exponentiation loop in which the SPA techniques reveal an RSA decryption key. This trace depicts a sequence of squares and multiplications involved in the modular exponentiation performed by the device: multiplications absorb more power than squares, thus the power trace in correspondence of multiplications presents higher peaks. Every exponentiation loop iteration of the algorithm computes a square, but multiplications are only executed with a 1-exponent-bit. This brings to deduce the operations pattern : each 1-exponent-bit appears as a shorter bump followed by a high one, instead a 0-exponent-bit looks like a shorter bump with no subsequent high one. As a result, the exponent bits may be retrieved as illustrated [Koc+11].

The *Differential Power Analysis* (DPA) is an evolution of the SPA: it is more difficult, but also more powerful and hard to prevent too [KJJ99]. In contrast to SPA, it requires to collect a large number of power traces. As the name suggests, it uses statistical tools such as correlation measurements between the leakage and processed data [Riv11]. In the scientific community, several DPA variants have been implemented over time. The basic method concerns partitioning a power traces collection into subsets, then calculating the difference between the averages of these subsets. If the subsets partition is correlated to the traces measurements, the subsets' averages difference will approach a non-zero value. Differently, if the assignment choice of a trace to a subset is uncorrelated to the traces measurements, the subsets' averages difference will get closer to zero as the number of traces grows. With enough power traces, this technique is effective even with noise in the measurements.

2.3.2 Classification and Countermeasures

In the literature, the SCAs are divided into two categories (as depicted in Figure 2.9) [Cla+10]:

- *horizontal* attacks
- *vertical* attacks

Horizontal attacks are single-trace, namely, the leakage is extracted from a single measurement, that is split into many parts through statistical tools. When the same guessable component of the secret is utilized in several internal operations of the algorithm, an horizontal approach may be useful [Bau+13]. Many complex examples of horizontal attacks exist in the literature: for instance, the authors in [Wal01] realize a specific method of attack on a cryptosystem, in which bits of a secret key can be guessed independently of the others, assuming that computations are performed sequentially on a single bit of the key and using information leaked through DPA. Another example of an advanced horizontal attack is illustrated in [NC17], in which the authors attack the `cswap` operation involved in the scalar multiplication implementation, aiming to spot the leakage of the conditional swap operation and then making a clustering to recover most of the key bits.
Vertical attacks, instead, are more-than-one-trace, namely, sensitive information are extracted from different algorithm executions. The vertical approach exploits the existence of a relationship between the different inputs, known by the attacker, and the secret key [Kab+19]. The classical DPA and *Correlation Power Analysis* (CPA) techniques fall into this class [Cla+10]. Another dichotomy of SCAs is [VS11]:

- *profiled* attacks
- *non-profiled* attacks

Typically, a profiled SCA uses known keys to build a detailed leakage model of the target device. In fact, an attacker requires two identical devices to execute a profiled SCA: the *target* device, over which he has limited control, and the *profiling* device, over which he has complete knowledge and control of the inputs and keys. A profiled SCA includes two phases: a profiling step in

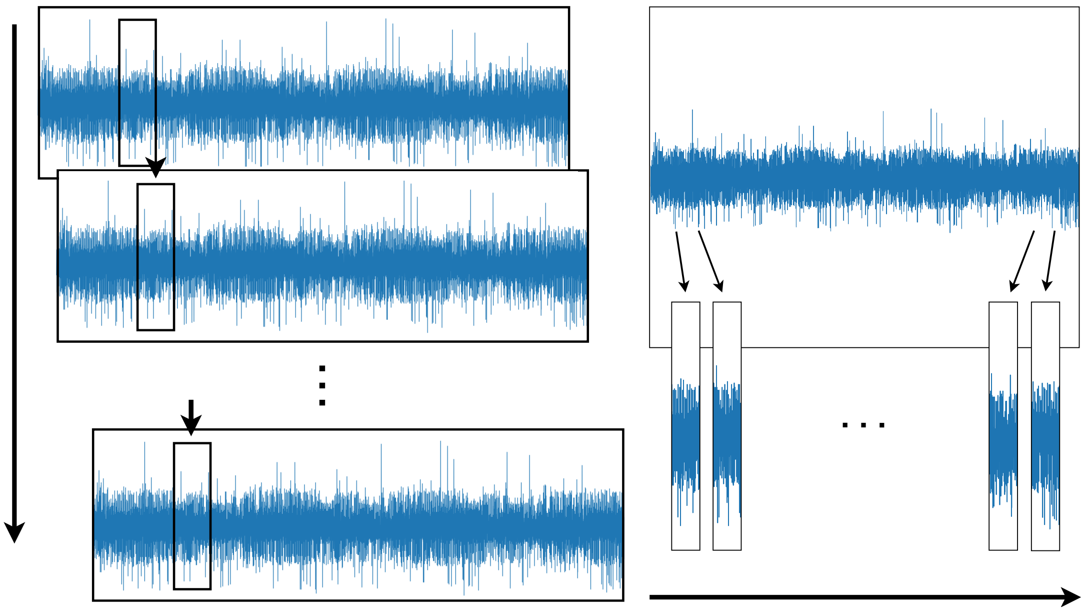


Figure 2.9: Vertical (left) and horizontal (right) SCA categories

which the target device's leakage model is profiled from the profiling device, and an attack step in which traces gathered from the target device are categorized depending on the leakage profile to retrieve the secret key value [Tim19].
By contrast, a non-profiled attack has no prior knowledge about the target device or its leakage model. So, briefly, it works by comparing key-dependent leakage models with actual measurements: if the attack is successful, the key candidate yielding the best comparison is the one managed by the target device.
Generally, there might be a significant difference between these two scenarios. As a result, profiled and non-profiled SCAs can be complementary and provide alternative perspectives on embedded device security [VS11].

There are two main categories of countermeasures against SCA [VBB16]:

- *theoretical* countermeasures, which follow arithmetic properties and can be proved in certain models
- *practical* countermeasures, which can not be formally proved

Practical countermeasures aim to block, or at least limit, the information leakages from side channels. Included in this class are countermeasures such as special shielding to decrease electromagnetic emissions, power line conditioning and filtering to counter power analysis, physical enclosures against micro-monitoring devices, noise and jitter additions to jam the signals emitted, and the use of shuffling techniques. Theoretical countermeasures intend to break the link between the information leakage and the secret data, by making one uncorrelated to the other. Masking is a well-known countermeasure of this class. The idea is to randomly divide a secret into several shares, such that the attacker needs all of them to rebuild it [PR13; ISW03].

2.3.3 ML and NN SCAs

In recent years, NNs have been gaining increasing popularity in the physical security field: because of their high scalability, they are finding application in numerous contexts. The advantage of being highly parallelizable, together with the increasing computational power of hardware, has contributed to their diffusion, allowing them to exploit dedicated computational models for hardware acceleration, such as *Graphics Processing Units* (GPUs) or *Tensor Processing Units* (TPUs), designed specifically for NNs [Jou+17].
Since the area of ML research advances, computer security researchers have attempted to apply automatic learning approaches in security-related issues. ML algorithms and, particularly, NNs are increasingly being used in the SCA scenario by the scientific community and there are several examples in the literature.
[Hos+11] is an early work which starts a comprehensive study on the ML application in SCA. The researchers propose the application of a ML technique on SCAs, using a kernel-based learning algorithm as a robust power traces analyzer: the chosen side channel is the power consumption, and the target is a software implementation of the AES symmetrical block cipher algorithm. The ML model includes two phases: the profiling phase (known as the training in ML terms) and the classification phase (known as the testing in ML terms). For a detailed explanation, refer to [Hos+11].
In [MPP16], the authors show that the ML techniques return a better statistical analysis, in comparison to traditional methods: a deep NN can surpass in results a classical *Template Attack* (TA) [CRR02], a widely used profiled SCA, since DL algorithms allow a better generalization of data distribution, without making assumptions. They used several ML techniques and NN models against different AES implementations. For a detailed explanation, refer to [MPP16].
In [Pic+17], the authors provide a detailed examination of ML techniques used in *side-channel analysis*. The ML benefits may be shown in situations where the leakage model is not well known. In this case, ML techniques are utilized to approximate the models in profiling phase. They use *Support Vector Machines* (SVM), a kernel-based ML family of methods to properly classify data [Vap99]; *Random Forest* (RF), a well-known ensemble ML method that relies on a number of decision trees [Bre01]; *Rotation Forest* (RF) [RKA06], a classifier ensemble ML method based on

feature extraction; *MultiBoost* (MB) [FS97], an ensemble learning method that forms a strong learner from a set of weak learners to minimize training errors. The results confirm that such techniques are much more powerful than usually considered. In fact, for a number of scenarios, the ML approach yields the best results compared to TAs. For a detailed explanation, refer to [Pic+17].
In [WPP21], it is shown how to extract functional features from side channel leakages for a successful TA through DL. The authors denote this combination as *DL-assisted TA*, in which they apply a DL approach of *similarity learning* to determine the most meaningful data embedding, based on the input traces, to use as input to a TA. To accomplish this task, they leverage the *Triplet Network* model [Wan+14], composed of three deep networks, each one identical to the others, with shared weights and a triple input composed of three samples: *positive*, *anchor* and *negative*. Positive and anchor samples share the same label, which is different for the negative: if the anchor sample represents a reference input and the positive sample a matching input, the negative sample represents a non-matching input. A *Triplet Loss* function minimizes the distance between the anchor and the positive, while it maximizes the distance between the anchor and the negative. For a detailed explanation, refer to [WPP21].
In [Per+21], the authors present a new DL-based iterative framework for horizontal attacks on implementations of public-key cryptosystems. The novel approach improves single-trace attacks, by lowering the amount of incorrect bits in a recovered private key. Protected ECC implementations are the target, in particular they attack the same software library of the research in [NC17], exploiting the `cswap` leakage. The framework is based on an unsupervised learning model (paragraph 2.2.2). The results achieved are significant, averaging more than 90% accuracy, that is a great improvement from state-of-the-art horizontal attacks. This accuracy is due to the use of a *Convolutional Neural Network* (CNN) [LB+95]: according to the same authors, the trained NN can significantly increase the number of right bits in a random private key related to a single trace. The attack framework is motivated by the assumption that deep NN generalize effectively, even when the training set includes a considerable number of noisy labels. For a detailed explanation, refer to [Per+21].
In [WPB19], the authors focus on different ML techniques, in order to mount a PA attack on the *Edwards-curve Digital Signature Algorithm* (EdDSA) [Ber+12]: they consider RF, SVM and CNN. They show that such techniques, especially the CNN, can be extremely powerful attacks compared to a classic SCA like a TA. In fact, when considering all available features of the dataset, CNN performs the best, giving a 100% accuracy: this implies the attack is perfect, breaking the ECC implementation with only a single trace in the attack phase. So, all the ML techniques prove great performance, but CNN is the best without dimensionality reduction: it is interesting because the CNN used by the authors here is the same as a related work, where it

performed extremely well against an AES implementation [Kim+19]. The network has not been further adapted for this project, hence, CNNs can work well on several SCA scenarios. For a detailed explanation, refer to [WPB19].
In [Wei+20], the authors extend the work in [WPB19], providing results for an additional protected target. The attack goals are two implementations of scalar multiplication on one of the most popular ECC curves for applications (i.e., Curve25519) [Ber06]. In the first implementation the power consumption is exploited as a side channel, while in the second one the authors analyze electromagnetic emanations to find side channel leakages. Also they consider several profiling attack methods, in addition to RF, SVM, CNN and the classical TA: *Gradient Boosting* (XGB) [CG16], a classification algorithm that combines the predictions of several weak learners to create a stronger learner; *Naive Bayes* (NB), a Gaussian classifier between the classification algorithms that applies Bayes's theorem. All the proposed profiling attack methods works well in unprotected implementation, but DL perform better in protected implementation and only CNN can easily break it. To conclude, all techniques exhibit strong performance, but CNN is the best if no dimensionality reduction is applied. For a detailed explanation, refer to [Wei+20].
In [Muk+18] the authors focus on recovering secret key information, by performing ML-based analysis on leaked power consumption signals from a *Field Programmable Gate Array* (FPGA) implementation of the ECC algorithm Double-And-Add-Always (Algorithm 2.7). The analysis of captured power signals is accomplished through four main classification techniques, in order to classify and recover secret key bits of the ECC algorithm. These are three ML algorithms (i.e., RF, SVM and NB) and one simple NN-based algorithm, the *Multilayer Perceptron* (MLP) [RHW86], one of the first NN models. The methodology adopted aims to attack one bit at a time, focusing on the least-significant three bits of the last nibble (i.e., bit 2, bit 3 and bit 4) of the secret key. The properties of the captured power signals can be employed as features, since they provide an accuracy of about 90%, so the authors can recover the secret key from the leakage with 90% accuracy. For a detailed explanation, refer to [Muk+18].
In the literature, several other works have applied deep NNs to profile attacks against symmetric [CDP17; Zai+20] and asymmetric cryptographic algorithms [Car+19] too: for further details on ML and NN based SCAs, also refer to the systematization of knowledge in [Pic+21].

Further Analysis

Below we focus on a few specific attacks, which also employ SCAs, along with other techniques in order to successfully break ECDSA implementations.
Some attacks can recover the secret key of a cryptographic algorithm, without prior knowledge and with minimum information thanks to SCAs, such as a small number of bits of a sensi-

tive operand. In the literature, the *LLL* attack on the digital signature schemes is well-known [FGR12]. This algebraic attack relies on the *Lenstra–Lenstra–Lovász (LLL) lattice basis reduction* algorithm [LLL82]. The attack exploits information on a few ephemeral key bits, retrieved by side-channel analysis, in order to recover the private key. Through this approach, collecting a relatively small number of digital signatures and their side channel related data, the authors might attack the ECDSA algorithm (Algorithm 2.2). The risk of an efficient attack is concrete, even leaking a few bits of the ephemeral key used during the cryptographic algorithm computation. According to the authors of [FGR12], three bits of the ephemeral key are sufficient on each digital signature, to recover the secret key in a few hundreds signatures (even two bits in specific cases).

In [Ara+20], the authors present a new class of side channel vulnerabilities in implementations of the Montgomery ladder used in ECDSA scalar multiplication. They prove that an exposure of *less-than-one* bit of the ephemeral key can lead to a full private key recovery. A leakage of less than one bit of information about the nonce means that it reveals the most significant bit of the nonce, but with probability < 1. For more details, refer to [Ara+20].

In [Ben+14], the authors propose a SCA based on cache hits and misses (FLUSH+RELOAD cache attack), in order to partially recover the ephemeral key used in a ECDSA implementation. Then, they apply a standard lattice technique to extract the private key. They prove the attack effectiveness by recovering the secret key with a high probability and by observing only a relatively small number of digital signatures (about 200 signatures for a 256-bit elliptic curve). For more details, refer to [Ben+14].

In [Roc+21], the authors describe a SCA focused on the secure element of an hardware device manufactured by Google (*Google Titan Security Key*), by the observation of its local electromagnetic radiations during the computation of ECDSA signatures. They prove that the electromagnetic side channel signal leaks partial information about the ECDSA ephemeral key. Then, they recover the sensitive information with a non-supervised ML technique. Finally, they mount a LLL attack to exploit this information with the goal of a private key recovery, using a few thousands of ECDSA signatures.

2.4 SCA Countermeasures

In this section, we introduce a few ECC scalar multiplication algorithms (subsection 2.4.1), some of which present specific countermeasures of SCAs. Then, we focus on the ECC implementation considered in this work (subsection 2.4.2), in order to analyze a few of its details and SCAs preventions.

2.4.1 Scalar Multiplication

There are several details to consider in an implementation of scalar multiplication on elliptic curves, certainly including the following:

- an implementation must be efficient and secure
- the point at infinity is a particular one to consider, since standard formulas may fail [RCB16]
- by examining the code flow on the target device, an attacker could perform special techniques to attempt to recover the scalar, a secret value that should not be revealed (section 2.3)

There are advantages and disadvantages, depending on the implementation used for scalar multiplication, but a possible distinction that can be made is between *regular* and *non-regular* ones.

Non-regular scalar multiplication algorithms

The simplest and most immediate implementation of the scalar multiplication is surely the non-regular one, where the flow of operations is not constant, but determined by the scalar value.

Sequential Addition To compute scalar multiplication, the simplest and non-optimal way is to add a point to itself numerous times, through sequential addition. The algorithm (Algorithm 2.4) requires k sums (where k is the positive integer), so $\Theta(k)$.
But there is an architectural security issue inherent to the non-regularity of the algorithm: the duration of computation is linearly proportional to the size of the input scalar. The larger a scalar is, the longer it takes to compute the result. It is possible for an attacker to learn information about the scalar, by examining the time it takes the device to complete the cryptographic operation [Koc96]. This problem could determine the recovery of the secret key in a cryptographic attack scenario (subsection 2.3.1).

Algorithm 2.4: Sequential Addition

```
input: $P \in \mathcal{E}(\mathbb{F}_p), k \in \mathbb{N}$
output: $kP$
begin
    $Q \leftarrow 0$
    for $i \leftarrow 1$ to $k$ do
        $Q \leftarrow Q + P$
    end
    return $Q$
end
```

Double and Add The first non-regular algorithm known in the literature [Riv11] is the *Binary Exponentiation*, also known as *Double and Add*. It runs a loop which scans the scalar bits and executes a point doubling, followed by a point addition when the current bit of the scalar is equal to 1.

There are two implementations that scan the scalar bits from left to right (Algorithm 2.5) and in the opposite direction from right to left (Algorithm 2.6), respectively. In all implementations this algorithm uses the binary representation of the scalar $k = (k_{n-1}, \dots, k_0)_2 \in \mathbb{N}$, i.e.: $k = \sum_{i=0}^{n-1} k_i 2^i$, where $k_i \in \{0, 1\}\ \forall i \in [0, n-1]$. So:

$$kP = \sum_{i=0}^{n-1} k_i 2^i P \tag{2.15}$$

The computational cost of the algorithm is reduced with respect to the sequential addition algorithm, due to the binary representation of the scalar, and it is in the order of $O(\log_2 k)$.

The Double and Add algorithm is subject to some security weaknesses, leading to specific attacks, as demonstrated by Kocher in [KJJ99]. It is insecure when a secret scalar is involved and a side-channel analysis of the implementation is permitted (e.g. a board that performs an ECDSA signature). Intuitively, the reason of such a flaw is due to the binary representation of the scalar. In fact, scanning the scalar bit by bit, there is always a point doubling, but a point addition only for the loop iterations where the scalar bit equals 1. Thus, doubling and addition have a different power absorption, therefore a different *side channel signature* and one operation can be distinguished from the other.

Algorithm 2.5: Left to Right (L2R) Double and Add

```
input: $P \in \mathcal{E}(\mathbb{F}_p)$, $k = (k_{n-1}, \dots, k_0)_2 \in \mathbb{N}$ with $k_{n-1} = 1$
output: $kP$
begin
    $Q \leftarrow \mathcal{O}$
    for $i \leftarrow n-1$ to $0$ do
        $Q \leftarrow 2Q$
        if $k_i = 1$ then
            $Q \leftarrow Q + P$
    end
    return $Q$
end
```

Algorithm 2.6: Right to Left (R2L) Double and Add

```
input: $P \in \mathcal{E}(\mathbb{F}_p)$, $k = (k_{n-1}, \dots, k_0)_2 \in \mathbb{N}$ with $k_{n-1} = 1$
output: $kP$
begin
    $Q \leftarrow \mathcal{O}$
    for $i \leftarrow 0$ to $n-1$ do
        if $k_i = 1$ then
            $Q \leftarrow Q + P$
        $P \leftarrow 2P$
    end
    return $Q$
end
```

Regular scalar multiplication algorithms

To be able to thwart the attack techniques mentioned in section 2.3, the scalar multiplication algorithm flow must be made independent on the value of the scalar manipulated. Such algorithms are called *regular*.

Double and Add Always L2R (Algorithm 2.5) and R2L (Algorithm 2.6) Double and Add algorithms, previously seen, perform conditional addition driven by secret data: this is dangerous for sensitive scalar. A solution could be designing an elliptic scalar multiplication algorithm with a constant flow of point operations, independent on the value of the secret data.
This is the *Double and Add Always* approach, first proposed in [Cor99] as a Double and Add variation (Algorithm 2.7), which performs a dummy operation in the binary algorithm loop whenever the scalar bit is equal to 0. Hence there are not conditional block and, in every loop iteration, a point doubling and a point addition are independently executed.

Algorithm 2.7: Double and Add Always

```
input: $P \in \mathcal{E}(\mathbb{F}_p)$, $k = (k_{n-1}, \dots, k_0)_2 \in \mathbb{N}$ with $k_{n-1} = 1$
output: $kP$
begin
    $Q_0 \leftarrow \mathcal{O}$
    for $i \leftarrow n-1$ to $0$ do
        $Q_0 \leftarrow 2Q_0$
        $Q_1 \leftarrow Q_0 + P$
```

```
            $Q_0 \leftarrow Q_{k_i}$
        *end*
        *return* $Q_0$
    *end*
```

Montgomery Ladder The *Montgomery ladder* was initially proposed, for the first time in [Mon87], as a scalar multiplication algorithm for a specific kind of elliptic curves with efficient point arithmetic: the so-called *Montgomery curves*.
The Montgomery ladder algorithm (Algorithm 2.8), such as Double and Add Always (Algorithm 2.7), is based on loop invariants the point registers Q_0 and Q_1 [Riv11]. Whatever the processed scalar bit value, there is always a point addition, followed by a point doubling. This allows to compute the point multiplication in constant time, making the algorithm regular. Note that the algorithm swaps the two operands from the `if` block to the `else` block.

Algorithm 2.8: Montgomery ladder

```
*input*: $P \in \mathcal{E}(\mathbb{F}_p), k = (k_{n-1}, \dots, k_0)_2 \in \mathbb{N}$ with $k_{n-1} = 1$
*output*: $kP$
*begin*
    $Q_0 \leftarrow \mathcal{O}$
    $Q_1 \leftarrow P$
    *for* $i \leftarrow n-1$ *to* $0$ *do*
        *if* $k_i = 0$ *then*
            $Q_1 \leftarrow Q_0 + Q_1$
            $Q_0 \leftarrow 2Q_0$
        *else*
            $Q_0 \leftarrow Q_0 + Q_1$
            $Q_1 \leftarrow 2Q_1$
    *end*
    *return* $Q_0$
*end*
```

Montgomery ladder with *(X, Y)*-only co-*Z* addition A Montgomery ladder algorithm based on projective coordinates is the *Montgomery ladder with (X, Y)-only co-Z addition* [Riv11] (Algorithm 9), also used in the target ECC implementation (subsection 2.4.2) to perform the scalar multiplication required for ECDSA.

Algorithm 2.9: Montgomery ladder with *(X, Y)*-only co-*Z* addition

```
*input*: $P \in \mathcal{E}(\mathbb{F}_p), k = (k_{n-1}, \dots, k_1, k_0)_2 \in \mathbb{N}$ with $k_{n-1} = 1$
*output*: $kP$
*begin*
    $(R_1, R_0) \leftarrow$ XYCZ-IDBL$(P)$
    *for* $i \leftarrow n-2$ *to* $1$ *do*
```

```
        $b \leftarrow k_i$
        $(R_{1-b}, R_b) \leftarrow$ XYCZ-ADDC$(R_b, R_{1-b})$
        $(R_b, R_{1-b}) \leftarrow$ XYCZ-ADD$(R_{1-b}, R_b)$
    end
    $b \leftarrow k_0$
    $(R_{1-b}, R_b) \leftarrow$ XYCZ-ADDC$(R_b, R_{1-b})$
    $\lambda \leftarrow$ FinalInvZ$(R_0, R_1, P, b)$
    $(R_b, R_{1-b}) \leftarrow$ XYCZ-ADD$(R_{1-b}, R_b)$
    return $(X_0\lambda^2, Y_0\lambda^3)$
end
```

The original algorithm is reported in Algorithm 2.9. It uses special functions (`XYCZ-IDBL`, `XYCZ-ADDC`, `XYCZ-ADD`, `XYCZ-FinalInvZ`) not present in the Montgomery ladder, since this algorithm variant deals with the co-Z factor, introduced where the addition of projective points share the same Z-coordinate. The pseudo-code of all these functions is in the appendix of [Riv11].

The Montgomery ladder co-Z performs better than the basic Montgomery ladder [Riv11]. Generally, co-Z algorithms perform better than the ones with standard coordinates [Bal+12]. Further mathematical aspects are not in the scope of this work, for details refer to [Riv11].

2.4.2 Countermeasures applied on Micro-ecc implementation

The *micro-ecc* library is a small and open-source ECDH and ECDSA implementation for 8-bit, 32-bit, and 64-bit processors [Mac14]. It is written in C, with optional inline assembly for ARM and other platforms. According to the author, it is resistant to known SCAs and reasonably fast. Some of its features include small code size and no dynamic memory allocation. The integer representation is in *little-endian*: to reduce code size, all large integers are represented using little-endian words, so the least significant word is first. It provides support for different standard elliptic curves, but this work focuses only on the `secp160r1`. For more details, refer to [Mac14].

Internals and Projective Z-coordinate

We focus on the `secp160r1` curve. In this curve, the size is of 160 bits, the public key must be at least twice the curve size (in bytes) long, and the private key must be 21 bytes long (the first byte will almost always be 0). The ephemeral key is a scalar value of 160 bits.

The `uECC_sign` function (Listing A.3) produces a digital signature on the hash of the specified message. In particular, in this function:

1. a random scalar k is randomly generated
2. the `uECC_sign_with_k_internal` function (Listing A.4) is called with the newly generated value:

- this function calls a `regularize_k` function (Listing A.5), to regularize k

The regularization function of the ephemeral key aims to prevent the side-channel analysis of k. Two values $k0$ and $k1$ are computed from k:

- $k0$ is the result from the sum of k and n order of the elliptic curve
- $k1$ is the result from the sum of $k0$ and n

At the end, a *carry* value is updated and returned, used to decide which between $k0$ and $k1$ will be employed as scalar in the `EccPoint_mult` scalar multiplication function (Listing A.6). In our case, it seems that $k1$ is always selected, therefore simplifying:

$$scalar = k + 2n \tag{2.16}$$

The $2n$ in Equation 2.16 should explain why we always get 0x02 as the value of the most significant byte of the scalar.

After the regularization function return, `uECC_sign_with_k_internal` (Listing A.4) calls the scalar multiplication function `EccPoint_mult` (Listing A.6) with the selected k. Among other activities, this function performs:

1. an initial double (`XYcZ_initial_double`)
2. in `secp160r1` case, an iterative loop of 160 iterations over the 160 bits of k, starting from the the most significant:
    - inside the loop `XYcZ_addC` and `XYcZ_add` are executed, which perform a fixed sequence of field operations (modular and not)
3. the least significant bit of k, at position 0, is managed differently, outside the loop

In base of this explanation, in the construction of the tree of fingerprint operations (subsection 3.2.1), we discard the most significant bit of the ephemeral key k.

Note that, the parameter `initial_Z` of `EccPoint_mult` represents the projective Z-coordinate of the base point of the curve (subsection 2.1.2). This value changes every time at the beginning of the iterative loop of the scalar multiplication. In fact, with a *Random Number Generator* (RNG) function specified, the `uECC_sign_with_k_internal` function gets a random `initial_Z` value as a random coordinate. In order to mount our attack, we assume $Z = 1$.
We have developed a more general attack, without assumptions about Z, and this will be discussed in a successive publication.

Exploited Leakage

The ECC scalar multiplication algorithm is implemented by single field operations, thus modular. These operations are: addition, subtraction, multiplication and square. In micro-ecc, the functions that implement these field operations are, respectively: `uECC_vli_modAdd`, `uECC_vli_modSub`, `uECC_vli_modMult_fast`, and `uECC_vli_modSquare_fast`.
Addition and subtraction are one the opposite of the other. In their implementation, the modular reduction is performed considering the opposite operation. If the operation result overflows (or underflows) the modulus, the opposite operation brings the result in the bounds defined by the modulus:

- in the addition, with a overflowing result, the modulus is subtracted from it
- in the subtraction, with a underflowing result, the modulus is added to it

As example, the `uECC_vli_modSub` function is listed in Listing 2.10 (for the `uECC_vli_modAdd` function, refer to [Mac14]). In this work, we are not interested in the exploration of multiplication and square implementations. To access the micro-ecc code of all the operations, refer to [Mac14].

```c
/*
    Computes result = (left - right) % mod.
    Assumes that left < mod and right < mod, and that result does not overlap mod.
*/
uECC_VLI_API void uECC_vli_modSub(uECC_word_t *result,
                                  const uECC_word_t *left,
                                  const uECC_word_t *right,
                                  const uECC_word_t *mod,
                                  wordcount_t num_words) {
    uECC_word_t l_borrow = uECC_vli_sub(result, left, right, num_words);
    if (l_borrow) {
        /*
            In this case, result == -diff == (max int) - diff. Since -x % d == d - x,
            we can get the correct result from result + mod (with overflow).
        */
        uECC_vli_add(result, result, mod, num_words);
    }
}
```

Listing 2.10: `uECC_vli_modSub`

In section 2.3, we present a general overview of the main SCAs in literature. A SCA exploits weaknesses in the implementation of cryptographic algorithms. In this way, a leakage produced by a device about the processed information, during a cryptographic computation, can be unsafe. Focusing on the leakage of `uECC_vli_modSub` (Listing 2.10), the result depends on the operands value. In fact, a subtraction is always performed. Whereas, only when the operands of the function underflow the modulus, an addition is also performed to apply the modulus. So, we can infer that a field operation performing modular reduction lasts longer.
An attacker who could recognize the duration of the field operations, can also retrieves information about the operands, and thus, about the value of one bit of the scalar. If an attacker is able to exploit this leakage, and so, to retrieve a few scalar bits on each digital signature, then he could recover the secret key used for the ECC scalar multiplication in a few hundreds signatures, with a LLL attack (subsubsection 2.3.3): three bits are sufficient [FGR12], but, under certain conditions, even less than one [Ara+20].

Chapter 3

Simulation

In the previous chapter, we introduced the theory background of this work. In this chapter, we present a simulation environment which aims to reduce the variables number to be managed, in order to improve the work of [Nav20]. We choose to move to an easier environment for us to control, namely a simulation environment, to upgrade and improve a NN in charge of performing a SCA. After the achievement of satisfying results on the simulation with the NN, we move to the real use case, in order to break an ECC implementation.
After a brief description of the software tools involved (section 3.1) and an introduction to the previous work (section 3.2), we will explore the simulation environment (section 3.3, section 3.4) and the LSTM architecture we have implemented, analyzing the training and testing phases. Finally, we will expose the simulation results (section 3.5).

3.1 The Software Framework

To develop this work we make use of several services.

Google Colaboratory (Colab) [Col], is a product from *Google Research*. Colab is free, the only requirement is to own a *Google account*. Colab allows to write and execute arbitrary `Python` programming language code through the browser. More technically, Colab is a hosted *Jupyter Notebook* [Klu+16] service that requires no setup to use, while providing free of charge access to computing resources like GPUs and TPUs. Programs written on Colab are called notebooks and are automatically saved in the *Google Drive* associated with the used account. For the base account used in this work, Colab resources are neither guaranteed or unlimited and the usage restrictions can change. Notebooks run by connecting to virtual machines that have a maximum lifetime of 12 hours, and also they disconnect from virtual machines if left idle for too long.

Kaggle [Ant+10] is a Jupyter Notebooks environment similar to Google Colab. It is a good alternative to the previous one, with free GPUs access and a huge repository of community published data and code.

SageMath [The21] is a free open-source mathematics software system. It builds on top of many existing open-source packages and is `Python`-based.

TensorFlow [Mar+15] is an end-to-end open source platform for ML. It has a wide range of tools, libraries, and community resources that allow researchers and developers to build and deploy ML models.
TensorFlow's high-level APIs are based on the *Keras* [Cho+15] API standard for defining and training NNs. Keras is a high-level API for building and training DL models and `tf.keras` is TensorFlow's implementation of this API. While TensorFlow is a library oriented to multiple ML tasks, Keras is a high-level NN library that runs on top of TensorFlow.

STM32CubeIDE [STM] is an integrated development environment (IDE) of *STMicroelectronics*. It is an all-in-one multi-OS development tool. STM32CubeIDE is an advanced C/C++ development platform with peripheral configuration, code generation and compilation, debug features for *STM32* microcontrollers and microprocessors. It is based on the Eclipse®/CDT™ framework and `GCC` toolchain for the development, and `GDB` for the debugging.

3.2 Previous Work

As we said in chapter 1, in this work we start from the results of [Nav20]. Thus, in this section, we briefly present those results. In [Nav20], a new technique was proposed to exploit the power consumption of a modern device running an ECDSA algorithm implementation, in order to retrieve the *handled* private key. Some traces of power consumption of an off-the-shelf microcontroller performing sensitive ECC operations were acquired, in order to detect possible correlations between the power absorbed by the device and the value of critical variables handled by it. In [Nav20], a power consumption trace has an hash message as input and its ECDSA signature as output, and it records the consumption of the chip during this cryptographic computation. Briefly, an ECDSA execution trace is composed of three basic information:

- the samples of power consumption recorded
- the clock timing of one field operation of the ECC scalar multiplication in the ECDSA

implementation (the clock timing is synchronized with the capture board sampling rate)

- a label identifying the single field operation above

In order to compose a training dataset to train a LSTM network, the author built a sort of *minimal traces*, composed by specific real power consumption templates of the field operations of interest:

- *addition*:
  - *without overflow* (`ADD`)
  - *with overflow* (`modADD`)
- *subtraction*:
  - *without underflow* (`SUB`)
  - *with underflow* (`modSUB`)
- *multiplication* (`MUL`)
- *square* (`SQR`)

The extracted templates were composed in a training dataset with labels, that were given in input to an LSTM, using the supervised learning technique to associate a class, which identifies the operation itself, to each specific minimal trace. So all the minimal traces were shuffled and merged together into a unique large power consumption trace.
In order to provide the LSTM with the causality information, a *sliding window* was applied on the traces, associating a label to each window. In particular, every window was associated with one of the following labels:

- *short operation* (SO)
- *long operation* (LO)

depending on the first operation appeared in the window. The author considered `ADD`, `modADD`, `SUB`, and `modSUB` as SO, since they last less than LO, which were `MUL` and `SQR`. Thus, a binary classificator NN was trained, which was in charge of retrieving the class of the first operation in the window.
The NN accuracy in output was variable, depending on the window of operations analyzed by the LSTM: it started from 49% for the worst case up to 93% for the best one.

> *The first neural network presented [. . .] could correctly reconstruct the sequence of operations (up to a certain precision), proving that the approach proposed is a possible solution for the problem of recognizing the modular operation on a complete power consumption trace. However, an attacker cannot use the raw neural network output to perform a direct practical attack* [Nav20].

The main problem was that the NN accuracy was not sufficient to exploit the result on a LLL algorithm (subsubsection 2.3.3) to retrieve the private key. A purpose of this work is to improve and optimize the results in [Nav20]. Therefore, this study can be considered the continuation of the work started in [Nav20]. An achievement of this work is to consider a device as a human being by tracking its running activities, similar to the HAR methodology (subsubsection 2.2.5), with the aim of accomplishing the LLL attack.

3.2.1 Fingerprint Operations

In [Nav20] the author explains the basic assumptions for the attack. In order to mount a LLL attack (subsubsection 2.3.3), for a set of ECDSA signatures, the attacker needs the input message (or the hash of the input message), the output signature, as long as as much bits of the ephemeral key as possible (despite three are sufficient [FGR12], or even less than one [Ara+20]). In particular, the authors of [Rya19] and [Nav20] observe that individual bits of the ephemeral key of a single ECDSA execution can be retrieved if the attacker is able to recognize the duration of particular operations, dubbed *fingerprint operations*. The fingerprint operation is a modular one that behaves differently when the processed bit is 0 or 1, because it performs overflow in one case, but does not in the other one. So, knowing if a fingerprint operation performs modular reduction (so lasts longer), makes it possible to identify the value of the corresponding key bit handled. For a more detailed explanation, refer to [Nav20].

3.3 Simulation Goal

In [Nav20], the author immediately begins with the real use case, whereas in this work, we begin with a simulation. We build a simulation environment, to reduce uncontrolled variables and understand how to improve previous results. The final goal is always to mount a LLL attack against a modern microcontroller running the ECDSA algorithm (Algorithm 2.2), implemented by the micro-ecc library for embedded systems (subsection 2.4.2).
With this aim, we need a NN that can precisely distinguish the two classes of operations previously defined (LO and SO), in order to be able to determine the value of the handled key bit (subsection 3.2.1). As mentioned at the beginning (chapter 1), the technical limitations identified

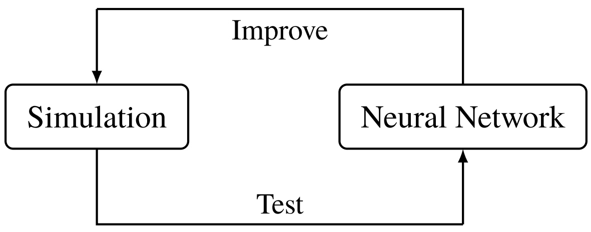


Figure 3.1: Simulation environment design as an iterative loop

in [Nav20], such that the real attack implementation was not possible, were the leakage model of the device (section 4.1) and the acquisition system of the power consumption traces (section 4.2). Also the management of the dataset to train the NN was critical, as it was unbalanced.
In order to improve the results of [Nav20], we start by using a simulation, where we can better control all the parameters. Then, after obtaining satisfying results on the simulation, we move to the real environment. The goal of a simulation environment is to reduce the number of variables, managing the leakage model and the trace acquisition system. In this way, we can then focus only on the LSTM, to better evaluate its performance and adjust it for the test in the experimental use case with real power consumption traces.

3.4 Simulation of the Target Implementation

Ideally, the design of the simulation environment follows an iterative loop (as depicted in Figure 3.1). We start from a simple raw simulation of the real use case and we evaluate the behavior of the NN on it. Thus, for each loop iteration, we introduce a change in our simulation: for every refinement added, we always test the NN behavior on it and, eventually, update some parameter to increase its accuracy.
The purpose is to obtain a detailed simulation of the hardware at hand, where we can train a NN to achieve maximum accuracy in distinguishing SOs from LOs. Hence, to bring the best version obtained of our LSTM on the real use case.

The power consumption model we choose for the simulation is the *Hamming Weight* (HW) [Tho83]. The HW of a generic bit string is defined as the number of non-zero elements in the string. The HW is a model generally adopted in the literature, because it best approximates the physical phenomenon underlying the power consumption [Cha+99; RPD09]. The idea to realize the attack is to exploit the dependence between the power consumption of a chip and the data processed by the chip. Practically, in the simulation there is a loop that, for each bit of the key, performs a certain sequence of operations. The HW of the result of each intermediate operation

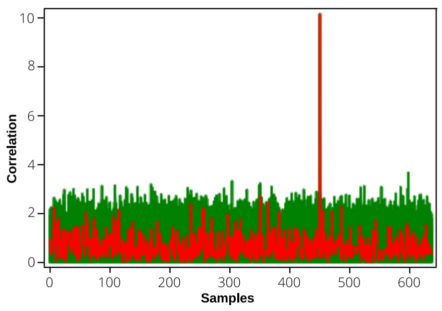


Figure 3.2: DPA: key size of 160 bits, 8 bits guessed with 100 synthetic power consumption traces

of the loop becomes a point of the synthetic power consumption trace.

We choose the Montgomery Ladder co-Z (Algorithm 2.9) to simulate the scalar multiplication (subsection 2.1.3) of the ECDSA algorithm, because it is also implemented by the micro-ecc library (subsection 2.4.2).

3.4.1 Simulated Dataset

First, we start with an implementation on `SageMath` [The21] of the ECDSA algorithm with Montgomery Ladder co-Z, by producing the HW, resulting from every single iteration of the scalar multiplication loop, as a leakage. With as few as one hundred synthetic power consumption traces, we are able to successfully apply a DPA (section 2.3), guessing with certainty up to 8 bits of the secret key on an elliptic curve of 160 bits (as depicted in Figure 3.2). The purpose of mounting a DPA is twofold. First of all, it helps to understand how a SCA works, but it also provides a test to understand if the implementation is correct. Briefly, we use this DPA attack to validate the simulation.

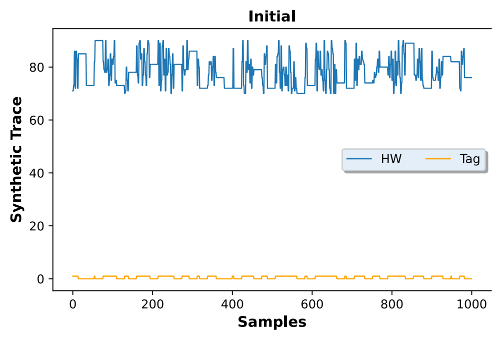
In order to have handy GPU power, we move to `Colab` [Col] for the next steps with the NN. When the original LSTM is applied to our synthetic power consumption traces, the results obtained are not similar to those presented in [Nav20]. After a careful examination, we can observe that this is due to the difference in leakage model and leakage pattern between the raw first simulation and the real experiments.
So we build a *simulated dataset*, composed by a certain number of synthetic power consumption traces, with the following characteristics:

1. we simplify by using only two operations classes (SO and LO), referring to these as two operations, since we are interested in a binary classification

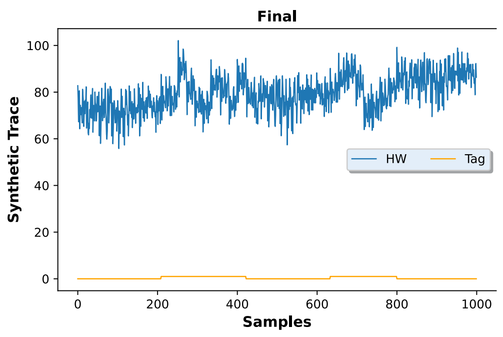
2. we associate a *boolean tag* to each operation, in order to identify them. So one sample of a synthetic power consumption trace is a pair, consisting of the HW value and a `true` or `false` tag
3. we introduce a *horizontal component* into the synthetic power consumption traces, since operations have their own duration. A way to model this information is to repeat a sample as many times as the duration of the operation performed (Figure 3.4a)
4. we carefully *balance* the simulated dataset, in order to divide the operations equally, depending on the duration. In every synthetic power consumption trace, the number of SO samples should be similar to the number of LO samples
5. for each operation, we apply a *numerical pattern*, specific and equal in size to the duration of that operation (Figure 3.4b):
    - values in the pattern are within a limited range
    - the result of the pattern application to a synthetic power consumption trace is, for each trace sample, the sum of the HW with the corresponding pattern value
    - first, we apply a *random pattern*, to make the synthetic traces similar to the real ones, empirically set in a $[-15, +15]$ range
    - finally, we apply an *average pattern*, extracted from the real power consumption traces of the field operations:
        (a) we pre-process the real power consumption traces of the field operations collected, cutting the duration of each operation to the shortest recorded
        (b) we extract an average pattern, computed by the mean of the values of the real power consumption trace of the field operations
        (c) we associate the LO with the `MUL` average pattern, as we associate the SO with the `ADD` average pattern
        (d) since the raw average patterns were flattened to zero, in order to avoid NN numerical problems, we multiply them by a constant coefficient (350), empirically set in order to obtain a range of values similar to the random pattern case, where a good accuracy was achieved
6. to closely simulate a real use case, we add a Gaussian statistical *vertical noise* on the samples, with a *mean* of 0 and a *variance* of 1.5 (both empirically set values, Figure 3.4c)
7. to closely simulate a real use case, in particular the jitter of an operation size, we add a 10% *horizontal noise*, since an operation may differ in duration in a real use case (Figure 3.4d):

- with the horizontal noise addition, we need to apply also a longer pattern, so we stretch the average pattern by a small percentage, depending on the horizontal noise value. The average pattern is stretched simply by concatenating to it a new portion. Since both are unused here, we use the `SQR` average pattern to stretch the LO pattern, whereas the `modADD` average pattern to stretch the SO pattern

In the simulation environment, we can manipulate the length of SO and LO, depending on the different aspects that we want to test. But we always respect the real use case magnitude order between them, and we keep the dataset balanced.
The introduction of these refinements, in each simulation loop iteration, is depicted in Figure 3.4. An example of a synthetic power consumption trace is depicted in Figure 3.3. In particular, we compare two portions of different synthetic traces, respectively before (Figure 3.3a) and after (Figure 3.3b) the changes introduced by the simulation.
Along with the samples is clearly visible a *ground-truth* given by the boolean tags at the bottom, which identifies the class of the operation.
We start by collecting 50 synthetic power consumption traces and, gradually, we lower the number to just under a dozen. The size of one single synthetic power consumption trace can differ from 12 thousands samples at the beginning to over 300 thousands samples, with all the refinements introduced.

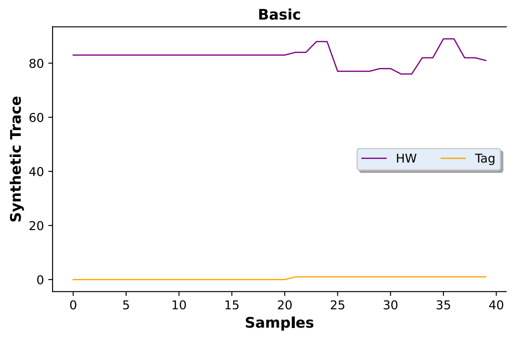


(a) The first thousand samples of a synthetic power consumption trace at the beginning of the simulation

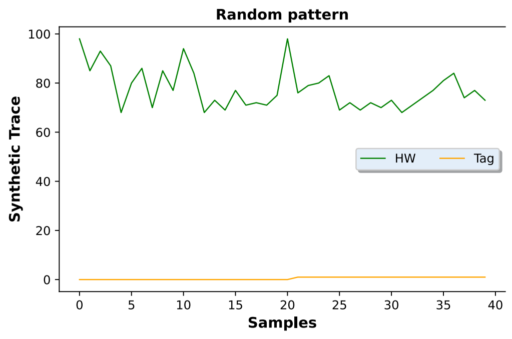


(b) The first thousand samples of a synthetic power consumption trace at the end of the simulation

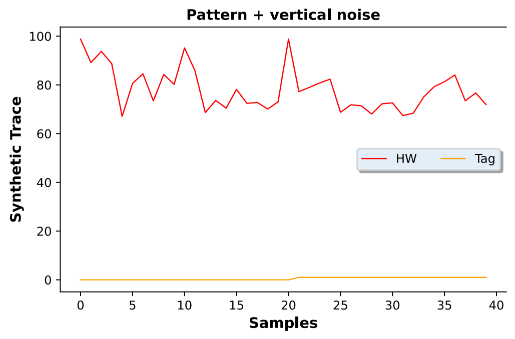
Figure 3.3: A portion of different synthetic power consumption traces, at the beginning (left) and at the end (right) of the simulation



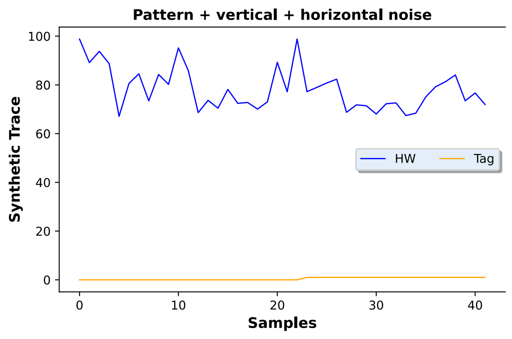
(a) Zoom-in of a synthetic power consumption trace with duration of operations



(b) Zoom-in of a synthetic power consumption trace with random pattern



(c) Zoom-in of a synthetic power consumption trace with vertical noise



(d) Zoom-in of a synthetic power consumption trace with horizontal noise

Figure 3.4: Evolution of a synthetic power consumption trace

3.4.2 Sliding Window

In order to comply with the NN requirements, the synthetic power consumption traces samples must be pre-processed to form a dataset to feed the LSTM.
The LSTM designed in subsection 3.4.3 requires a fixed-size input flow, obtained through an implementation of the *sliding-window* algorithm. This is a technique of dynamic programming. The window concept is the basis of the algorithm. A window is an atomic element and is composed of a subset of contiguous data. A window can slide over the data, forming a different portion of sequential data at each advance step. The pseudo-code implementation of the algorithm is in Algorithm 3.1.

Algorithm 3.1: Sliding-Window

```
input: $traces\_num$ = traces number, $window$ = window size, $step$ = step size, $SO\_time$ = shortest SO duration
output: dataset
```

```
begin
for $trace \leftarrow 0$ to $traces\_num$ do
    for $i \leftarrow 0$ to $trace\_size - window + 1$ do
        $X\_dataset \leftarrow X\_dataset + HW\_traces[trace][i : i + window]$
        $tags\_window \leftarrow tags\_traces[trace][i : i + window]$
        if #true in $tags\_window >= SO\_time$ then
            $Y\_dataset \leftarrow Y\_dataset + [1]$
        else
            $Y\_dataset \leftarrow Y\_dataset + [0]$
        $i \leftarrow i + step$
    end
end
$dataset \leftarrow X\_dataset, Y\_dataset$
return $dataset$
end
```

Window Classification

In this work, every window is composed of a fixed number of samples of a power consumption trace. A window contains approximately a number of samples between the average length of SO and LO.
We need to empirically extract the minimum length that a SO can have (`SO_time` in Algorithm 3.1). In this work, this value is 153. Thus, a window is `true`-classified if it contains at least 153 contiguous SO samples number. That is, the window is framing an entire SO. Conversely, the window of samples is `false`-classified because it is framing a LO or otherwise only partially a SO.

Window Size

The size of the window and the advance step are critical, because they can introduce classification errors. These values are a trade-off between the computational cost and the NN accuracy. With small windows, the dataset size increases and the LSTM may misclassify operations. Otherwise, with big windows, the LSTM may perform classification approximations.
We always look for an intermediate window size between the two operations to be classified (SO and LO). In the real use case, the shortest subtraction is 153 samples long, while the shortest `MUL` exceeds 2 thousand samples (about 2700 samples). We have tested different sizes: for the window from 110 to 1000, and for the advance step from 1 to 10. After several tries, we obtain the best results establishing the window size of 500 and the step of 10. With these values, the dataset processed by the LSTM is about 240 thousand windows long.

3.4.3 LSTM Architecture

In Algorithm 3.2 we provide the final pseudo-code version of the LSTM design, used both in the simulation phase, with synthetic power consumption traces, and in the experimental phase, with real power consumption traces.
In this work, we choose an LSTM among the different possible structures of NNs for several reasons. First, it processes sequences of data and performs well with time series, this can come in handy since there may be different delays between SOs to recognize. It has temporal consciousness, maintaining memory over time (subsection 2.2.5). Moreover, we were inspired by the HAR method and literature, where the LSTM works well and accurately recognizes activity patterns (subsubsection 2.2.5).

Algorithm 3.2: LSTM Model

```
begin
    $model \leftarrow Sequential()$
    $model \leftarrow model + Conv1D(filters = 64, kernel\_size = 3, activation = ReLU, input\_shape)$
    $model \leftarrow model + AveragePooling1D(pool\_size = 10)$
    $model \leftarrow model + LSTM(1000)$
    $model \leftarrow model + Dropout(0.5)$
    $model \leftarrow model + Dense(1000, activation = ReLU)$
    $model \leftarrow model + Dense(2, activation = softmax)$
    $model \leftarrow compile(loss = categorical\_crossentropy, optimizer = SGD, metrics = accuracy)$
    $model \leftarrow build()$
end
```

The LSTM architecture adopted in this work is inspired by the NN designed in [Nav20].
A ML *model* is a function with learnable parameters that maps an input to an output. The model is trained on data to find the best parameters. An accurate mapping from the input to the expected output is achieved by a well-trained model. In this work, the model is the LSTM architecture, according to its meaning in the framework used (section 3.1). The model is *Sequential*, which is a plain stack of layers where the input of one layer is the output of the previous one. The sequential model allows adding layers sequentially.

Convolutional Layer

The `Conv1D` layer applies a *filter*, that is convolved with the input data over a single dimension. The *convolution* is a mathematical operation and the main part of the layer. It is a specialized kind of linear operation that involves the multiplication of a set of weights with the input. The convolution operation is typically denoted with an asterisk [GBC16]:

$$s(t) = (x * w)(t) \tag{3.1}$$

The first argument (the function x) to the convolution is often referred to as the *input*, and the second argument (the function w) as the *kernel* or filter. The output is referred to as the *feature map* (Equation 3.1).
The multiplication used is a dot-product (also known as scalar product) and the filter is smaller than the input data. The application of a filter smaller than the input is intentional, since it allows the same filter to be multiplied by the input data several times at different points on the input.

In ML applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters, updated by the learning algorithm. These multidimensional arrays are usually referred as *tensors* [GBC16]. The convolution is a commutative operation, resistant to noise and we use it here in order to amplify some features. The framework refers to:

- the number of output filters in the convolution as the `filters` (Algorithm 3.2)
- the shape of the filter as the `kernel_size` (Algorithm 3.2)
- the activation function, that pass each output value in the feature map, as the `activation` (Algorithm 3.2); here we decide to use the *Rectified Linear Units* (`ReLU`), one of the most commonly used activation function in DL models
- the first layer in the model needs an `input_shape` argument (Algorithm 3.2), which specifies for the first time the shape of the input

The convolutional layer is in charge of boosting the performance of the LSTM. For more details, refer to [GBC16] and the framework documentation [Mar+15; Cho+15].

Pooling Layer

The usage of a *pooling* layer, after the convolutional one, is of common use. Following one or more convolutional layers, a pooling layer is used to consolidate the features learned in the previous feature maps [GBC16]. Briefly, pooling can be considered an operation for generalizing feature representations. As a result, generally the model overfitting problem on the training data is reduced.
Through down-sampling the previous feature maps, the pooling operation allows to get the representation approximately invariant to small translations of the input. With a local translation invariance, the values of most of the pooled outputs do not change with a small amount of input translations [GBC16]. With the application of a pooling layer, the NN statistical efficiency can

be considerably improved.

The pooling layer performs upon each feature map separately, computing a new set of pooled feature maps. There are several functions used in the pooling operation: here we use the *average pooling*. The `AveragePooling1D` layer downsamples the input representation by taking the average value over the pool defined by `pool_size` (Algorithm 3.2).
The pooling layer is in charge of boosting the performance of the LSTM. For more details, refer to [GBC16] and the framework documentation [Mar+15; Cho+15].

LSTM Layer

The `LSTM` layer is the NN core. Compared to the size of the original input, the input coming to this hidden layer is smaller. The required data shape is of the type `[batch, timesteps, feature]`:

- *batch* is the number of input data entries the LSTM will process, so the total number of windows
- *timesteps* is the number of samples for each input window
- *feature* is the number of variables in the computation. In this work, the only feature considered is the power consumption over time

According to the dataset size and after several tries, here we choose a LSTM units number of 1000 (Algorithm 3.2). For more details, refer to [GBC16] and the framework documentation [Mar+15; Cho+15].

Dropout Layer

Dropout refers to dropping out neurons in a NN. By dropping a unit out, it is temporarily removed from the network, along with all its incoming and outgoing connections [Sri+14]. The dropout operation of a neuron is performed temporarily setting its weight equal to zero. Dropout is a computationally efficient regularization, which can also be combined with other techniques of regularization to yield further improvements [GBC16].

The `Dropout` layer helps prevent overfitting problems, providing a better generalization of the data distribution, by randomly switching off some neurons in the training phase of the LSTM. In this way, the NN discards the experience of those dropped neurons. Here we choose a 0.5 dropout rate (Algorithm 3.2), that is a common probability value for a hidden layer [Sri+14]. With this rate, a frequency of 50% of input units are randomly set to 0 at each step during training time. For more details, refer to [GBC16; Sri+14] and the framework documentation [Mar+15; Cho+15].

Dense Layer

The *Dense* layer is a fully-connected NN layer, so each neuron in the dense layer receives input from every neuron of its previous layer. A fully-connected layer is typically used in the final steps of a NN.
The dense layer computes a matrix-vector multiplication in the background. The values used in the matrix are the hyperparameters trainable and updatable by backpropagation. The dense layer can apply operations such as rotation, scaling, and translation. It computes a dimensional vector as output. Important arguments are:

- the `units`, which defines the size of the dense layer output and represents the dimensionality of the output vector
- the `activation` function, which introduces the non-linearity into the NN connections

Here we choose to state two `Dense` layers:

- a hidden layer, with 1000 units, like the size of the previous LSTM layer, and a `ReLU` activation (Algorithm 3.2)
- a final output layer, with 2 units, like the number of classes to identify, and a `softmax` activation (Algorithm 3.2), that converts a values vector in a probability distribution

The output of the LSTM is a pair of probabilities, one for each possible output class. The sum of all the value pairs is always equal to 1. The class with the maximum probability is considered the final choice of the NN. For more details, refer to [GBC16] and the framework documentation [Mar+15; Cho+15].

Loss and Optimizer Functions

Once the model is implemented, it can be configured for the training with losses and metrics with the `compile` function. The function requires to specify, among others arguments, the loss and the optimization functions (both described in subsubsection 2.2.3):

- we use the `categorical_crossentropy` as `loss`, specifically designed for the multiclass classification problem. In this work, we simplify to two classes, so loss is configured as a binary classification

- we originally used `Adam` as `optimizer`, a stochastic gradient descent method based on adaptive estimation of first-order and second-order moments. Then we change it in `SGD`, a stochastic gradient descent with momentum optimizer. With this optimizer, the NN achieves higher accuracy and its performances are less sensitive to other parameters updates

- we use the `accuracy` as `metrics`, which calculates how often predictions equal labels, so in this work measures the $\frac{\text{correct classification}}{\text{total samples}}$ ratio

Loss, optimization and metrics are all difficult mathematical problems that will not be further explored in this study: for a deep analysis, can be considered [GBC16; BN06; Cal20].
Finally, the `build` function puts it all together and builds the model. For more details, refer to the framework documentation [Mar+15; Cho+15].

3.4.4 LSTM Training and Testing Phases

The model is trained with the `fit` function (Algorithm A.1 in appendix).
A training dataset with its label set is required for this function. The NN utilizes the associated entry in the label set for each entry in the dataset to check and improve the accuracy of its prediction. Eventually, its internal state is updated.
The function defines some important hyperparameters:

- `epochs` specifies the number of times the entire training dataset is processed by the model as input

- `batch_size` represents the number of samples per batch of computation, so it sets the samples number to process before updating the internal model state

- `validation_split` instructs the model on the data percentage from the input dataset to use for the validation process, described in subsubsection 2.2.3 (in this case 0.2, so a 20%)

Two-thirds of the original simulated dataset composed by synthetic power consumption traces are dedicated to training and one-third is used for testing.

The model can be evaluated with the `evaluate` function (Algorithm A.2), that returns the loss value and metrics values for the model in test mode.
Finally, the model can be used to make predictions with the `predict` function (Algorithm A.2). For more details, refer to the framework documentation [Mar+15; Cho+15].

3.5 Simulation Results

With the final version of the LSTM architecture, we achieve a testing accuracy of more than 99% on different simulated datasets, each one composed of synthetic power consumption traces described above. Thanks to the simulation environment, we have learned important aspects that we will replicate in the real use case:

- the balancing of the training dataset, in order to avoid underfitting or overfitting problems, with the application of a multiplicative factor, in order to prevent numerical issues for the NN (subsection 3.4.1)
- the window classification and the size values of the window and of the advance step in the sliding window algorithm (subsection 3.4.2)
- the design of the LSTM (subsection 3.4.3), in particular the initial convolutional and pooling layers, useful to extract the most relevant features and to give rotational and positional invariance at the model to train. Also, the use of an SGD optimizer, that maximized the accuracy results obtained

Thus, we can move to the real use case with this settings.

Chapter 4

Experiments

The following chapter exploits the LSTM, explored in the previous chapter, to mount a SCA on a off-the-shelf microcontroller. In this chapter we present the target device (section 4.1), the acquisition tool (section 4.2) and the real power consumption traces captured (section 4.3), with the pre-processing of the training dataset (subsection 4.3.1) and of the testing dataset (subsection 4.3.2).

4.1 Target Device

As previously mentioned (section 2.3), information leaking from unwanted channels allows SCAs. In this work, the considered side channel is the power consumption, which is a characteristic that depends strictly on the selected target. The target device for our experiments is *STM32F415* microcontroller [STM22].
The STM32F415 (`STM32F415RG`) is an embedded system manufactured by *STMicroelectronics*. It is a board equipped, among other things, of an Arm® 32-bit Cortex®-M4 CPU with frequency up to 168 MHz, 192+4 Kbytes of SRAM and 1024 Kbytes of Flash Memory, up to 17 timers, up to 15 communication interfaces, and up to 140 I/O ports with interrupt capability. This board also integrates a crypto-hash processor providing hardware acceleration for cryptographic computation, which is not used in this work. For more details, refer to the official documentation in [STM22].

4.2 Acquisition Device

The acquisition device involved in this work to collect the power consumption traces is the `CW1200` *ChipWhisperer-Pro* [Inc].

ChipWhisperer is an open-source toolchain to perform SCAs, maintained by *NewAE Technology Inc.*. According to the authors, it is easy and affordable, and allows side-channel research in a well documented, cost-effective, and repeatable way. ChipWhisperer is mostly focused on PA attacks and glitching attacks. This utility is natively compatible with STM32 boards and may be used to measure the target power consumption over time. For all the details about ChipWhisperer Pro, that will not be covered in this work, refer to the official documentation in [Inc]. The acquisition of the raw traces is already explained in [Nav20] (5.3.2 subsection) and is not in the scope of this work. The entire acquisition setup remains unchanged and we always work on the previously acquired power consumption traces.

4.3 Real Power Consumption Traces

With an acquisition device (ChipWhisperer-Pro, section 4.2) we collect a few power consumption traces of a target device (STM32F415, section 4.1) running an implementation of ECDSA (micro-ecc, subsection 2.4.2). We use a portion of these traces for LSTM training, while we dedicate the remaining part to testing.
For each power consumption trace, we also collect the custom RNG seed, the ephemeral key k and the output digital signature. This allows a posteriori verification of our results. Each power consumption trace consists of a pair of lists:

- a list of samples, which are the measurements of power consumption
- a list of element pairs, where each pair consists of:
  - a numeric tag identifying the field operation performed (`ADD` and `modADD`, `SUB` and `modSUB`, `MUL`, `SQR` and other routine operations not of interest to us):
    - note that, here additions with and without overflow share the same tag, as well as subtractions with and without underflow
  - the clock ticks counter when the field operation starts

A power consumption trace measures between 7 and 10 million samples approximately. The acquisition sampling rate has been set to `1x` frequency, thus for every tick of the clock, the tool measures one sample of power consumption (one sample per clock tick). With this real power consumption trace structure, we can infer a *ground-truth* of the field operations. We can filter the operations more interesting to us. Also, we can accurately derive the duration of a field operation performed, since we know the clock ticks counter of when it starts and the clock ticks counter of when the next field operation starts. An example of a portion of a real power consumption trace is depicted in Figure 4.1, where the power consumption measurements and the ground-truth of field operations are shown (note that, we add a 20 value to the power consumption measurements, in order to plot a visually effective graph not overlapping with the ground-truth).

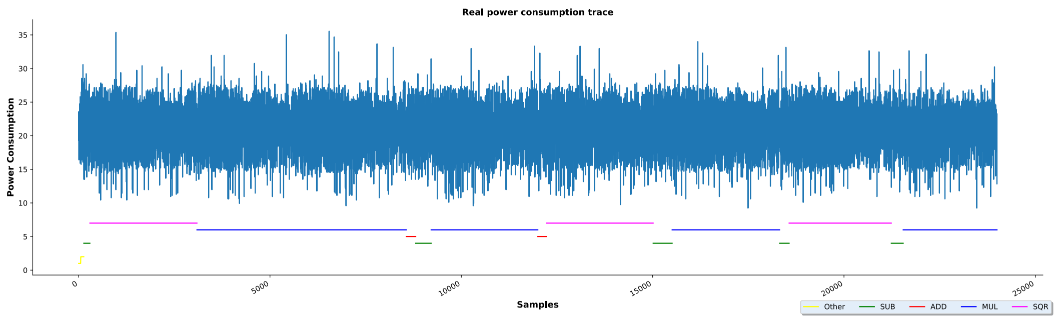


Figure 4.1: The first 24 thousand samples of a real power consumption trace

4.3.1 Pre-processing of the Training Dataset

From one real power consumption trace, we build the training dataset to train the LSTM as follows:

1. since the power consumption measurements were flattened to zero, in order to avoid the NN numerical problems, we multiply them by a constant coefficient (as it worked in the simulation phase subsection 3.4.1). Here we use a 175 multiplicative factor (in order to get a similar magnitude order of the simulation phase)
2. we classify the field operations of a real power consumption trace with a OvR approach (subparagraph 2.2.2):
   (a) the subtraction (`SUB` and `modSUB`) represents the SO class:
      - we choose it because is the most frequent field operation performing modular reduction, with a number of occurrences between about 2500 and just over 3400 (depending on the length of the trace) and an average duration of over 200 samples

   (b) all other operations, except the subtraction, represent the LO class
3. we compute the duration of every single field operation and we append a boolean tag to each operation sample:
   - `true` for a SO sample (i.e., a sample of a subtraction)

- `false` for a LO sample (i.e., a sample of any operation, except subtractions)

4. in order to properly train the NN, we balance the dataset to obtain a similar number of samples of both SO and LO (as it worked in the simulation phase subsection 3.4.1):
   - we implement this by considering the samples number, regardless of the field operations: for SO, we consider the samples from all the subtractions in the trace; while for LO, we consider a similar number of samples from all the other operations
   - for each class, we consider a few more than 540 thousand samples
5. we apply a sliding window algorithm on this formatted samples vector, in order to compute a training dataset
6. then, we perform a classification of the windows, in a identical way of the simulation environment (subsection 3.4.2), in order to compute the label set corresponding to the training dataset:
   - we measure the SO minimum length (in this work, a 153 long subtraction)
   - we tag a window as `true` if it contains this contiguous number of SO samples
   - otherwise, we tag the window as `false`

The resulting input dataset, ready to feed the LSTM, is the training dataset with its label set. During the building process of the training dataset, we conduct several tests on the order of the field operations in the composition of SO and LO classes. We both tried keeping the original order of the field operations, and changing it with a shuffle, according to a Gaussian statistical distribution. Based on the results obtained, we select the former.

4.3.2 Pre-processing of the Testing Dataset

The pre-processing of the testing dataset is almost identical to that of the training dataset, except for the balancing. In this phase, the balance is superfluous. In fact, in order to balance the training dataset, during the pre-processing we applied a filter on the number of samples of the considered field operations. This is no longer the case in the pre-processing of the testing dataset.
We process every power consumption trace selected for testing, thus never seen before by the NN, in its entirety. Briefly, the pre-processing of the testing dataset includes:

1. the scaling via multiplication of the measurements of the real power consumption trace with a constant coefficient (the same one used in the training phase)
2. the computation of the duration of the field operations and the OvR classification

3. the application of the sliding window algorithm, in the same way as the training phase
4. the windows classification with a boolean tag, in the same way as the training phase

The testing dataset size, obtained from a single power consumption trace, is just over 700 thousand windows, each one of 500 samples.

Chapter 5

Results

In the following chapter, we present the results obtained in this work: the raw testing accuracy provided by the LSTM (section 5.1), the application of some generic metrics (section 5.2), and finally the results achieved by the post-processing activity on the LSTM predictions (section 5.3).

5.1 Accuracy Result

We collect four real power consumption traces for testing and, thus, never seen before by the LSTM. For each trace, we observe a few more than 2500 SOs, while just over 3500 LOs. We explore the various dimensionalities of these traces in Table 5.1:

Table 5.1: Sizes of four real power consumption traces, acquired for testing, with derived datasets

Trace	*Trace Size*	*Trace Samples*		*Dataset size*	*Dataset Windows*	
		SO samples	*LO samples*		`true`-*tag*	`false`-*tag*
`T1`	7032251	524697	6507554	703176	86948	616228
`T2`	7002209	522537	6479672	700171	86739	613432
`T3`	7005521	525957	6479564	700503	87086	613417
`T4`	7027186	524247	6502939	702669	86893	615776

1. the first column (*Trace*) identifies a real power consumption trace acquired
2. the second column (*Trace Size*) represents the size of the real power consumption trace considered, expressed in number of samples
3. the third column (*Trace Samples*) contains two sub-columns (*SO samples* and *LO samples*), representing, respectively, the total number of SO samples and of LO samples in the real power consumption trace considered:

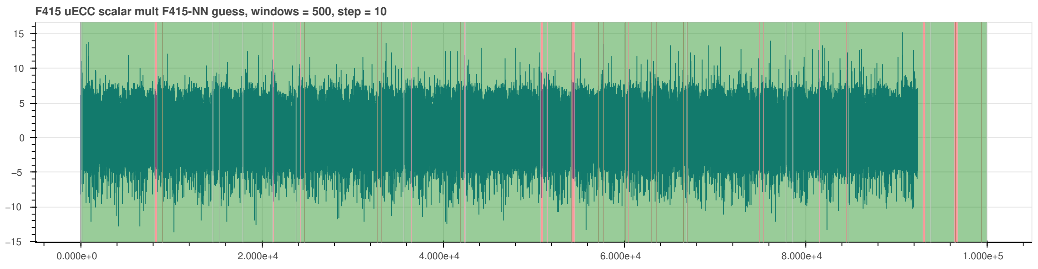


Figure 5.1: LSTM correct guesses (green) and errors (red) on a portion of a real power consumption trace of testing

- row-wise, the sum of the *SO samples* and the *LO samples* cells values returns the previous *Trace Size* cell value

4. the fourth column (*Dataset Size*) represents the size of the testing dataset, derived from the real power consumption trace considered, expressed in number of windows of samples

5. the fifth column (*Dataset Windows)* contains two sub-columns (`true`-*tag* and `false`-*tag*), representing, respectively, the number of `true`-tagged windows and of `false`-tagged windows on the testing dataset considered:

   - row-wise, the sum of the `true`-*tag* and the `false`-*tag* cells values returns the previous *Dataset Size* cell value

From the trained LSTM, for each power consumption trace available, we get a testing accuracy of more than 97%. We consider the testing accuracy as the most significant parameter, since it is a measure that considers the context of the real use case. Note that, we do not perform any post-processing on the raw results of the trained network. In Figure 5.1, we plot the accuracy results for the initial portion of a real power consumption trace, used during the testing phase.

Table 5.2: Accuracy results on four real power consumption traces

Trace	*Predictions*		*Accuracy*	*Loss*
	#right	*#wrong*		
`T1`	686165	17011	0.9758	0.0621
`T2`	683800	16371	0.9766	0.0603
`T3`	683410	17093	0.9756	0.0646
`T4`	683536	19133	0.9728	0.0702

In Table 5.2, we summarize the accuracy results of the trained LSTM obtained on the acquired real power consumption traces:

1. the first column (*Trace*) identifies a real power consumption trace acquired
2. the second column (*Predictions*) contains two sub-columns (*#right* and *#wrong*), representing, respectively, the number of correct and wrong predictions of the LSTM on the testing dataset derived from the real power consumption trace considered
3. the last columns (*Accuracy* and *Loss*) represent the LSTM results of accuracy and loss on the testing dataset

5.2 Similarity Metrics

As far as we know, there is not a standard method to evaluate the result of the trained NN, independently of the testing accuracy score. However, as a first method to evaluate the results of the LSTM, in addition to the testing accuracy score, we use some *similarity metrics*. These methods allow to measure the accuracy of a clustering algorithm [Mei05; VEB10]. Note that, these generic metrics do not take the context into account, so the result may be less accurate than the actual use case.

From the LSTM, we obtain a vector of predictions (*raw predictions*), which we compare with the ground-truth of the real power consumption trace (*raw ground-truth*). The functions we choose to apply here range from 0 (generally, no mutual information) to 1 (generally, perfect correlation), and are as follows:

- the *Variation of Information* (VI), a measure of the distance between two clusterings in terms of the information difference between them [Mei05] (note that, with only this metric low values are better than high)
- the *Normalized Mutual Information* (NMI), a normalization of the *Mutual Information* (MI) score to scale the results between 0 and 1 [VEB10]:
  - MI is a non-negative quantity, considered as the most basic similarity measure
- the *Adjusted Mutual Information* (AMI), an adjustment of the NMI score to account for chance. When two clusterings are identical, AMI $= 1$, while AMI $= 0$ when the MI between the two clusterings equals its predicted value [VEB10]
- *Adjusted Rand Score* (ARS), a *Rand Index* (RI) adjusted for chance:

  - RI computes a similarity score between two clusterings, by examining all pairs of samples and counting pairs assigned in the same or different clusters, in the ground-truth and predicted clusterings [Ped+11]

- *V-measure* (VM) evaluates the *agreement* of two independent assignments on the same dataset [Ped+11]

All these metrics functions are already implemented and taken from the `scikit` module of `Python` [Ped+11; Bui+13]. The results are summarized in Table 5.3.

Table 5.3: Results of similarity metrics to compare raw predictions with raw ground-truth

ID	*VI*	*NMI*	*AMI*	*ARS*	*VM*
`T1`	0.0	0.756	0.756	0.867	0.756
`T2`	0.0	0.757	0.757	0.871	0.757
`T3`	0.0	0.752	0.752	0.866	0.752
`T4`	0.0	0.738	0.738	0.852	0.738

As mentioned above, the applied metrics are generic and do not have knowledge of the context, they do not know the dataset. They are not as accurate as the testing accuracy provided by the trained network, but still return a good result.
The results of every similarity metrics in Table 5.3 confirm the results of our LSTM. But we are interested in something different, so we do some post-processing on raw data.

5.3 Post-processing Results

The trained LSTM is able to predict the presence of a SO (in this work, `SUB` and `modSUB`) in an examined window of real power consumption trace samples. This for all windows of the dataset of an entire real power consumption trace.
We do post-processing on the raw predictions because, in order to mount the attack, we only care about specific windows. Thus, the post-processing aims to enhance certain features to find the useful information.
The goal we want to obtain, through the post-processing, is to be able to accurately estimate the duration of certain short modular operations. We would like to classify a SO as performing modular reduction (so it lasts longer) or not, to understand when it overflows.
In particular, our NN is reliable to generally predict the presence of any SO, but we are just interested in classifying the duration of those fingerprint SOs, as explained in subsection 3.2.1. This in order to be able to retrieve the value of the corresponding bit of the ephemeral key. Thus, with

the aim of mounting the attack presented in subsubsection 2.3.3.

From the vectors of the raw LSTM predictions and of the raw ground-truth of a real power consumption trace, via post-processing, we obtain two vectors:

- the post-processed predictions, containing the number of SO (performing and not performing modular reduction) as *guessed by the network*
- the post-processed ground-truth, containing the number of SO (performing and not performing modular reduction) of the dataset

These two vectors contain the information about the *duration* of the SOs in a single real power consumption trace. Below how we do post-processing. From the raw vector, we count the number of sequential `true`-window predicted by the LSTM: with more than a certain threshold, we classify the SO operation as performing modular reduction, otherwise not.
After a few empirical tries, we set these thresholds:

- 40 and more, for a SO performing modular reduction (we observe that, generally, a subtraction greater than or equal to 40 `true`-tagged windows underflows)
- from 15 up to 40, for a SO not performing modular reduction
- under 15, we consider the result a false positive (we observe that, generally, a sequence less than 15 `true`-tagged windows is not a subtraction)

The post-processing results seem not to be as excellent as in the raw predictions, but there are a few points to consider (remembering that, an SO can be a `SUB` or `modSUB`):

- in the post-processed ground-truth and predictions, the SOs number is lower than in the real power consumption trace. This since, in a trace, multiple sequential SOs can appear, but they are classified as a unique long lasting SO:
  - this case becomes difficult to distinguish from one SO performing modular reduction
- there are *false positives* in the post-processed predictions, with a fine thresholds tuning we will be able to reduce the false positives:
  - note that, the threshold values depend on the single real power consumption trace
- we notice that some errors are *false errors*:

– there are less SOs performing modular reduction (so, more SOs not performing modular reduction) in the post-processed ground-truth than in the post-processed predictions. This is due to the implementation, because a sequence of contiguous SOs in the post-processed ground-truth is always predicted by the LSTM as one SO performing modular reduction in the post-processed predictions. But the SOs positions are well-known in the iterative loop of the ECC scalar multiplication, so we could retrieve this information

Table 5.4: Post-processing results on the four real power consumption traces

Trace	*Post-processed Ground-truth* `(total #SO)`	*Post-proc. Ground-truth SO performing modulus* `#yes`	`#no`	*Post-processed Predictions* `(total #SO)`	*Post-proc. Predictions SO performing modulus* `#yes`	`#no`
`T1`	1770	1209	561	1803	1386	417
`T2`	1770	1188	582	1799	1323	476
`T3`	1770	1223	547	1803	1376	427
`T4`	1770	1214	556	1794	1440	354

We summarize in Table 5.4 the post-processing results of the four previous real power consumption traces:

1. the first column (*Trace*) identifies a real power consumption trace
2. the second column (*Post-processed Ground-truth*) represents the total number of SOs observed in the dataset
3. the third column (*SO performing modulus*) contains two sub-columns (`#yes` and `#no`), representing respectively the number of SOs performing and not modular reduction, according to the post-processed ground-truth:
   - row-wise, the sum of the `#yes` and `#no` cells values returns the previous *Post-processed Ground-truth* cell value
4. the fourth column (*Post-processed Predictions*) represents the total number of predicted SOs by the trained LSTM
5. the fifth column (*SO performing modulus*) contains two sub-columns (`#yes` and `#no`), representing respectively the number of SOs performing and not modular reduction, according to the post-processed predictions:
   - row-wise, the sum of the `#yes` and `#no` cells values returns the previous *Post-processed Predictions* cell value

Chapter 6

Conclusions and Future Works

Powerful SCAs proposed in literature may pose security threats to IoT and embedded systems (section 2.3).
There are secure microcontrollers that offer high security, thanks to standard certifications. This, obviously, has an impact on costs, as well as usability. Meanwhile, the security of consumer devices is secondary to usability and cost, thus providing less secure devices in the end.
In addition, the software libraries used for development may include vulnerabilities that could be exploited by an attacker. This, in particular, for cryptographic software libraries.

In this work, we show how to use a LSTM to mount a specific SCA, based on power consumption leakage of an off-the-shelf microcontroller running a widely used ECDSA implementation. This thesis extends and improves the results achieved in [Nav20], and stems from the work of [Rya19]. The initial project aimed to show how a NN can be trained to break the security of ECC when implemented on a modern board.
We show that, under certain conditions ($Z = 1$, subsubsection 2.4.2), the LSTM correctly identifies the duration of single field operations during the ECC scalar multiplication.
In turn, this allows to retrieve a few bits of the ephemeral key on each digital signature, and thus recover the secret key in a few hundreds signatures with a practical attack (subsubsection 2.3.3). To the best of our knowledge, this is the first time that a LSTM is used in SCAs. The LSTM has been chosen because it performs well in HAR tasks (subsubsection 2.2.5), so we ported this approach to off-the-shelf microcontrollers performing sensitive ECC operations, succeeding in the *Operation Activity Recognition* (OAR).

Note that, the solution presented here is a *Proof of Concept* (PoC) adaptable to mount one, or more, practical attacks to retrieve the secret key of an ECDSA signature. This solution has the

potential to affect all the off-the-shelf microcontrollers where an attacker can perform PA. We plan to publish the results obtained also without assumptions about the projective Z-coordinate, but now it is beyond the scope of this work.
In the future, this work could be extended in several forms. First of all, we would like to evaluate our approach on different environments, with different boards and (or) different ECC implementations. Moreover, since this work is based only on power consumption, it could be interesting to test our approach exploiting different side-channels, even together simultaneously, as a NN can be trained with different features at the same time. In addition, we will search for more precise similarity metrics to describe the results obtained from the LSTM, that allow us to integrate the real use case. Finally, the publication of a scientific paper is planned for the presentation of the results achieved in this work.

Appendix A

Appendix

LSTM pseudo-code

Algorithm A.1: LSTM Training

```
begin
    $model \leftarrow fit(X\_train\_dataset, Y\_train\_dataset, epochs = 50, batch\_size = 64, validation\_split = 0.2)$
end
```

Algorithm A.2: LSTM Testing

```
begin
    $predictions \leftarrow predict(X\_test\_dataset)$
    $model \leftarrow evaluate(X\_test\_dataset, Y\_test\_dataset)$
end
```

Micro-ecc Code

```c
static int uECC_sign_with_k_internal(const uint8_t *private_key,
                            const uint8_t *message_hash,
                            unsigned hash_size,
                            uECC_word_t *k,
                            uint8_t *signature,
                            uECC_Curve curve) {

    uECC_word_t tmp[uECC_MAX_WORDS];
    uECC_word_t s[uECC_MAX_WORDS];
    uECC_word_t *k2[2] = {tmp, s};
```

```c
int uECC_sign(const uint8_t *private_key,
              const uint8_t *message_hash,
              unsigned hash_size,
              uint8_t *signature,
              uECC_Curve curve) {
    uECC_word_t k[uECC_MAX_WORDS];
    uECC_word_t tries;

    for (tries = 0; tries < uECC_RNG_MAX_TRIES; ++tries) {
        if (!uECC_generate_random_int(k, curve->n, BITS_TO_WORDS(curve->num_n_bits))) {
            return 0;
        }

        if (uECC_sign_with_k_internal(private_key, message_hash, hash_size, k,
        ↪   signature, curve)) {
            return 1;
        }
    }
    return 0;
}
```

Listing A.3: uECC_sign

```c
    uECC_word_t *initial_Z = 0;
#if uECC_VLI_NATIVE_LITTLE_ENDIAN
    uECC_word_t *p = (uECC_word_t *)signature;
#else
    uECC_word_t p[uECC_MAX_WORDS * 2];
#endif
    uECC_word_t carry;
    wordcount_t num_words = curve->num_words;
    wordcount_t num_n_words = BITS_TO_WORDS(curve->num_n_bits);
    bitcount_t num_n_bits = curve->num_n_bits;

    /* Make sure 0 < k < curve_n */
    if (uECC_vli_isZero(k, num_words) || uECC_vli_cmp(curve->n, k, num_n_words) != 1) {
        return 0;
    }

    carry = regularize_k(k, tmp, s, curve);
    /* If an RNG function was specified, try to get a random initial Z value to improve
    ↪   protection against side-channel attacks. */
    if (g_rng_function) {
        if (!uECC_generate_random_int(k2[carry], curve->p, num_words)) {
            return 0;
        }
        initial_Z = k2[carry];
    }
    EccPoint_mult(p, curve->G, k2[!carry], initial_Z, num_n_bits + 1, curve);
    if (uECC_vli_isZero(p, num_words)) {
        return 0;
    }

    /* If an RNG function was specified, get a random number to prevent side channel
    ↪  analysis of k. */
    if (!g_rng_function) {
        uECC_vli_clear(tmp, num_n_words);
        tmp[0] = 1;
    } else if (!uECC_generate_random_int(tmp, curve->n, num_n_words)) {
        return 0;
    }

    /* Prevent side channel analysis of uECC_vli_modInv() to determine bits of k / the
    ↪  private key by premultiplying by a random number */
    uECC_vli_modMult(k, k, tmp, curve->n, num_n_words);     /* k' = rand * k */
    uECC_vli_modInv(k, k, curve->n, num_n_words);           /* k = 1 / k' */
    uECC_vli_modMult(k, k, tmp, curve->n, num_n_words);     /* k = 1 / k */

#if uECC_VLI_NATIVE_LITTLE_ENDIAN == 0
    uECC_vli_nativeToBytes(signature, curve->num_bytes, p); /* store r */
#endif

#if uECC_VLI_NATIVE_LITTLE_ENDIAN
    bcopy((uint8_t *) tmp, private_key, BITS_TO_BYTES(curve->num_n_bits));
#else
    /* tmp = d */
    uECC_vli_bytesToNative(tmp, private_key, BITS_TO_BYTES(curve->num_n_bits));
#endif

    s[num_n_words - 1] = 0;
    uECC_vli_set(s, p, num_words);
    uECC_vli_modMult(s, tmp, s, curve->n, num_n_words);     /* s = r*d */

    bits2int(tmp, message_hash, hash_size, curve);
    uECC_vli_modAdd(s, tmp, s, curve->n, num_n_words);      /* s = e + r*d */
    uECC_vli_modMult(s, s, k, curve->n, num_n_words);       /* s = (e + r*d) / k */
    if (uECC_vli_numBits(s, num_n_words) > (bitcount_t)curve->num_bytes * 8) {
        return 0;
    }
#if uECC_VLI_NATIVE_LITTLE_ENDIAN
    bcopy((uint8_t *) signature + curve->num_bytes, (uint8_t *) s, curve->num_bytes);
#else
    uECC_vli_nativeToBytes(signature + curve->num_bytes, curve->num_bytes, s);
#endif
    return 1;
}
```

Listing A.4: `uECC_sign_with_k_internal`

```
static uECC_word_t regularize_k(const uECC_word_t * const k,
                                uECC_word_t *k0,
                                uECC_word_t *k1,
                                uECC_Curve curve) {
    wordcount_t num_n_words = BITS_TO_WORDS(curve->num_n_bits);
    bitcount_t num_n_bits = curve->num_n_bits;
    uECC_word_t carry = uECC_vli_add(k0, k, curve->n, num_n_words) ||
        (num_n_bits < ((bitcount_t)num_n_words * uECC_WORD_SIZE * 8) &&
         uECC_vli_testBit(k0, num_n_bits));
    uECC_vli_add(k1, k0, curve->n, num_n_words);
    return carry;
}
```

Listing A.5: `regularize_k`

```
static void EccPoint_mult(uECC_word_t * result,
                          const uECC_word_t * point,
                          const uECC_word_t * scalar,
                          const uECC_word_t * initial_Z,
                          bitcount_t num_bits,
                          uECC_Curve curve) {
    /* R0 and R1 */
    uECC_word_t Rx[2][uECC_MAX_WORDS];
    uECC_word_t Ry[2][uECC_MAX_WORDS];
    uECC_word_t z[uECC_MAX_WORDS];
    bitcount_t i;
    uECC_word_t nb;
```

```c
    wordcount_t num_words = curve->num_words;

    uECC_vli_set(Rx[1], point, num_words);
    uECC_vli_set(Ry[1], point + num_words, num_words);

    XYcZ_initial_double(Rx[1], Ry[1], Rx[0], Ry[0], initial_Z, curve);

    for (i = num_bits - 2; i > 0; --i) {
        nb = !uECC_vli_testBit(scalar, i);
        XYcZ_addC(Rx[1 - nb], Ry[1 - nb], Rx[nb], Ry[nb], curve);
        XYcZ_add(Rx[nb], Ry[nb], Rx[1 - nb], Ry[1 - nb], curve);
    }

    nb = !uECC_vli_testBit(scalar, 0);
    XYcZ_addC(Rx[1 - nb], Ry[1 - nb], Rx[nb], Ry[nb], curve);

    /* Find final 1/Z value. */
    uECC_vli_modSub(z, Rx[1], Rx[0], curve->p, num_words); /* X1 - X0 */
    uECC_vli_modMult_fast(z, z, Ry[1 - nb], curve);        /* Yb * (X1 - X0) */
    uECC_vli_modMult_fast(z, z, point, curve);             /* xP * Yb * (X1 - X0) */
    uECC_vli_modInv(z, z, curve->p, num_words);       /* 1 / (xP * Yb * (X1 - X0)) */
    /* yP / (xP * Yb * (X1 - X0)) */
    uECC_vli_modMult_fast(z, z, point + num_words, curve);
    uECC_vli_modMult_fast(z, z, Rx[1 - nb], curve);   /* Xb * yP / (xP * Yb * (X1 -
                                                      X0)) */
    /* End 1/Z calculation */

    XYcZ_add(Rx[nb], Ry[nb], Rx[1 - nb], Ry[1 - nb], curve);
    apply_z(Rx[0], Ry[0], z, curve);

    uECC_vli_set(result, Rx[0], num_words);
    uECC_vli_set(result + num_words, Ry[0], num_words);
}
```

Listing A.6: EccPoint_mult

```c
/* Input P = (x1, y1, Z), Q = (x2, y2, Z)
   Output P + Q = (x3, y3, Z3), P - Q = (x3', y3', Z3)
   or P => P - Q, Q => P + Q
*/
static void XYcZ_addC(uECC_word_t * X1,
                      uECC_word_t * Y1,
                      uECC_word_t * X2,
                     uECC_word_t * Y2,
                     uECC_Curve curve) {
    /* t1 = X1, t2 = Y1, t3 = X2, t4 = Y2 */
    uECC_word_t t5[uECC_MAX_WORDS];
    uECC_word_t t6[uECC_MAX_WORDS];
    uECC_word_t t7[uECC_MAX_WORDS];
    wordcount_t num_words = curve->num_words;

    uECC_vli_modSub(t5, X2, X1, curve->p, num_words);   /* t5 = x2 - x1 */
    uECC_vli_modSquare_fast(t5, t5, curve);             /* t5 = (x2 - x1)^2 = A */
    uECC_vli_modMult_fast(X1, X1, t5, curve);           /* t1 = x1*A = B */
    uECC_vli_modMult_fast(X2, X2, t5, curve);           /* t3 = x2*A = C */
    uECC_vli_modAdd(t5, Y2, Y1, curve->p, num_words);   /* t5 = y2 + y1 */
    uECC_vli_modSub(Y2, Y2, Y1, curve->p, num_words);   /* t4 = y2 - y1 */

    uECC_vli_modSub(t6, X2, X1, curve->p, num_words);   /* t6 = C - B */
    uECC_vli_modMult_fast(Y1, Y1, t6, curve);           /* t2 = y1 * (C - B) = E */
    uECC_vli_modAdd(t6, X1, X2, curve->p, num_words);   /* t6 = B + C */
    uECC_vli_modSquare_fast(X2, Y2, curve);             /* t3 = (y2 - y1)^2 = D */
    uECC_vli_modSub(X2, X2, t6, curve->p, num_words);   /* t3 = D - (B + C) = x3 */

    uECC_vli_modSub(t7, X1, X2, curve->p, num_words);   /* t7 = B - x3 */
    uECC_vli_modMult_fast(Y2, Y2, t7, curve);           /* t4 = (y2 - y1)*(B - x3) */
    uECC_vli_modSub(Y2, Y2, Y1, curve->p, num_words);   /* t4 = (y2 - y1)*(B - x3) - E
                                                        = y3 */
    uECC_vli_modSquare_fast(t7, t5, curve);             /* t7 = (y2 + y1)^2 = F */
    uECC_vli_modSub(t7, t7, t6, curve->p, num_words);   /* t7 = F - (B + C) = x3' */
    uECC_vli_modSub(t6, t7, X1, curve->p, num_words);   /* t6 = x3' - B */
    uECC_vli_modMult_fast(t6, t6, t5, curve);           /* t6 = (y2+y1)*(x3' - B) */
    uECC_vli_modSub(Y1, t6, Y1, curve->p, num_words);   /* t2 = (y2+y1)*(x3' - B) - E
                                                        = y3' */
    uECC_vli_set(X1, t7, num_words);
}
```

Listing A.7: XYcZ_addC

```
/* Input P = (x1, y1, Z), Q = (x2, y2, Z)
   Output P' = (x1', y1', Z3), P + Q = (x3, y3, Z3)
   or P => P', Q => P + Q
*/
static void XYcZ_add(uECC_word_t * X1,
                     uECC_word_t * Y1,
                     uECC_word_t * X2,
```

```c
                     uECC_word_t * Y2,
                     uECC_Curve curve) {
    /* t1 = X1, t2 = Y1, t3 = X2, t4 = Y2 */
    uECC_word_t t5[uECC_MAX_WORDS];
    wordcount_t num_words = curve->num_words;

    uECC_vli_modSub(t5, X2, X1, curve->p, num_words);   /* t5 = x2 - x1 */
    uECC_vli_modSquare_fast(t5, t5, curve);             /* t5 = (x2 - x1)^2 = A */
    uECC_vli_modMult_fast(X1, X1, t5, curve);           /* t1 = x1*A = B */
    uECC_vli_modMult_fast(X2, X2, t5, curve);           /* t3 = x2*A = C */
    uECC_vli_modSub(Y2, Y2, Y1, curve->p, num_words);   /* t4 = y2 - y1 */
    uECC_vli_modSquare_fast(t5, Y2, curve);             /* t5 = (y2 - y1)^2 = D */

    uECC_vli_modSub(t5, t5, X1, curve->p, num_words);   /* t5 = D - B */
    uECC_vli_modSub(t5, t5, X2, curve->p, num_words);   /* t5 = D - B - C = x3 */
    uECC_vli_modSub(X2, X2, X1, curve->p, num_words);   /* t3 = C - B */
    uECC_vli_modMult_fast(Y1, Y1, X2, curve);           /* t2 = y1*(C - B) */
    uECC_vli_modSub(X2, X1, t5, curve->p, num_words);   /* t3 = B - x3 */
    uECC_vli_modMult_fast(Y2, Y2, X2, curve);           /* t4 = (y2 - y1)*(B - x3) */
    uECC_vli_modSub(Y2, Y2, Y1, curve->p, num_words);   /* t4 = y3 */

    uECC_vli_set(X2, t5, num_words);
}
```

Listing A.8: `XYcZ_add`

Micro-ecc License [Mac14]

Bibliography

[ABR99] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. "DHAES: An Encryption Scheme based on the Diffie-Hellman Problem". In: *IACR Cryptol. ePrint Arch.* (1999), p. 7.

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. "DHIES: An Encryption Scheme based on the Diffie-Hellman Problem". In: (2001).

[Age16] U.S. National Security Agency. "Commercial National Security Algorithm Suite and Quantum Computing FAQ". In: (Jan. 2016).

[AA18] Afshine Amidi and Shervine Amidi. *Vip Cheatsheet: Recurrent Neural Networks*. 2018.

[Ang+13] Davide Anguita et al. "A Public Domain Dataset for Human Activity Recognition using Smartphones". In: *Proceedings of the 21th International European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. 2013, pp. 437–442.

[Ara+20] Diego F. Aranha et al. "Ladderleak: Breaking ECDSA With Less Than One Bit Of Nonce Leakage". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 225–242.

[Bal+12] Brian Baldwin et al. "Co-Z ECC Scalar Multiplications for Hardware, Software and Hardware–Software co-design on Embedded Systems". In: *Journal of Cryptographic Engineering* 2.4 (2012), pp. 221–240.

[Bar+18] E.B. Barker et al. "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography.(National Institute of Standards and Technology, Gaithersburg, MD)". In: *NIST Special Publication (SP)* (Apr. 2018). DOI: `https://doi.org/10.6028/NIST.SP.800-56Ar3`.

[BM17] Elaine Barker and Nicky Mouha. *Recommendation for the Triple Data Encryption Algorithm (TDEA) block cipher*. Tech. rep. National Institute of Standards and Technology, 2017.

[Bau+13] Aurélie Bauer et al. "Horizontal and Vertical Side-Channel Attacks against Secure RSA Implementations". In: *Cryptographers' Track at the RSA Conference*. Springer. 2013, pp. 1–17.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying Hash Functions for Message Authentication". In: *Annual International Cryptology Conference*. Springer. 1996, pp. 1–15.

[Ben+14] Naomi Benger et al. ""Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2014, pp. 75–92.

[Ben12] Yoshua Bengio. "Deep Learning of Representations for Unsupervised and Transfer Learning". In: *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*. JMLR Workshop and Conference Proceedings. 2012, pp. 17–36.

[Ben09] Yoshua Bengio. *Learning Deep Architectures for AI*. Now Publishers Inc, 2009.

[Ber06] Daniel J. Bernstein. "Curve25519: new Diffie-Hellman Speed Records". In: *International Workshop on Public Key Cryptography*. Springer. 2006, pp. 207–228.

[Ber+12] Daniel J. Bernstein et al. "High-speed High-security Signatures". In: *Journal of Cryptographic Engineering* 2.2 (2012), pp. 77–89.

[Ber+13] Guido Bertoni et al. "Keccak". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2013, pp. 313–314.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller. "Differential Fault Attacks on Elliptic Curve Cryptosystems". In: *Annual International Cryptology Conference*. Springer. 2000, pp. 131–146.

[BN06] Christopher M. Bishop and Nasser M. Nasrabadi. *Pattern Recognition and Machine Learning*. Vol. 4. 4. Springer, 2006.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. "On the Importance of Checking Cryptographic Protocols for Faults". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1997, pp. 37–51.

[BL96] Dan Boneh and Richard J. Lipton. “Algorithms for Black-Box Fields and their Application to Cryptography”. In: *Annual International Cryptology Conference*. Springer. 1996, pp. 283–297.

[Bre01] Leo Breiman. “Random Forests”. In: *Machine Learning* 45.1 (2001), pp. 5–32.

[Bro09] Daniel R.L. Brown. “SEC 1: Elliptic Curve Cryptography”. In: *Certicom Research* 2 (May 2009).

[Bro22] Jason Brownlee. *Machine Learning Mastery*. Making Developers Awesome at Machine Learning. 2022. URL: `https://machinelearningmastery.com/`.

[BHW11] Aiden A. Bruen, James W.P. Hirschfeld, and David L. Wehlau. “Cubic Curves, Finite Geometry and Cryptography”. In: *Acta applicandae mathematicae* 115.3 (2011), pp. 265–278.

[BG15] Anna L. Buczak and Erhan Guven. “A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection”. In: *IEEE Communications surveys & tutorials* 18.2 (2015), pp. 1153–1176.

[Bui+13] Lars Buitinck et al. “API design for Machine Learning Software: Experiences from the scikit-learn Project”. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122.

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. “Convolutional Neural Networks with Data Augmentation against Jitter-based Countermeasures”. In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer. 2017, pp. 45–68.

[Cal20] Ovidiu Calin. *Deep Learning Architectures*. Springer, 2020.

[CHH02] Murray Campbell, A. Joseph Hoane Jr., and Feng-hsiung Hsu. “Deep Blue”. In: *Artificial Intelligence* 134.1-2 (2002), pp. 57–83.

[Car+19] Mathieu Carbone et al. “Deep Learning to Evaluate Secure RSA Implementations”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), pp. 132–161.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. “Template Attacks”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2002, pp. 13–28.

[Cha+99] Suresh Chari et al. “Towards Sound Approaches to Counteract Power-Analysis Attacks”. In: *Annual International Cryptology Conference*. Springer. 1999, pp. 398–412.

[CG16] Tianqi Chen and Carlos Guestrin. “Xgboost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 785–794.

[Cla+10] Christophe Clavier et al. “Horizontal Correlation Analysis on Exponentiation”. In: *International Conference on Information and Communications Security*. Springer. 2010, pp. 46–61.

[Col] Google Colaboratory. *Colab*. URL: `https://colab.research.google.com/`.

[Cor99] Jean-Sébastien Coron. “Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 1999, pp. 292–302.

[Cra+15] Michael Crawford et al. “Survey of Review Spam Detection using Machine Learning Techniques”. In: *Journal of Big Data* 2.1 (2015), pp. 1–24.

[DR99] Joan Daemen and Vincent Rijmen. “AES Proposal: Rijndael”. In: (1999).

[Dan15] Quynh Dang. *Secure Hash Standard*. en. Aug. 2015. DOI: `https://doi.org/10.6028/NIST.FIPS.180-4`.

[Dan+13] Jean-Luc Danger et al. “A Synthesis of Side-Channel Attacks on Elliptic Curve Cryptography in Smart-Cards”. In: *Journal of Cryptographic Engineering* 3.4 (2013), pp. 241–265.

[DP97] Erik De Win and Bart Preneel. “Elliptic Curve Public-Key Cryptosystems - An Introduction”. In: *State of the Art in Applied Cryptography*. Springer. 1997, pp. 131–141.

[Dea16] Jeffrey Dean. “Large-Scale Deep Learning for Building Intelligent Computer Systems”. In: (2016).

[Den12] Li Deng. “Three Classes of Deep Learning Architectures and Their Applications: a tutorial survey”. In: *APSIPA Transactions on Signal and Information Processing* 57 (2012), p. 58.

[DH76] Whitfield Diffie and Martin Hellman. “New Directions in Cryptography”. In: *IEEE transactions on Information Theory* 22.6 (1976), pp. 644–654.

[Dug+16] Margaux Dugardin et al. “Dismantling Real-World ECC with Horizontal and Vertical Template Attacks”. In: *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer. 2016, pp. 88–108.

[Dwo] Morris Dworkin. *Block Cipher Techniques*. URL: `https://csrc.nist.gov/projects/block-cipher-Techniques`.

[Dwo15] Morris Dworkin. *SHA-3 Standard: Permutation-based Hash and Extendable-Output Functions*. en. Aug. 2015. DOI: `https://doi.org/10.6028/NIST.FIPS.202`.

[Dwo+01] Morris Dworkin et al. *Advanced Encryption Standard (AES)*. en. Nov. 2001. DOI: `https://doi.org/10.6028/NIST.FIPS.197`.

[FV12] Junfeng Fan and Ingrid Verbauwhede. “An Updated Survey on Secure ECC Implementations: Attacks, Countermeasures and Cost”. In: *Cryptography and Security: From Theory to Applications*. Springer, 2012, pp. 265–282.

[FGR12] Jean-Charles Faugere, Christopher Goyet, and Guénaël Renault. “Attacking (EC)DSA Given Only an Implicit Hint”. In: *International Conference on Selected Areas in Cryptography*. Springer. 2012, pp. 252–274.

[Fau06] Kjell Magne Fauske. *Example: Neural Network*. Dec. 2006. URL: `https://texample.net/tikz/examples/neural-network/`.

[Fis38] Ronald A. Fisher. “The Statistical Utilization of Multiple Measurements”. In: *Annals of Eugenics* 8.4 (1938), pp. 376–386. DOI: `https://doi.org/10.1111/j.1469-1809.1938.tb02189.x`.

[Fis36] Ronald A. Fisher. “The Use of Multiple Measurements in Taxonomic Problems”. In: *Annals of Eugenics* 7.2 (1936), pp. 179–188. DOI: `https://doi.org/10.1111/j.1469-1809.1936.tb02137.x`.

[Fou+04] Pierre-Alain Fouque et al. “Defeating Countermeasures based on Randomized BSD Representations”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2004, pp. 312–327.

[FS97] Yoav Freund and Robert E. Schapire. "A Decision-Theoretic Generalization of On-line Learning and an Application to Boosting". In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 119–139.

[GHS10] Viíctor Gayoso Martıínez, Luis Hernández Encinas, and Carmen Sánchez Ávila. "A Survey of the Elliptic Curve Integrated Encryption Scheme". In: (2010).

[GST17] Daniel Genkin, Adi Shamir, and Eran Tromer. "Acoustic Cryptanalysis". In: *Journal of Cryptology* 30.2 (2017), pp. 392–443.

[Gir20] Damien Giry. *NIST Report on Cryptographic Key Length*. May 2020. URL: `https://www.keylength.com/en/4`.

[GPG18] Rocio Gonzalez-Diaz, Eduardo Paluzo-Hidalgo, and Miguel A. Gutiérrez-Naranjo. "Representative Datasets for Neural Networks". In: *Electronic Notes in Discrete Mathematics* 68 (2018), pp. 89–94.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[Guo+18] Wenbo Guo et al. "Lemna: Explaining Deep Learning based Security Applications". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 364–379.

[HL15] Rakel Haakegaard and Joanna Lang. "The Elliptic Curve Diffie-Hellman (ECDH)". In: (2015). URL: `https://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Haakegaard+Lang.pdf`.

[HMV06] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Science & Business Media, 2006.

[Hit+02] Yvonne Hitchcock et al. "Implementing an Efficient Elliptic Curve Cryptosystem over $GF(p)$ on a Smart Card". In: *ANZIAM Journal* 44 (2002), pp. C354–C377.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[Hoc+01] Sepp Hochreiter et al. *Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies*. 2001.

[Hos+11] Gabriel Hospodar et al. "Machine Learning in Side-Channel Analysis: a first study". In: *Journal of Cryptographic Engineering* 1.4 (2011), pp. 293–302.

[04] “IEEE Standard Specifications for Public-Key Cryptography - Amendment 1: Additional Techniques”. In: *IEEE Std 1363a-2004 (Amendment to IEEE Std 1363-2000)* (2004), pp. 1–167. DOI: `10.1109/IEEESTD.2004.94612`.

[Inc] NewAE Technology Inc. *CW1200 ChipWhisperer-Pro*. URL: `https://rtfm.newae.com/Capture/ChipWhisperer-Pro/`.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. “Private Circuits: Securing Hardware against Probing Attacks”. In: *Annual International Cryptology Conference*. Springer. 2003, pp. 463–481.

[JK15] H. Jabbar and Rafiqul Zaman Khan. “Methods to Avoid Over-Fitting and Under-Fitting in Supervised Machine Learning (Comparative Study)”. In: *Computer Science, Communication and Instrumentation Devices* 70 (2015).

[Jea16] Jérémy Jean. *TikZ for Cryptographers*. `https://www.iacr.org/authors/tikz/`. 2016.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. “The Elliptic Curve Digital Signature Algorithm (ECDSA)”. In: *International journal of Information Security* 1.1 (2001), pp. 36–63.

[Jou+17] Norman P. Jouppi et al. “In-datacenter Performance Analysis of a Tensor Processing Unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 1–12.

[JM97] Aleksandar Jurišic and A. Menezes. “Elliptic Curves and Cryptography”. In: *Dr. Dobb’s Journal* (1997), pp. 26–36.

[Kab+19] Ievgen Kabin et al. “Horizontal Attacks against ECC: from Simulations to ASIC”. In: *Computer Security*. Springer, 2019, pp. 64–76.

[Ant+10] Jeff Moser Anthony Goldbloom Ben Hamner et al. *Kaggle*. 2010. URL: `https://www.kaggle.com`.

[KMD20] John D. Kelleher, Brian Mac Namee, and Aoife D’arcy. *Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies*. MIT Press, 2020.

[KCP16] John Kelsey, Shu-jen Chang, and Ray Perlner. “SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash”. In: *NIST Special Publication* 800 (2016), p. 185.

[Cho+15] François Chollet et al. *Keras*. 2015. URL: `https://keras.io`.

[KG13] Cameron F. Kerry and Patrick D. Gallagher. “Digital Signature Standard (DSS)”. In: *FIPS PUB* (2013), pp. 186–4.

[KSD13] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. “FIPS PUB 186-4 FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION Digital Signature Standard (DSS)”. In: (2013).

[Kha+20] Nafiz Imtiaz Khan et al. “Prediction of Cesarean Childbirth using Ensemble Machine Learning Methods”. In: *Proceedings of the 22nd International Conference on Information Integration and Web-based Applications & Services*. 2020, pp. 331–339.

[Kim+19] Jaehun Kim et al. “Make Some Noise. Unleashing the Power of Convolutional Neural Networks for Profiled Side-Channel Analysis”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), pp. 148–179.

[Klu+16] Thomas Kluyver et al. “Jupyter Notebooks ? A Publishing Format for Reproducible Computational Workflows”. In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by Fernando Loizides and Birgit Scmidt. IOS Press, 2016, pp. 87–90. URL: `https://eprints.soton.ac.uk/403913/`.

[Kob87] Neal Koblitz. “Elliptic Curve Cryptosystems”. In: *Mathematics of Computation* 48.177 (1987), pp. 203–209.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *Annual International Cryptology Conference*. Springer. 1999, pp. 388–397.

[Koc+11] Paul Kocher et al. “Introduction to Differential Power Analysis”. In: *Journal of Cryptographic Engineering* 1.1 (2011), pp. 5–27.

[Koc96] Paul C. Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Annual International Cryptology Conference*. Springer. 1996, pp. 104–113.

[LS08] Kristin E. Lauter and Katherine E. Stange. “The Elliptic Curve Discrete Logarithm Problem and Equivalent Hard Problems for Elliptic Divisibility Sequences”. In: *International Workshop on Selected Areas in Cryptography*. Springer. 2008, pp. 309–327.

[Law+03] Laurie Law et al. “An Efficient Protocol for Authenticated Key Agreement”. In: *Designs, Codes and Cryptography* 28.2 (2003), pp. 119–134.

[LB+95] Yann LeCun, Yoshua Bengio, et al. “Convolutional Networks for Images, Speech, and Time Series”. In: *The Handbook of Brain Theory and Neural Networks* 3361.10 (1995), p. 1995.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep Learning”. In: *Nature* 521.7553 (2015), pp. 436–444.

[LKT13] Arjen K. Lenstra, Thorsten Kleinjung, and Emmanuel Thomé. “Universal Security”. In: *Number Theory and Cryptography*. Springer, 2013, pp. 121–124.

[LLL82] Arjen K. Lenstra, Hendrik Willem Lenstra, and László Lovász. “Factoring Polynomials with Rational Coefficients”. In: *Mathematische Annalen* 261.ARTICLE (1982), pp. 515–534.

[LD00] Julio Lopez and Ricardo Dahab. “An Overview of Elliptic Curve Cryptography”. In: (2000).

[Mac14] Ken MacKay. *micro-ecc: ECDH and ECDSA for 8-bit, 32-bit, and 64-bit Processors*. 2014. URL: `https://github.com/kmackay/micro-ecc`.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. “Breaking Cryptographic Implementations using Deep Learning Techniques”. In: *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer. 2016, pp. 3–26.

[Mag16] Krists Magons. “Applications and Benefits of Elliptic Curve Cryptography”. In: *SOFSEM (Student Research Forum Papers/Posters)*. 2016, pp. 32–42.

[MOP08] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Vol. 31. Springer Science & Business Media, 2008.

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org`.

[ME+10] V. Gayoso Martínez, L. Hernández Encinas, et al. "A Comparison of the Standardized Versions of ECIES". In: *2010 Sixth International Conference on Information Assurance and Security*. IEEE. 2010, pp. 1–4.

[McC07] John McCarthy. "What is Artificial Intelligence?" In: (2007).

[MP43] Warren S. McCulloch and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133.

[McC90] Kevin S. McCurley. "The Discrete Logarithm Problem". In: *Proc. of Symp. in Applied Math*. Vol. 42. USA. 1990, pp. 49–74.

[Mei05] Marina Meilă. "Comparing Clusterings: an Axiomatic View". In: *Proceedings of the 22nd International Conference on Machine Learning*. 2005, pp. 577–584.

[MOP] A.J. Menezes, Paul C. van Oorschot, and Vanstone P.C. *SA (1996). Handbook of applied Cryptography, CRC Press, ISBN: 0-8493-8523-7.*

[Mit+97] Tom M. Mitchell et al. "Machine Learning". In: *Burr Ridge, IL: McGraw Hill* 45.37 (1997), pp. 870–877.

[Mon87] Peter L. Montgomery. "Speeding the Pollard and Elliptic Curve Methods of Factorization". In: *Mathematics of Computation* 48.177 (1987), pp. 243–264.

[Moz95] Michael C. Mozer. "A Focused Backpropagation Algorithm for Temporal". In: *Backpropagation: Theory, Architectures, and Applications* 137 (1995).

[Muk+18] Naila Mukhtar et al. "Machine-Learning-based Side-Channel Evaluation of Elliptic-Curve Cryptographic FPGA Processor". In: *Applied Sciences* 9.1 (2018), p. 64.

[NC17] Erick Nascimento and Łukasz Chmielewski. "Applying Horizontal Clustering Side-Channel Attacks on Embedded ECC Implementations". In: *International Conference on Smart Card Research and Advanced Applications*. Springer. 2017, pp. 213–231.

[Nav20] Lorenzo Nava. "Side-Channel Attack on ECC Cryptosystem by using Neural Networks". MA thesis. Università degli Studi di Milano, Dec. 2020.

[NAY19] Van Quan Nguyen, Tien Nguyen Anh, and Hyung-Jeong Yang. “Real-Time Event Detection using Recurrent Neural Network in Social Sensors”. In: *International Journal of Distributed Sensor Networks* 15.6 (2019), p. 1550147719856492.

[Nis92] Corporate Nist. “The Digital Signature Standard”. In: *Communications of the ACM* 35.7 (1992), pp. 36–40.

[NRR17] Ida Nurhaida, Desi Ramayanti, and Rhema Riesaputra. “Digital Signature & Encryption Implementation for Increasing Authentication, Integrity, Security and Data Non-Repudiation”. In: *Int. Res. J. Comput. Sci* 4.11 (2017), pp. 4–14.

[Ora14] Miloš Oravec. “Feature Extraction and Classification by Machine Learning Methods for Biometric Recognition of Face and Iris”. In: *Proceedings ELMAR-2014*. IEEE. 2014, pp. 1–4.

[OR16] Francisco Javier Ordóñez and Daniel Roggen. “Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition”. In: *Sensors* 16.1 (2016), p. 115.

[PMB13] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the Difficulty of Training Recurrent Neural Networks”. In: *International Conference on Machine Learning*. PMLR. 2013, pp. 1310–1318.

[Ped+11] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[PW08] Wouter Penard and Tim van Werkhoven. “On The Secure Hash Algorithm Family”. In: *Cryptography in Context* (2008), pp. 1–18.

[Per+21] Guilherme Perin et al. “Keep It Unsupervised: Horizontal Attacks Meet Deep Learning”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2021), pp. 343–372.

[Pic+17] Stjepan Picek et al. “Side-Channel Analysis and Machine Learning: A Practical Perspective”. In: *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2017, pp. 4095–4102.

[Pic+21] Stjepan Picek et al. “SoK: Deep Learning-based Physical Side-Channel Analysis”. In: *Cryptology ePrint Archive* (2021).

[PR13] Emmanuel Prouff and Matthieu Rivain. “Masking against Side-Channel Attacks: A Formal Security Proof”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2013, pp. 142–159.

[QS01] Jean-Jacques Quisquater and David Samyde. “Electromagnetic Analysis (EMA): Measures and Countermeasures for Smart Cards”. In: *International Conference on Research in Smart Cards*. Springer. 2001, pp. 200–210.

[RF19] Muhammad Mahbubur Rahman and Tim Finin. “Unfolding the Structure of a Document using Deep Learning”. In: *arXiv preprint arXiv:1910.03678* (2019).

[RCB16] Joost Renes, Craig Costello, and Lejla Batina. “Complete Addition Formulas for Prime Order Elliptic Curves”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2016, pp. 403–428.

[Riv11] Matthieu Rivain. *Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves*. Cryptology ePrint Archive, Paper 2011/338. `https://eprint.iacr.org/2011/338`. 2011. URL: `https://eprint.iacr.org/2011/338`.

[RPD09] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. “Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2009, pp. 171–188.

[Riv+83] R. Rivest et al. “Cryptographic Communications System and Method, US 4405829 A”. In: (1983).

[RSA19] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*. Routledge, 2019.

[Roc+21] Thomas Roche et al. “A Side Journey to Titan”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 231–248.

[RKA06] Juan José Rodriguez, Ludmila I. Kuncheva, and Carlos J. Alonso. “Rotation Forest: a new Classifier Ensemble Method”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.10 (2006), pp. 1619–1630.

[Ros58] Frank Rosenblatt. “The Perceptron: a Probabilistic Model for Information Storage and Organization in the Brain”. In: *Psychological Review* 65.6 (1958), p. 386.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning Representations by Back-Propagating Errors”. In: *Nature* 323.6088 (1986), pp. 533–536.

[RN02] Stuart Russell and Peter Norvig. “Artificial Intelligence: a modern approach”. In: (2002).

[Rya19] Keegan Ryan. “Return of the Hidden Number Problem.: A Widespread and Novel Key Extraction Attack on ECDSA and DSA”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), pp. 146–168.

[The21] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.3)*. `https://www.sagemath.org`. 2021.

[Sri+14] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html`.

[Sta95] Secure Hash Standard. “FIPS Pub 180-1”. In: *National Institute of Standards and Technology* 17.180 (1995), p. 15.

[STM] STMicroelectronics. *STM32CubeIDE*. `https://www.st.com/en/development-tools/stm32cubeide.html`.

[STM22] STMicroelectronics. *STMicroelectronics STM32F405/415*. 2022. URL: `https://www.st.com/en/microcontrollers-microprocessors/stm32f405-415.html`.

[Tho83] Thomas M. Thompson. *From Error-Correcting Codes through Sphere Packings to Simple Groups*. Vol. 21. American Mathematical Soc., 1983.

[Tim19] Benjamin Timon. “Non-profiled Deep Learning-based Side-Channel Attacks with Sensitivity Analysis”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), pp. 107–131.

[TW05] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory*. 2005.

[Tur09] Alan M. Turing. “Computing Machinery and Intelligence”. In: *Parsing the Turing Test*. Springer, 2009, pp. 23–65.

[Tur08] James M. Turner. “The Keyed-Hash Message Authentication Code (HMAC)”. In: *Federal Information Processing Standards Publication* 198.1 (2008).

[Van92] Scott Vanstone. “Responses to NIST’s Proposal”. In: *Communications of the ACM* 35.7 (1992), pp. 50–52.

[Vap99] Vladimir Vapnik. *The Nature of Statistical Learning Theory*. Springer Science & Business Media, 1999.

[VBB16] Augusto Vega, Pradip Bose, and Alper Buyuktosunoglu. *Rugged Embedded Systems: Computing in Harsh Environments*. Morgan Kaufmann, 2016.

[VBP18] Alakananda Vempala, Eduardo Blanco, and Alexis Palmer. “Determining Event Durations: Models and Error Analysis”. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. 2018, pp. 164–168.

[VS11] Nicolas Veyrat-Charvillon and François-Xavier Standaert. “Generic Side-Channel Distinguishers: Improvements and Limitations”. In: *Annual Cryptology Conference*. Springer. 2011, pp. 354–372.

[VEB10] Nguyen Xuan Vinh, Julien Epps, and James Bailey. “Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance”. In: *The Journal of Machine Learning Research* 11 (2010), pp. 2837–2854.

[Wal01] Colin D. Walter. “Sliding Windows Succumbs to Big Mac Attack”. In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2001, pp. 286–299.

[Wan+14] Jiang Wang et al. “Learning Fine-grained Image Similarity with Deep Ranking”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 1386–1393.

[WPB19] Leo Weissbart, Stjepan Picek, and Lejla Batina. “One Trace Is All It Takes: Machine Learning-based Side-Channel Attack on EdDSA”. In: *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer. 2019, pp. 86–105.

[Wei+20] Leo Weissbart et al. “Systematic Side-Channel Analysis of Curve25519 with Machine Learning”. In: *Journal of Hardware and Systems Security* 4.4 (2020), pp. 314–328.

[WPP21] Lichao Wu, Guilherme Perin, and Stjepan Picek. “The Best of Two Worlds: Deep Learning-assisted Template Attack”. In: *Cryptology ePrint Archive* (2021).

[Zai+20] Gabriel Zaid et al. “Methodology for Efficient CNN Architectures in Profiling Attacks”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.1 (2020), pp. 1–36.

[Zel03] Andreas Zell. *Simulation Neuronaler Netze*. Vol. 1. 5.2. Addison-Wesley, 2003.

[Zha19] Jiawei Zhang. *Basic Neural Units of the Brain: Neurons, Synapses and Action Potential*. 2019. DOI: `10.48550/ARXIV.1906.01703`. URL: `https://arxiv.org/abs/1906.01703`.

[ZL99] Jianying Zhou and Kwok-Yan Lam. “Securing Digital Signatures for Non-Repudiation”. In: *Computer Communications* 22.8 (1999), pp. 710–716.

Acknowledgments

First of all, I want to thank my advisor Prof. Danilo Bruschi, and my co-advisor Eng. Guido Bertoni. Then, I wish to thank *Security Pattern* for the opportunity of this work, especially Ph.D. Alberto Battistello and Lorenzo Nava for all the support they have given me during this period, in which I had the pleasure of working with them: it was fun collaborating together! I also would like to thank Prof. Andrea Lanzi and Prof. Fabio Pierazzi for all the tips and suggestions they provided to me.

I have to thank my family for their support. My mother and sister are the most important people I have in this world. Together we overcame many obstacles, I hope better times will come now.
Giorgia, *if you ever get hurt and you feel that you are going down, this little angel is gonna whisper in your ear. It is gonna say: Get up, ’cause Micio loves you. Okay?*
I just wanna say one thing to my mom … Yo mom, I did it!
Lucky me to have the little Olimpia in my life. I want to remember my beloved Rocky too: I know that you are always with me. I am grateful that I still have grandparents who love me, even if they are far away. I wish my uncle not to stop fighting, as mom did. I am with you.

Finally, I want to thank some dear friends, whom I was lucky enough to meet on my way. The wise stone Gian, Marco, Clio, Ferra, Altin, Albi, Teo, Ale. Thanks to all the colleagues of the *Laboratory of Information and Network Security* (LaSER). Together we have collaborated on several projects. You are more than a simple laboratory to me.
Thanks to all of you.

> *I am not the richest, smartest, or most talented person in the world, but I keep going, and going, and going!*