Evaluation and Implementation of the Ed25519 Digital Signature Algorithm in Rust

Eduardo Yuzo Nakai

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Informatics, in the scope of the double diploma programme with the Federal University of Technology - Paraná.

Work oriented by:

Professor PhD José Exposto

Professor Dr. Rodrigo Campiolo

Professor PhD Tiago Pedrosa

Bragança

2020-2021

Evaluation and Implementation of the Ed25519 Digital Signature Algorithm in Rust

Eduardo Yuzo Nakai

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master Degree in Informatics, in the scope of the double diploma programme with the Federal University of Technology - Paraná.

Work oriented by:

Professor PhD José Exposto

Professor Dr. Rodrigo Campiolo

Professor PhD Tiago Pedrosa

Bragança

2020-2021

Acknowledgement

I would like to thank my advisors Rodrigo Campiolo, José Exposto and Tiago Pedrosa for letting me explore cryptography and try applying my own ideas. Thank you for the help and support during these two years.

Abstract

Cryptography can be classified as secret-key and public-key cryptography. Both have distinct features and differs in performance, complexity, flexibility and security. Secret-key cryptography, considering its simplicity and performance, is commonly used for securing communications. The invention of public-key cryptography made it possible to develop more flexible cryptographic schemes and algorithms, such as key exchanges and digital signatures, hence extending the possibilities and the field of cryptography. Cryptographic implementations are primordial for securing the Internet, and as a consequence, correctness, security and efficiency are more emphasized. In this sense, this work addresses the evaluation and the implementation of Ed25519, an instance of the Edwards-curve Digital Signature Algorithm for digital messages authentication. The implementation lies on Rust: a safe, modern, high-level and strongly-typed programming language. This work has two contributions: (i) an Ed25519 implementation in Rust that considers readability, modularity and ease of use, and (ii) an evaluation of the Ed25519 implementation from a security/performance perspective. The implementation was comprised by three modules: field arithmetic, curve arithmetic and the interface. The security perspective presented essential qualities of cryptographic implementations, such as functional correctness, memory safety, constant-time operations and usability. The performance evaluation showed low execution times and proved to be as fast as implementations written in C; Rust's RAM consumption showed similar results in comparison to implementations written in C.

Keywords: Edwards-curve Digital Signature Algorithm; Elliptic Curve Cryptography; cryptographic library; security analysis; performance evaluation.

Resumo

A criptografia pode ser classificada em simétrica e assimétrica. Ambos diferem-se com relação ao desempenho, complexidade, flexibilidade e segurança. A criptografia simétrica é comumente utilizada para estabelecer comunicações secretas. A criptografia assimétrica foi capaz de ampliar as primitivas criptográficas ao possibilitar primitivas para assinaturas digitais e acordo de chaves. Implementações criptográficas são primordiais na proteção de comunicações e informações. Nesse sentido, a segurança, corretude e eficiência são priorizadas. Este trabalho aborda a avaliação e implementação do algoritmo de assinaturas digitais Ed25519, uma instância do *Edwards-curve Digital Signature Algorithm*. Este trabalho tem como objetivo realizar a implementação do algoritmo Ed25519 em Rust, uma linguagem de programação de alto nível, moderna e fortemente tipada, com foco em *memory safety*. Dessa forma, as contribuições são: (i) uma implementação do Ed25519 em Rust que suporta legibilidade, modularidade e facilidade de uso e (ii) a avaliação da implementação em termos de desempenho e segurança. O desenvolvimento envolveu a implementação de três módulos: aritmética de corpos, aritmética de curvas e interface. Na perspectiva da segurança, foi possível apresentar atributos essenciais às implementações criptográficas, tais como *correctness*, *memory safety*, usabilidade e operações em tempo constante. A implementação apresentou tempos de execução aceitáveis e próximos das implementações em C; além disso, Rust apresentou consumo de memória RAM similar às implementações em C.

Keywords: *Edwards-curve Digital Signature Algorithm*; Criptografia de Curva Elíptica; biblioteca criptográfica; análise de segurança; análise de desempenho.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The topic of this dissertation is the implementation of a cryptographic library. More specifically the implementation of the Ed25519 digital signature algorithm (introduced by Bernstein et al. [1]). We named our implementation *ed25519-fun*.

In this context, Elliptic Curve Cryptography (ECC) is a public-key cryptosystem proposed independently by Neal Koblitz and Victor Miller in 1985. The underlying mathematics revolves around elliptic curves and its intractability is based on the Elliptic Curve Discrete Logarithm Problem [2]. The Ed25519 system is classified as ECC and is based on the Curve25519 as twisted Edwards curve, providing fast operations and side-channel resilience [1].

Taking a step back, cryptography can be understood as the study of mathematical techniques to secure communications [3]. This is primarily accomplished by creating libraries that implements cryptographic primitives. These libraries lie at the core of the Internet, and consequently, as well as ensuring their correctness and safety [4], the quality of the implementation is also tied with efficiency [5].

Brumley et al. [5] illustrated the difficulty of implementing cryptographic primitives. In parallel, the authors showed how an arithmetic bug could be exploited to mount a full key recovery attack on the Elliptic-curve Diffie-Hellman protocol. Lazar et al. [6] analyzed 269 CVEs (from January 2011 to May 2014) tagged as “Cryptographic Issues” and classified them as plaintext disclosure, man-in-the-middle attacks, brute-force attacks and side-channel attacks, 83% of which were related to the misuse of cryptographic libraries and 17% were tied to implementation bugs. Since 2018, OpenSSL issued 22 CVEs[1], which enabled attacks classified as side-channel attacks, denial of service attacks, padding oracle attacks.

Hence, this work dives into the properties and conditions that cryptographic libraries rely on. We chose to implement the Ed25519 algorithm in Rust - a modern programming language that aims for safety and performance. As a result, we were able to document the implementation and explore aspects of performance and security that surrounds the Ed25519 algorithm and its implementation in Rust.

1.1 Objective

Our objective is the evaluation and implementation of the Ed25519 signature system in Rust. Therefore, we focus on the security and performance analysis and attempt to justify the use of Rust for the Ed25519 algorithm implementation. We accomplish this within three main tasks:

- Implementation of the Ed25519 algorithm in Rust considering *good programming practices.*
- *Performance evaluation* comparing the execution times and memory consumption of various Ed25519 implementations.
- Evaluation of the Ed25519 algorithm and its implementation in Rust from a *security perspective.*

1.2 Focus

This work has been directed towards the objective presented in Section 1.1 in the following ways:

[1]https://www.openssl.org/news/vulnerabilities.html

- This work is about the comprehension and implementation of the Ed25519 algorithm in Rust. The implementation is simple and has been influenced by existing Ed25519 implementations, namely the *ed25519-donna* [7], *ed25519-dalek* [8] and *ed25519-java* [9] implementations.

- This work is about emphasizing the properties of the Ed25519 system. Along with its properties lies the explanation of the mathematical techniques behind the execution of the Ed25519 primitives.

- This work is about the analysis of our Ed25519 implementation. This analysis is twofold. Firstly, an empirical study is performed. Secondly, the relationship between Rust and secure and fast applications, as well as performance and security details of the Ed25519 signature system.

- The success of this work is contingent on the success of three factors. The first factor is the successful execution of the regression tests performed by our Ed25519 implementation in Rust. The second is a comprehensible writing about the core functions and properties of the implementation and its mathematical techniques. And lastly, an empirical analysis and the study of Rust features, performance and security must be presented.

- This work is not about the design of a novel implementation of the Ed25519 algorithm.

Within the analysis of Ed25519‘s performance and security properties and the study of its mathematical perspective and its development in Rust, this work focuses on delivering a more comprehensible document that comprises some advanced cryptographic topics.

1.3 Contributions

Our main contributions are defined by:

- An Ed25519 implementation that prioritizes good programming practices, such as readability, modularity and ease of use.
- A scientific evaluation of our Ed25519 implementation in Rust, from a performance/security perspective.

1.4 Structure

The remainder of this dissertation is organized as follows. Chapter 2 describes the theoretical background needed for this work. Chapter 3 emphasizes important works related to the implementation and analysis of the Ed25519 public-key signature system. Chapter 4 presents the methodology used for the Ed25519 implementation and the approach taken for the description of Ed25519's properties and its evaluation. Chapter 5 describes the Ed25519 signature system implementation. Chapter 6 presents the evaluation of the Ed25519 implementation in two perspectives: performance and security. Chapter 7 shows the final considerations.

Chapter 2

Background

The purpose of this chapter is to introduce secret-key cryptography and public-key cryptography, and also to explore elliptic curves, specially the twisted edwards curves, and the Rust programming language.

We begin in Section 2.1 with an introduction to the Rust programming language. Then, we explain secret-key and public-key cryptography in Section 2.2. The theory for elliptic curves can be found in Section 2.3. Lastly, Section 2.4 introduces the Ed25519 signature system. Section 2.5 discusses the main ideas presented in this chapter.

2.1 Rust

Rust is a programming language developed at Mozilla Research that offers performance and safety features. Rust does not depend on a garbage collector, and supports zero-cost abstractions[1], type-safety and memory safety, as well as low-level control [11]. Rust automatically runs a *destructor* when a variable goes out of scope [11] and does not use garbage collection. Instead, ownership tracking allows Rust to know when a resource goes out of scope, and thus memory leaks are avoided.

During execution, resources are uniquely owned by one alias at a time. C++ enforces

[1]Rust does not use more CPU, RAM, or code space for tracking type states [10].

ownership with smart pointers, move semantics and RAII (Resource Acquisition Is Initialization), however, undefined/unsafe behavior is still possible [11]. Rust's ownership discipline, which states that "if ownership of an object (of type T) is shared between multiple aliases (shared references of type &T), then none of them can be used to directly mutate it", prevents use-after-free, data races and iterator validation [11]. Rust's unique pointers manages ownership tracking of heap memory, and borrowed references manages safe aliases [12].

Listings 2.1, 2.2 and 2.3 illustrates simple code examples for *ownership*, *borrowing* and *lifetimes*.

Listing 2.1: Ownership. Source: Rust By Example [13].

```rust
// Resources can only have one owner.
// Ownership can be transferred. If that happens, the previous owner
// can no longer be used.

// This function takes ownership of the heap allocated memory
fn destroy_box(c: Box<i32>) {
    println!("Destroying a box that contains {}", c);
    // `c` is destroyed and the memory freed
}
fn main() {
    // _Stack_ allocated integer
    let x = 5u32;
    // *Copy* `x` into `y` - no resources are moved
    let y = x;
    // Both values can be independently used
    println!("x is {}, and y is {}", x, y);
    // `a` is a pointer to a _heap_ allocated integer
    let a = Box::new(5i32);
    println!("a contains: {}", a);
    // *Move* `a` into `b`
    let b = a;
    // The pointer address of `a` is copied (not the data) into `b`.
    // Both are now pointers to the same heap allocated data, but
    // `b` now *owns* it.

    // !Error! `a` can no longer access the data, because it no longer
    // owns the heap memory
    //? println!("a contains: {}", a);

    // This function takes ownership of the heap allocated memory from `b`
    destroy_box(b);

    // Since the heap memory has been freed at this point, this action would
    // result in dereferencing freed memory, but it's forbidden by the compiler
    // !Error! Same reason as the previous error
    //? println!("b contains: {}", b);
}
```

Listing 2.2: Borrowing. Source: Rust By Example [13].

```
// Access the data without taking ownership over it.

// This function takes ownership of a box and destroys it
fn eat_box_i32(boxed_i32: Box<i32>) {
    println!("Destroying box that contains {}", boxed_i32);
}
// This function borrows an i32
fn borrow_i32(borrowed_i32: &i32) {
    println!("This int is: {}", borrowed_i32);
}
fn main() {
    // Create a boxed i32, and a stacked i32
    let boxed_i32 = Box::new(5_i32);
    let stacked_i32 = 6_i32;
    // Borrow the contents of the box. Ownership is not taken,
    // so the contents can be borrowed again.
    borrow_i32(&boxed_i32);
    borrow_i32(&stacked_i32);
    // `boxed_i32` can now give up ownership to `eat_box` and be destroyed
    eat_box_i32(boxed_i32);
}
```

Listing 2.3: Lifetimes. Source: Rust By Example [13].

```
// Lifetimes are annotated below with lines denoting the creation
// and destruction of each variable.
// `i` has the longest lifetime because its scope entirely encloses
// both `borrow1` and `borrow2`. The duration of `borrow1` compared
// to `borrow2` is irrelevant since they are disjoint.
fn main() {
    let i = 3; // Lifetime for `i` starts.
    {
        let borrow1 = &i; // `borrow1` lifetime starts.
        println!("borrow1: {}", borrow1);
    } // `borrow1 ends.
    {
        let borrow2 = &i; // `borrow2` lifetime starts.
        println!("borrow2: {}", borrow2);
    } // `borrow2` ends.
}   // Lifetime ends.
```

2.2 Cryptography

Cryptography is, as defined by Bruce Schneier, “the art and science of keeping messages secure”. Meanwhile, cryptanalysis focuses on trying to breach the system [14]. Both complements each other and creates an adversarial way of thinking: there is a defender and an attacker. When building cryptographic algorithms or cryptographic protocols we must consider all the possible ways we could break the system [3].

Modern cryptography can be divided into two main categories: secret-key and public-key cryptography. Prior to the invention of public-key cryptography, communication channels were secured using secret-key cryptosystems such as the Data Encryption Standard [3]. They are classified as such for performing encryption and decryption with the same key. Public-key cryptosystems is an elegant concept invented by Diffie and Helman in 1976 [15]. The understanding that it is possible to form a secure communication without having to share private information is astounding. Such concept opened a door that housed a wide variety of possibilities for the expansion of the cryptographic scientific literature. The RSA and ElGamal systems are instances of public-key cryptosystems. Contrasting secret-key cryptography, they generate key pairs: a public key and a private key. They can be used to perform digital signatures, key exchange and encryption.

Figure 2.1 illustrates cryptographic primitives, which are related to aspects of information security: confidentiality, data integrity, authentication and non-repudiation. They can be evaluated in various dimensions: security level, desired functionality and security property, methods of operation, performance and implementation complexity. Such dimensions vary in terms of the security goals of the application, available computational resources and trade-off between performance and security [14].

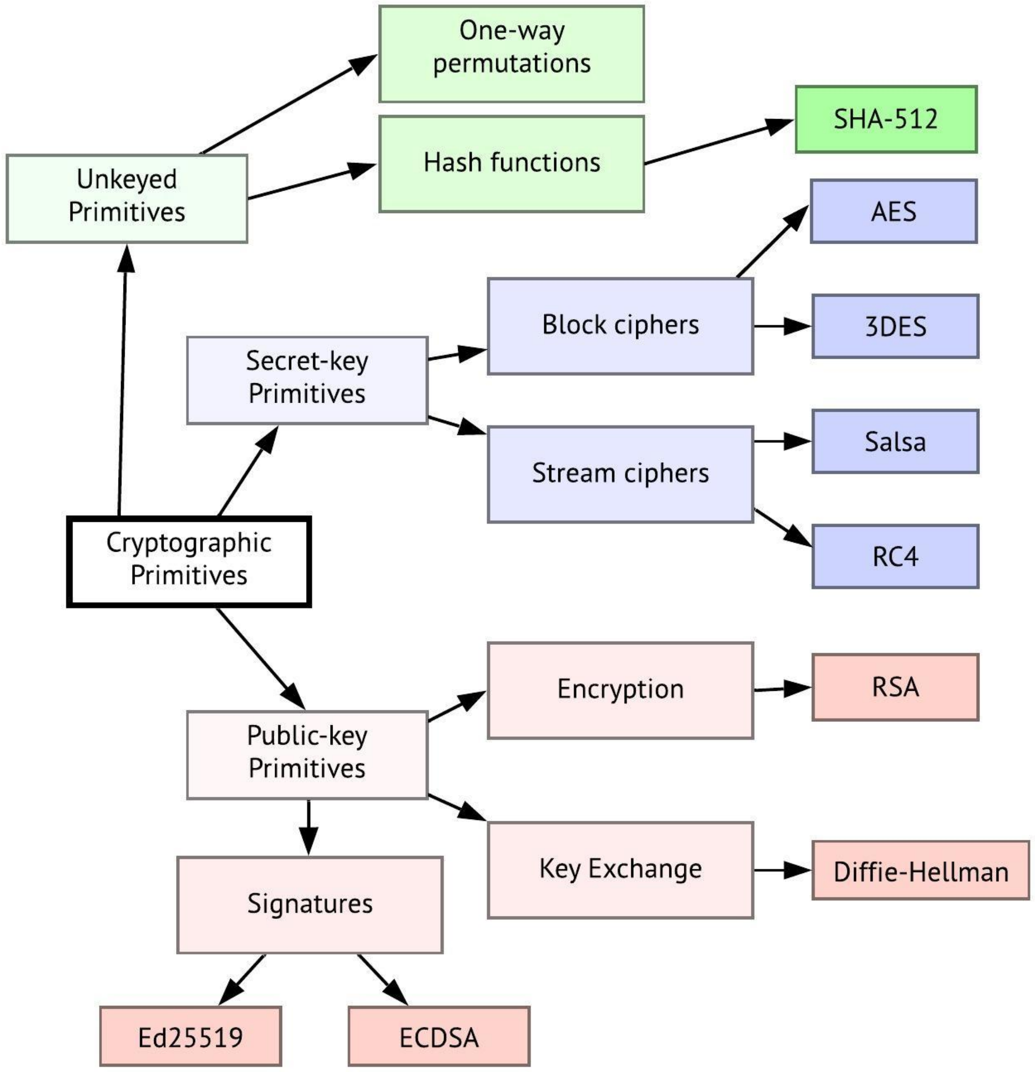

Figure 2.1: Classification of cryptographic primitives. Adapted from: Menezes et al. [3].

Some of the published papers that heavily impacted cryptography were the following:

“La Cryptographie Militaire” [16]. Set of requirements for cypher systems. Kerckhoffs states that the attacker knows the details of the encryption and decryption functions. The security of the communication should depend on the secret key instead of the cryptosystem. This way, cryptographic algorithms (e.g. RSA and AES) can be exhaustively studied, allowing them to evolve [17].

“Communication Theory of Secrecy Systems” [18]. Theoretical and mathematical approach of secrecy systems. Presents the ideas of confusion and diffusion, redundancy, entropy, and unicity distance [3].

There are several other works worth mentioning, such as: "New Directions in Cryptography" [15] (introduction of public key cryptography), "A Method for Obtaining Digital Signatures and Public Key Cryptosystems" [19] (first public-key cryptosystem, i.e. RSA), "Secure Communications Over Insecure Channels" [20] (introduction of public key distribution systems) and many others, which enables a deeper understanding of public key cryptography.

2.2.1 Symmetric Cryptography

The most common use for secret-key cryptography is establishing secure communication channels. For each communication channel we need a secret key K, which is shared between the entities. The key K is used for performing message encryption and decryption [21].

Example 2.2.1 describes an example of secret-key cryptography showed in Figure 2.2.

Example 2.2.1. ***Secret-key Encryption***

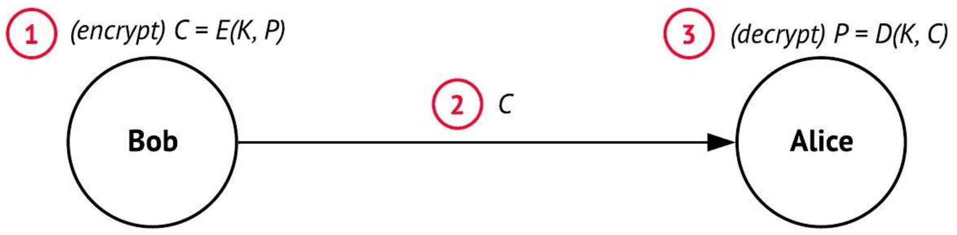

Figure 2.2: Example of secret-key cryptography.

1. *Bob transforms the plaintext P into the ciphertext C with the encryption function E: $C = E(P, K)$, s.t. P and K are inputs for the encryption function and K is the secret key.*

2. *Bob sends C to Alice and the message C becomes public information.*

③ *Alice retrieves the plaintext P from the ciphertext C using the decryption function D: $P = D(C, K)$, s.t. C and K are inputs for the decryption function and K is the secret key.*

Secret-key systems can be further classified as block ciphers and stream ciphers [22]:

Block cipher. A block cipher operates over iterations of permutations and substitutions. Essentially, each iteration needs a *round function*, defined by the state of the message and a *subkey*, as well as a *key schedule*, formed by N subkeys derived from the secret key. Examples of block ciphers: AES and 3DES.

Stream cipher. Instead of encryption algorithms being applied on blocks of data (block cipher), a stream cipher is applied on one bit at a time in a data stream. The plaintext and ciphertext have the same length. Examples of stream ciphers: RC4 and Salsa.

One of the main challenges for secret-key cryptosystems is key distribution: formulate an efficient method for distributing keys between participating entities [14].

2.2.2 Public-key Cryptography

Public-key cryptography was introduced by Whitfield Diffie and Martin E. Hellman [15]. The main idea is the introduction of mathematically linked key pairs - public key and secret key -, where a secure public channel can be established by only using public known information and techniques.

Table 2.1 presents families of public-key cryptosystems, such as key exchange, digital signature and encryption, exemplified in Examples 2.2.2, 2.2.3 [23] and 2.2.4 [3], respectively.

Table 2.1: Classification of public-key cryptosystems [24].

Key Exchange	Technique used for establishing keys between communicating entities.
Digital Signature	Technique used for data origin authentication and non-repudiation.
Encryption	Technique used for establishing secure communications (confidentiality).

Example 2.2.2. ***Public-key Encryption***

Example of public-key encryption, illustrated in Figure 2.3.

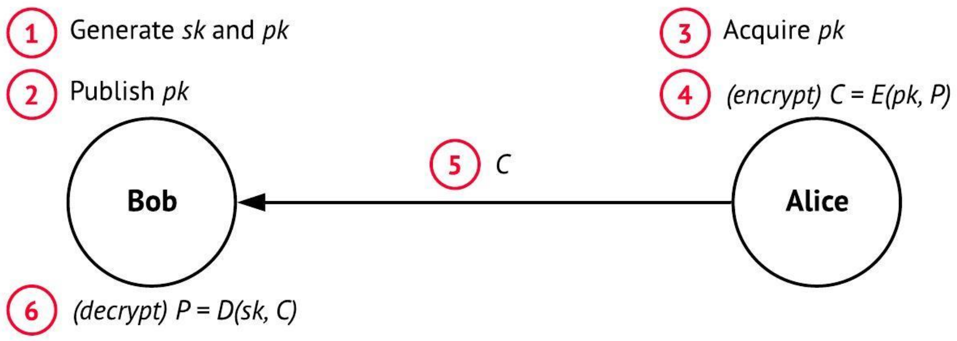

Figure 2.3: Public-key encryption.

1. *Bob generates the key pair: pk (public key) e sk (secret key).*
2. *Bob publishes the public key pk (public information).*
3. *Alice acquires the key pk.*
4. *Alice encrypts the message with the public key pk.*
5. *Alice sends the encrypted message to Bob.*
6. *Bob uses the secret key sk to decrypt the message.*

The communication is unidirectional. Alice also needs to generate a key pair in order to receive encrypted messages.

Example 2.2.3. ***Key Exchange***

An example of the Diffie-Hellman algorithm, illustrated in Figure 2.4.

1. *Bob chooses an integer $a \pmod{n}$.*
2. *Bob computes $A = g^a$.*
3. *Bob sends A to Alice.*

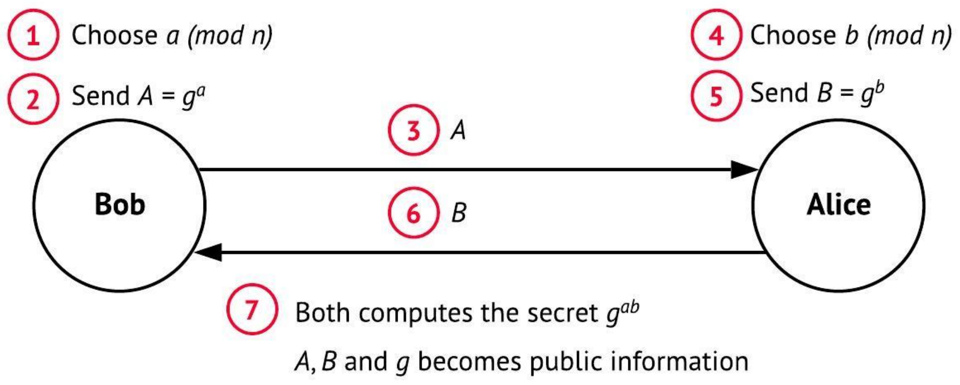

Figure 2.4: Key exchange.

④ *Alice chooses an integer $b \pmod n$.*

⑤ *Alice computes $B = g^b$.*

⑥ *Alice sends B to Bob.*

⑦ *The secret key g^{ab} can be computed as $g^{b(a)}$ (Bob) and $g^{a(b)}$ (Alice).*

The elements $A = g^a$, $B = g^b$ and g becomes public. Computing g^{ab} is infeasible and determines the Diffie-Hellman problem, which is equivalent to the discrete logarithm problem.

Example 2.2.4. ***Digital Signature***
Generic example of digital signatures, where Bob generates the signature and Alice performs the signature verification, illustrated in Figure 2.5.

1. *Bob generates the signature as follows:*

   ① *Generate the secret key sk and public key pk, and publish pk.*

   ② *Compute $h = H(m)$, where h is the result of the hash transformation H applied on the message m.*

   ③ *Compute $s = S(sk, h)$, where S is the signature generation function, with sk and h as inputs.*

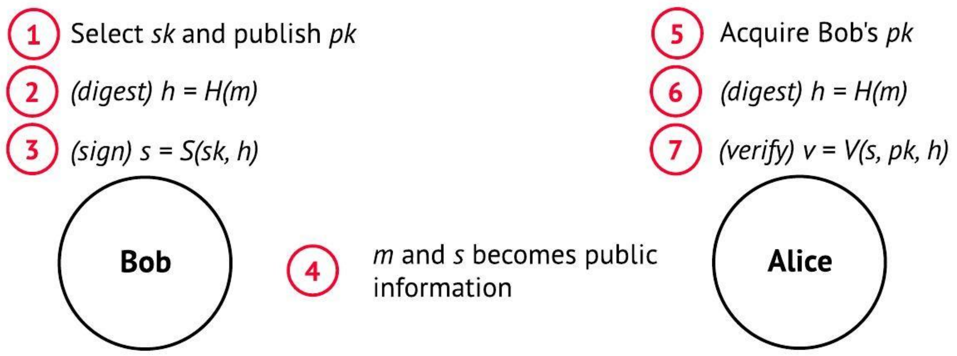
1 Select sk and publish pk
2 (digest) h = H(m)
3 (sign) s = S(sk, h)
Bob
4 m and s becomes public information
5 Acquire Bob's pk
6 (digest) h = H(m)
7 (verify) v = V(s, pk, h)
Alice

Figure 2.5: Digital signature.

> 4 *m and s becomes public information.*

2. *Alice verifies the message as follows:*

   5 *Acquire the public key pk and the message m (published by Bob).*

   6 *Compute $h = H(m)$.*

   7 *Compute $v = V(s, pk, h)$, where v is the result of the verification function, with s, pk and h as inputs. Accept the signature iff the verification function successfully verifies signature.*

*Hash functions takes a block of data as input and generates a fixed-size bit string **hash value**. If the input changes, the hash function generates a different hash value.*
Message integrity and origin authentication are security properties provided by digital signatures.

2.2.3 Secret-key & Public-key Cryptography

Secret-key cryptosystems presents the following pros and cons. Pros: (i) high throughput, (ii) smaller keys, as shown in Table 2.2, (iii) faster algorithms. Cons: (i) secret keys must be shared and kept secret within communicating entities, (ii) key management and (iii) key distribution mechanisms are necessary.

Contrasting secret-key cryptosystems, public-key cryptography presents the following pros and cons. Pros: (i) the secret key is not shared between communicating entities, (ii) digital signature, key exchange and encryption algorithms are possible, (iii) in public-key encryption systems, the number of keys necessary are less in comparison to secret-key encryption systems. Cons: (i) the security depends on the difficulty of a mathematical problem, (ii) slower in comparison to secret-key systems [3].

Table 2.2 shows, for each mathematical problem, the key sizes necessary to achieve a certain security level. In this context, we are able to draw the following remarks: (i) the difficulty of the integer factorization and discrete logarithm problems are equivalent, (ii) the contrast in key sizes between secret-key and public-key systems (to achieve a certain security level), which also varies depending on the underlying problem.

Table 2.2: Security level of cryptosystems. Source: Paar and Pelzl [24].

Algorithm	**Cryptosystem**	**Security Level (bit)**			
		80	**128**	**192**	**256**
Integer Factorization	RSA	1024	3072	7068	15360
Discrete Logarithm	DH, DSA, Elgamal	1024	3072	7680	15360
Elliptic Curves	ECDH, ECDSA	160	256	384	512
Secret-key	AES, 3DES	80	128	192	256

2.3 Elliptic Curves

The use of elliptic curves within cryptography was originally independently introduced by Neal Koblitz and Victor Miller. Essentially, an elliptic curve C is defined by a curve equation built on a field F for coordinates [25]. In order to understand elliptic curves, knowledge about group theory [26] and number theory [27] are required.

We start by giving an introduction to finite fields in Section 2.3.1. Then, we explain elliptic curves and the Elliptic Curve Discrete Logarithm Problem in Sections 2.3.2 and 2.3.3.

2.3.1 Finite Fields

Fields are abstractions of number systems. Examples of fields are real numbers ($\mathbb{R}$), complex numbers ($\mathbb{C}$) and rational numbers ($\mathbb{Q}$) [28]. Finite fields are defined by a finite number of elements (e.g. $F_{2^{255}-19}$, which is a finite field of size $2^{255}-19$). Definition 2.3.1 gives a more formal definition for *fields*.

Definition 2.3.1. *A field F has the following properties [28]:*

1. $(F, +)$ *is an abelian group with additive identity denoted by* 0.
2. $(F \backslash \{0\}, \cdot)$ *is an abelian group with multiplicative identity denoted by 1.*
3. *The distributive law holds:* $(a+b) \cdot c = a \cdot c + b \cdot c$ *for all* $a, b, c \in F$.

From Definition 2.3.1, the field F has two operations: addition and multiplication. The subtraction $a-b$, s.t. $a, b \in F$, can be written as $a-b = a+(-b)$, where $-b$ is the negative of b and is unique in F. Division in F can be written as: $a, b \in F, b \neq 0, \frac{a}{b} = a \cdot b^{-1}$, s.t. b^{-1} is the inverse of b and unique in F and $b \cdot b^{-1} = 1$.

Prime Fields

Lets consider F_p a prime field of order p. Additions and multiplications are calculated modulo p. Any integer a, where $a \in \{0, 1, 2, 3, ..., p-1\}$, $a \, mod \, p$ denotes the remainder r, s.t $0 \leq r \leq p-1$ (reduction modulo p).

Example 2.3.1. *Arithmetic operations in F_{29}, s.t.* $\{0, 1, 2, 3, ..., 26, 27, 28\}$ *represents the elements in F_{29}:*

1. *Addition:* $17 + 20 = 8 \Longleftrightarrow 37 \, mod \, 29 = 8$.
2. *Subtraction:* $17 - 20 = 26 \Longleftrightarrow -3 \, mod \, 29 = 26$.
3. *Multiplication:* $17 \cdot 20 = 21 \Longleftrightarrow 340 \, mod \, 29 = 21$.
4. *Inversion:* $17^{-1} = 12 \Longleftrightarrow 17 \cdot 12 \, mod \, 29 = 1$.

Binary Fields

The finite field F_{2^m} is called a binary field. F_{2^m} can be represented as binary polynomials of degree at most $m-1$:

$$F_{2^m} = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_2z^2 + a_1z^1 + a_0z^0 : a_i \in \{0,1\}\}.$$

Multiplication in F_{2^m} is calculated *modulo* $f(z)$, where $f(z)$ is a irreducible binary polynomial of degree m. Addition is calculated *modulo* 2. For any binary polynomial $a(z)$, $a(z)\,mod\,f(z)$ denotes the remainder $r(z)$ of degree less than m, resulted from the division $a(z)/f(z)$, which is called *reduction modulo* $f(z)$.

Example 2.3.2. *Arithmetic operations for* F_{2^4}*, such that the elements* $\{0, 1, z, z+1, z^2, z^2+1, z^2+z, z^2+z+1\} \in F$ *and* $f(\,z) \;= z^3+z+1$*, is given by:*

1. *Addition:* $(z^2+z+1)+(z+1) = z^2+2z+2 = z^2$.

2. *Subtraction:* $(z^2+z+1)-(z+1) = z^2$.

3. *Multiplication:* $(z^2+z+1)\cdot(z+1) = (z^3+1)\,mod\,(z^3+z+1) = z$.

4. *Inversion:* $(z^2+z+1)^{-1} = z^2 \iff (z^2+z+1)\cdot z^2\,mod\,(z^3+z+1) = 1$.

Modular arithmetic (showed in Examples 2.3.1 and 2.3.2) is fundamental for modern cryptography. It changes the way we compute efficient additions, subtractions, multiplications and inversions in a field F. Common groups for ECC are most primes and binaries [29].

2.3.2 Elliptic Curves

An elliptic curve E defined over a field F is given by the following equation:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

where a_i are elements in the field F. The set of points in $E(F_{p^k})$ satisfies the equation and the single point O at infinity. Coordinates satisfying such equation are tuples (e.g. affine coordinate system (x, y)) and its elements are in the finite field F. The scalar multiplication, addition and doubling laws allows arithmetic operations in the elliptic curve (group law). Elliptic curves have generators and a neutral point denoted ∞ [25].

Weierstrass Curve

Given a finite field F, the Weierstrass curve can be represented by the following equation:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

with discriminant different than zero (the curve is non-singular).

Montgomery Curve

A Montgomery curve can be represented by the equation:

$$M_{a,b} : By^2 = x^3 + Ax^2 + x,$$

where $A, B \in F$ and $(A^2 - 4) \neq 0$.

Twisted Edwards Curve

We expose more details for Twisted Edwards curves, since the Ed25519 scheme is an instantiation of the EdDSA, which is based on the specified curve.

The Twisted Edwards Curve was introduced in 2008 by Daniel Bernstein and Tanja Lange [30]. It is given by the equation:

$$E_{e,a,d} : ax^2 + y^2 = 1 + dx^2y^2,$$

s.t. the following conditions are satisfied:

- The field F is not binary ($char(F) \neq 2$).

- $a \neq 0$, $b \neq 0$ and $a \neq b$.

Arithmetic for twisted Edwards curves can be illustrated as follows [31]:

Point Addition. Consider the points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$. The addition law for Twisted Edwards is:

$$P \oplus Q = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \right) = (\, x_3, y_3)$$

The neutral element is $(0, 1)$ and the negative of (x, y) is $(-x, y)$. The addition formula is complete and also works for point doubling.

Point Doubling. Consider the point $P = Q = (x_1, y_1) = (x_2, y_2)$. The doubling law for the Twisted Edwards is $2P = P \oplus Q = (x_1, y_1) \oplus (x_2, y_2)$:

$$\begin{aligned} P \oplus Q &= \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \right) \\ 2P &= \left(\frac{x_1y_1 + y_1x_1}{1 + dx_1x_1y_1y_1}, \frac{y_1y_1 - ax_1x_1}{1 - dx_1x_1y_1y_1} \right) \\ = \left(\frac{2x_1y_1}{1 + dx_1^2y_1^2}, \frac{y_1^2 - ax_1^2}{1 - dx_1^2y^21} \right) &\iff \left(\frac{2x_1y_1}{1 + dx_1^2y_1^2}, \frac{y_1^2 - ax_1^2}{2 - ax_1^2 - y_1^2} \right) = (\, x_3, y_3) \end{aligned}$$

Repeated Doubling. Consider calculating $k \cdot P$, s.t. P is a point in the curve E and $k = 200$. We have [32]:

$$k \cdot P = 200 \cdot P = 2(2(2(P + 2(2(2(P + 2P))))))$$

2.3.3 Elliptic Curve Discrete Logarithm Problem

Consider the elliptic curve E defined over a field F and a point $P \in E(F)$, the Elliptic Curve Discrete Logarithm Problem (ECDLP) is represented as $Q = l \cdot P$, given Q as multiple of P and $l \in [0, n-1]$, s.t. $l = \log_P Q$. If the elliptic curve parameters are carefully chosen, the ECDLP is infeasible: the best general-purpose attack takes fully exponential time [28].

2.4 Edwards-curve Digital Signature Algorithm

Ed25519 is the most popular instance of the Edwards-curve Digital Signature Algorithm (EdDSA), and is based on the Curve25519 as twisted Edwards curve, providing $\approx$ 128-bits of security. The curve equation is [25]:

$$ax^2 + y^2 = 1 + dx^2y^2,$$

s.t. $a = -1$ and $d = -\frac{121665}{121666}$. The curve uses a prime field F_q, where $q = 2^{255} - 19$.

The RFC 8032 document [33] comprises the following advantages of Ed25519:

- High performance in various platforms.
- Generating a random number for each signature is not needed.
- Resilience against side-channel attacks.
- Complete formulas (e.g. addition law): the formula works for all points in the curve.
- Hash collision resistance.
- 128-bit security level.
- Small 32 byte keys and 64 byte signatures.

The EdDSA scheme has eleven carefully selected parameters (they vary between EdDSA signature systems, such as Ed25519 and Ed448). Table 2.3 specifies the main parameters for the Ed25519 scheme.

Next, we expose details regarding point encoding and decoding, key generation, signature generation and signature verification [33].

Point encoding. A curve point (x, y) is encoded as a 255-bit little-endian encoding of y. Then, the least significant bit of x is copied to the most significant bit of the last octet of the encoded y.

Point decoding. Related to the retrieval of the x coordinate:

Table 2.3: Ed25519 parameters. Source: RFC 8032 [33].

p	$2^{255} - 19$. Defines the underlying field F.
b	256.
Cofactor	8.
F_p encoding	$(b-1)$-bit little-endian encoding of $\{0, 1, ..., p-1\}$.
H	Hash function with 64 byte output (i.e. SHA-512).
d	$-\frac{121665}{121666} \in \mathbb{F}_p$.
a	-1.
B	Group generator $(x, \frac{4}{5}) \in E$ with positive x.
L	Group order $2^{252} + 27742317777372353535851937790883648493$.

1. Interpret the octet string (encoded point) as an integer in little-endian representation. x_0 is the least significant bit of x, which is the most significant bit of the interpreted string.

2. x is retrieved by computing the square root:

$$x^2 = (y^2 - 1)/(dy^2 + 1)^{-1} \pmod{n},$$

   which is calculated as shown in the next steps.

3. Consider $u = y^2 - 1$ and $v = dy^2 - a$. Compute the candidate root $w = (u/v)^{p+3/8} = uv^3(uv^7)^{(p-5)/8} \pmod{p}$.

4. Then, check three cases:

   - If $vw^2 = u \pmod{p}$, the square root is $x = w$.
   - If $vw^2 = -u \pmod{p}$, the square root is $x = w * 2^{(p-1)/4}$.
   - Else, the square root modulo p does not exist, and decoding fails.

5. If $x = 0$ and $x_0 = 1$, decoding fails.

6. If $x \pmod 2 = x_0$, then $x = x$. Else, $x = p - x$.

7. Finally, return the decoded point (x, y).

Key generation. The secret key Q and public key $\underline{A}$ are generated as follows:

1. Generate the b bits secret key Q using a cryptographically secure random number generator.

2. Hash the secret key: $H(Q) = (h_0, h_1, h_2, ..., h_{2b-1})$, with an output of size $2b$ bits (e.g. SHA-512).

3. We only use the lower 32 bytes of the digest $(h_0, h_1, h_2, ..., h_{b-1})$ to generate the public key. Set $h_0 = h_1 = h_2 = 0$, $h_{b-2} = 1$ and $h_{b-1} = 0$.

4. After pruning the buffer, interpret the result as a little-endian integer denoted s. Perform the fixed-base scalar multiplication $A = [s]B$.

5. The public key $\underline{A}$ is the encoding of the point $A = [s]B$.

Signature generation. In order to sign the message M we need both secret and public keys (Q and $\underline{A}$, respectively). The steps for generating the signature $(\underline{R}, \underline{S})$ are described below:

1. Hash the secret key: $H(Q) = (h_0, h_1, h_2, ..., h_{2b-1})$, with an output of size $2b$ (e.g. SHA-512).

2. We only consider the second half of the digest $(h_b, h_{b+1}, h_{b+2}, ..., h_{2b-1})$ for the next step.

3. Compute $r = H(h \parallel M)$, with an output of size $2b$ bits.

4. Perform r modulo L, then compute the point $R = [r]B$. $\underline{R}$ is the encoding of the point $R = [r]B$.

5. Compute the integer s as shown in the key generation algorithm.

6. Compute $k = H(\underline{R} \parallel A \parallel M)$, then reduce k modulo L.

7. Finally, $S = R + k * s$.

8. After encoding the point S, return the signature $(\underline{R} \parallel \underline{S})$.

Signature verification. The signature verification takes the public key $\underline{A}$ and the message m as inputs and tries to verify the signature $(\underline{R}, \underline{S})$:

1. Decode the first half of the signature as a point R. Then, decode the second half as an integer S. Reject the signature if the decoding fails.

2. Check that the integer S is in the range $0 \leq S \leq L$. Reject the signature if it fails.

3. Decode the public key $\underline{A}$ as a point A. Reject the signature if it fails.

4. Compute $k = H(R \parallel A \parallel M)$ as a 64 byte little-endian integer.

5. The signature is valid if the following is true: $[S]B \stackrel{?}{=} R + [k]A$.

Figure 2.6 presents an overview for Ed25519. The *Ed25519 Protocol* comprises operations (i.e. key generation, signing and verification) that are implemented over a group of points on an *Elliptic Curve*. Which, in its turn, is defined by the selected *Finite Field* (e.g. $F_{2^{255}-19}$). It is important to correctly implement the underlying mathematical structure and EdDSA protocol, so that undesired behaviors do not propagate into the *Application*. Specific optimizations techniques can be implemented depending on the features that the underlying *Platform* provides.

2.5 Conclusions

This chapter has showed the background theory for cryptography and elliptic curves. We have also introduced Rust, a fast and safe programming language. For last, we have explained the Ed25519 signature algorithm and provided an overview of a implementation project involving Ed25519.

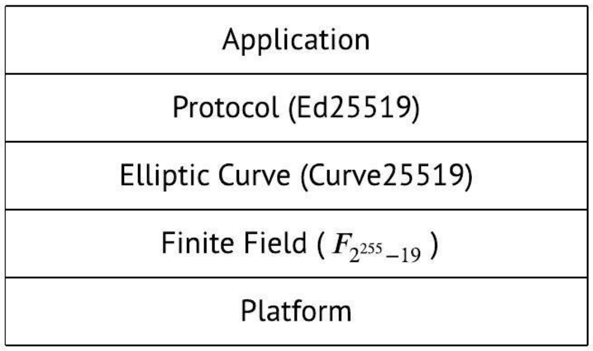
Application
Protocol (Ed25519)
Elliptic Curve (Curve25519)
Finite Field ($F_{2^{255}-19}$)
Platform

Figure 2.6: Ed25519 project stack.

Chapter 3

Related Work

There is a vast amount of work related to Ed25519 implementations and cryptographic performance and security analysis. We do not attempt to find and study all applicable works. Instead, we described some of the major Ed25519 implementations displayed in [34] (Section 3.1). Next, we focused on works that evaluated - performance wise - cryptographic implementations (Section 3.2). Lastly, we gathered works that verifies Ed25519's security (Section 3.3). Section 3.4 summarizes the main ideas presented in this chapter.

3.1 Ed25519 Implementation

This section presents existing implementations of the Ed25519 signature algorithm.

3.1.1 ed25519-dalek

The *ed25519-dalek* library [8] provides a Rust implementation of Ed25519 key generation, signing and verification. The implementation was built on top of the Curve25519 implementation *curve25519-dalek*. The *curve25519-dalek* library implements *u32*, *u64*, *avx2* and *ifma* backends. Safety features include constant-time logic, zeroisation of sensitive data, memory safety and no invalid points. Although security is prioritized, the implementation is very fast. The library offers serialization to/from bytes for secret and public keys and signatures, as well as batch signature verification.

3.1.2 ed25519-donna

ed25519-donna [7] offers fast and constant-time 32-bit and 64-bit implementations of Ed25519. It also leverages SSE2 instruction sets and offers batch signature verification. It is possible to disable compilation against OpenSSL and use a custom hash function instead.

3.1.3 ed25519-java

The *ed25519-java* [9] implementation is based on the *ref10* implementation in SUPERCOP. This implementation uses radix-$2^{25,26}$ and provides a fast and constant-time port of the radix-2^{51} operations in *ref10*, as well as a generic version using BigIntegers.

3.1.4 crypto/ed25519

The *crypto/ed25519* [35] package implements the Ed25519 algorithm in Go, compatible with RFC 8032. The code is a port of the *ref10* implementation in SUPERCOP and uses the radix-$2^{25,26}$ representation.

3.1.5 PyNaCl

PyNaCl [36] is a Python binding to *libsodium*, providing digital signatures, secret-key and public-key encryption, hashing, message authentication and key derivation. *libsodium* [37] is a *portable, cross-compilable, installable and packageable* fork of NaCl. The Libsodium library includes optimized Curve25519 implementations for 32-bit and 64-bit platforms, as well as a vectorized assembly implementation for SIMD architectures.

3.2 Performance Evaluation

ECC has been heavily considered for securing resource constrained devices due to its smaller signatures and key sizes and faster computation. TinyECC [38] is a configurable library for ECC operations in wireless sensors networks. Such configuration allows different combinations of optimizations that have different RAM and ROM consumption, execution time and energy consumption. The evaluation of the TinyECC library included resource consumption and performance measurements analysis on a range of sensor platforms. The metrics used were the following: ROM and RAM consumption, execution time and energy consumption. Regarding optimization techniques, there were six in total, namely the Barrett reduction, hybrid multiplication and hybrid squaring, projective coordinate systems, sliding window for scalar multiplications, Shamir's trick and curve specific optimization. The best configuration for storage and computational efficiency were presented.

HACL [4] is a modern C cryptographic library that implements the ChaCha20 and Salsa20 encryption algorithms, Poly1305 and HMAC message authentication, SHA-256 and SHA-512 hash functions, Curve25519, and the Ed25519 signature system. The code verification considered memory safety, timing side-channels and functional correctness with respect to RFC specifications. The authors also stated the library's compatibility with NaCl, TLS and Mozilla's NSS. Regarding the performance evaluation, the authors performed benchmarks on 32-bit and 64-bit platforms, using the CPU performance counter for measuring the average number of cycles needed to perform a typical operation. HACL was compared against OpenSSL, Libsodium and TweetNaCl. HACL's Ed25519 implementation proved to be much faster than TweetNaCl's, but slower compared to the others, since precomputed tables were not used. An important remark is the use of 64x64 bit multiplications resulting in 128-bit integers, which improves performance on supporting platforms.

3.3 Security Verification

Chalkias et al. [34] analyzes the security of Ed25519 implementations and standardization. The analysis is based on inconsistencies and incompatibilities in batch and single signature verification, binding and malleability guarantees and ambiguity in verification equations. The paper also emphasizes the convergence to a single interoperable Ed25519 scheme by considering security properties, such as repudiation resilience and strong unforgeability. It also approaches conflicting variants of Ed25519 implementations and correctness definitions, as well as recommendations, procedures and practical tools for choosing/building Ed25519 applications that aim for compatibility and offer a higher security notion. Similarly, Brendel et al. [39] helps with the understanding of Ed25519 variants and its properties. Therefore, the authors provide detailed analysis and security proofs for existential and strong unforgeability and resilience against key substitution attacks.

Bernstein et al. [40] focuses on the security analysis of the NaCl cryptographic library. More specifically, the security impact of NaCl's design features, such as *no data flow from secrets to load addresses, no data flow from secrets to branch conditions; no padding oracles; centralizing randomness; avoiding unnecessary randomness; extremely high speed; and cryptographic primitives chosen conservatively in light of the cryptanalytic literature*, were evaluated. Hence, these features are essential for the design of cryptographic libraries.

3.4 Conclusions

This chapter presented existing Ed25519 implementations and works that evaluated cryptographic libraries with regards to security and performance.

After doing a superficial analysis of Ed25519 implementations we have learned the following:

- Platform based optimizations, such as implementations tailored for 32-bit, 64-bit and SIMD architectures.
- Easy to use and misuse resistant interfaces.

- Readability and documentation leads to better code auditing and maintainability.
- Constant-time operations, memory safety, performance and resource consumption.

We were also able to understand security in two dimensions. (i) Ed25519 algorithm: procedures taken to offer security properties, such as strong unforgeability, non-repudiation and backwards-compatibility. (ii) Ed25519 implementation: prevent side-channel attacks by implementing constant-time operations; functional correctness. In terms of performance, ECC implementations can leverage optimization techniques with varying levels of resource consumption and execution time. Therefore, efficiency must be considered, especially for resource constrained devices, and should not conflict with security.

Chapter 4

Approach

This chapter describes the approach used to implement the Ed25519 signature system in Rust and the analysis process of Ed25519 security and performance. We organized this chapter in three sections. Section 4.1 describes the implementation process. Section 4.2 defines the methods for performing the analysis of the Ed25519 implementation and Rust. Finally, Section 4.3 summarizes this chapter.

4.1 Implementation

The process of transforming mathematical techniques into code is not an easy task. The ultimate goal of creating another Ed25519 implementation in Rust was to comprehend and document the process and the path taken to achieve our goals.

The Ed25519 system refers to the implementation of the following modules: field element arithmetic, group element arithmetic and an interface defining the cryptographic operations (key generation, signature generation and signature verification).

The implementation process consisted of a theoretical study of the signature system and the Rust programming language, followed by the implementation itself:

- **Step 1 (theory):** perform a study related to the Ed25519 signature system and the Rust programming language.

The study includes materials and documentations provided by existing implementations: namely the *ed25519-dalek* [8], *ed25519-java* [9] and *ed25519-donna* [7]. As well as articles and books, specially the original work [1] and the RFC 8032 [33]. The Rust community provides excellent documentations (i.e. The Rustonomicon [41]), fundamental to the success of this work.

- **Step 2 (implementation):** implement each layer in the following sequence, starting from the field element arithmetic, group element arithmetic and lastly, the key generation, signature and verification functions. The implementation should include aspects of *readability*, *modularity* and *ease of use.*

- **Step 3 (tests):** test the implementation in two ways.

  Unit testing. Ensure appropriate behavior for functions (i.e. underlying arithmetic for field elements and group elements).

  Regression testing. This can be achieved by testing the output of the key generation, signature generation and signature verification functions by consuming the test vectors available in [42].

- **Step 4 (library API):** build an appropriate interface for our Rust library.

  This step involves the implementation of an interface that provides the functionality of the Ed25519 signing algorithm. The interface should have the following characteristics:

  – Design an interface similar to existing Ed25519 implementations, such as the *ed25519-dalek* [8] library.

  – Provide clear documentation for data types and functions with examples regarding its usage.

  – Use Rust's type system to ensure that the signature, public key and secret key are distinguishable from each other. In other words, they are essentially arrays of bytes with another level of abstraction.

- **Step 5 (documentation):** The goal of the documentation is an overview of the Ed25519 algorithm and the exploration of its mathematical perspective reflected in the implementation. Thus, the documentation comprises the following:
  - Overview of the Ed25519 signing algorithm.
  - Describe the implementation, specially its structure, interface and general decisions considered in the development process, as well as explore core functions and its mathematical perspective when applicable.

4.2 Analysis

Lastly, we carried out a study of security and performance involving the Ed25519 algorithm and implementation.

4.2.1 Performance Evaluation

All benchmarks were performed in an Intel^{R} Core^{TM} i7-9750H Processor (64-bit) running Ubuntu 20.04 5.8.0-53-generic (64-bit)(with 16 GiB of RAM). Ultimately, our goal is to measure the performance of Ed25519 implementations with different configurations. We specially want to evaluate Rust's performance. In order to evaluate performance we considered the execution time and memory consumption:

Execution speed. The execution speed was measured using benchmarking libraries built for that specific language. We give special attention when comparing Rust and C implementations, considering that C is a fast programming language. Thus, we use Rust's Foreign Function Interface (FFI) to communicate with C cryptographic libraries. Consequently, two main factors arises. The first is the possibility to use the same benchmarking library to evaluate both C and Rust libraries. Second, this is a fair comparison if we consider that the overhead introduced is negligible [43]–[45]. Also, Table 4.1 shows similar execution times when comparing Rust's FFI with C's *clock()* function.

Table 4.1: Comparison between two benchmarking techniques: call C functions from Rust using FFI; native in C using the *clock()* function (using the *ed25519-donna* implementation).

ed25519-donna	**With FFI (μs)**	**Without FFI (μs)**
Key Generation	10.215	10.682
Signature Generation	11.279	11.317
Signature Verification	36.436	35.975

Memory consumption. We used Valgrind's Massif[1] tool to measure the memory consumption of programs that included all three Ed25519 operations (this was done for each implementation we selected).

Experiment Setup (*execution speed*)

The Ed25519 system involves three main operations: key and signature generation and signature verification. Therefore, these operations were the focus of our benchmarks.

To understand the impact of optimizations, we adopt two variables:

- Architecture: 32-bit or 64-bit implementations.
- Scalar multiplication: use of precomputed data to speed up scalar multiplications.

We also choose major Ed25519 implementations [34] in the following programming languages: C, Go, Rust, Java and Python. Ten implementations were selected in total, and are listed in Table 4.2. Table 4.2 also presents the main features provided by the implementation, as well as the benchmarking library.

Table 4.2: Ed25519 implementations used in the performance evaluation, as well as a description of the benchmarking libraries used, such as *criterion*, *timeit*, *testing* and *JMH*.

Benchmarking Environment

[1]https://www.valgrind.org/docs/manual/ms-manual.html

C and Rust/criterion library: collects 100 samples in estimated 5s of execution • 3s of warmup. Each sample contains a different number of iterations (*criterion* does not analyze individual iterations). ***Python/timeit library:*** used to collect the average execution time from at least 5s of execution, this was repeated three times and the lowest measurement was considered. ***Go/testing package:*** default configurations • 100.000 iterations. ***Java/JMH library:*** returns the average execution time from 300.000 iterations • 100.000 iterations warmup.
Library: ed25519-fun [46]. **Features:** 64-bit. **Environment:** rustc version *1.53.0-nightly* • criterion version *0.3*.
Library: ed25519-dalek version *1.0.1* [8]. **Features:** 32-bit • 64-bit • SIMD instructions. **Environment:** rustc version *1.53.0-nightly* • criterion version *0.3*.
Library: ed25519-java version *0.3.0* [9]. **Features:** 32-bit. **Environment:** JDK version *14.0.1* • JMH version *1.28*.
Library: Go package crypto/ed25519 [35]. **Features:** 32-bit. **Environment:** Go version *1.16* • Go package *testing*.
Library: ed25519 (oasisprotocol) [47]. **Features:** 32-bit • 64-bit. **Environment:** Go version *1.16* • Go package *testing*.
Library: Bouncy Castle version *1.68* [48]. **Features:** 32-bit. **Environment:** JDK version *14.0.1* • JMH version *1.28*.

Library: ed25519 (orlp) [49]. **Features:** 32-bit. **Environment:** GCC version *9.3.0* • Library *time.h* for *clock_t clock(void)* function • Rust crates (FFI): libc version *0.2* • criterion version *0.3*.
Library: ed25519-donna [7]. **Features:** 32-bit • 64-bit • SSE2 instructions. **Environment:** GCC version *9.3.0* • Library *time.h* for *clock_t clock(void)* function • Rust crates (FFI): libc version *0.2* • criterion version *0.3*.
Library: PyNaCl version 1.4.0 [36]. **Features:** 32-bit • 64-bit • SIMD instructions. **Environment:** Python version *3.8.5* • Python library *timeit*.
Library: pyca version *4.1* [50]. **Features:** 64-bit. **Environment:** Python version *3.8.5* • Python library *timeit*.

To simplify, we create three scenarios. In the first two scenarios we perform benchmarks that reflects the impact of the optimizations. The third scenario presents benchmarks for all selected implementations. These benchmarks were applied for all of the Ed25519 operations (namely key generation, signing and verification).

Experiment Setup (*memory consumption*)

We used Valgrind's Massif (version 3.15.0) to measure memory consumption of programs, such that each program used a specific Ed25519 library and included key generation, signing and verification functions pertaining to that library. The --*pages*-*as*-*heap* option replaces heap profiling for lower-level page profiling: the stack, heap, data and BSS segments are all measured; the --*stacks* option, in the other hand, measures the stack and the heap. Both options were used in the performance analysis.

4.2.2 Security

To gain more insights into the features of the Ed25519 signature system/implementation and what makes Rust a suitable language for cryptographic implementations, we also performed a study focused on security properties. Straightforwardly, we discussed security in three perspectives:

1. Implementation: discussion of security considerations in the implementation level, such as constant-time operations.
2. Programming language: discussion of Rust's safety features.
3. Algorithm: classification of our implementation in respect to the security properties exposed by Brendel et al. [39] and Chalkias et al. [34].

4.3 Conclusions

As stated in this chapter, this work focuses on the Ed25519 implementation in Rust and its analysis. The implementation is composed by the comprehension of the theory, Ed25519 implementation in Rust and the documentation. The analysis targets the implementation's performance and security.

Chapter 5

Ed25519 Implementation

This chapter focuses on the implementation details (the implementation can be found in GitHub[1]) and is organized as follows. Section 5.1 describes the implementation itself (field arithmetic, curve arithmetic and protocol). Section 5.2 expands on implementation properties, such as modularity, readability and ease of use. Unit testing and regression testing are presented in Section 5.3. The interface was exemplified in Section 5.4. Lastly, Section 5.5 summarizes this chapter.

5.1 Implementation

The implementation can be divided as: field element, group element and interface. They are presented in Sections 5.1.1, 5.1.2 and 5.1.3, respectively.

5.1.1 Field Element Arithmetic

This section describes the field element representation and arithmetic, i.e. the radix-2^{51} representation, field element addition, subtraction, multiplication, squaring and inversion.

[1]https://github.com/yuzonightly/ed25519-fun.git

Radix-2^{51} Representation

Considering the goal to reduce the number of instructions when performing field element arithmetic and the inability to perform 255-bit integer arithmetic on mainstream computers, we take an element x of field $\mathbb{F}_{2^{255}-19}$ as five 64-bit limbs $(x_0, x_1, x_2, x_3, x_4)$: $x = \sum_{i=0}^{4} x_i 2^{51i}$ [1].

x is represented as:

$$x = x_0 + x_1 2^{51} + x_2 2^{102} + x_3 2^{153} + x_4 2^{204}, \text{ with } 0 \leq x_i \leq 2^{51}.$$

Addition and Subtraction

Field Element additions can be done simply by performing limb by limb addition. Dealing with carries is not necessary. The reason lies on the absence of large addition chains.

One addition and one subtraction for each limb is needed to perform subtraction on field elements. Since we are storing them as 64-bit unsigned, in order to avoid underflow we first add a multiple of p (e.g. $k \cdot p$), and then perform the subtraction: $s = (x + k \cdot p) - y$. Listing 5.1 shows the code that performs the subtraction.

Listing 5.1: Subtraction on field elements.

```rust
let mut h = [0u64; 5];

// Perform h = self - g
// The element [twoP0, TwoP1234, TwoP1234, TwoP1234, TwoP1234] is a multiple
// of p: 2 * (2^255 - 19).
h[0] = (self.0[0] + TwoP0) - g.0[0];
h[1] = (self.0[1] + TwoP1234) - g.0[1];
h[2] = (self.0[2] + TwoP1234) - g.0[2];
h[3] = (self.0[3] + TwoP1234) - g.0[3];
h[4] = (self.0[4] + TwoP1234) - g.0[4];

// Perform field element reduction
FieldElement::reduce(h)
```

Multiplication

The result of the multiplication of two field elements x and y, $z = x{\cdot}y = z_0 + z_1 2^{51} + z_2 2^{102} + \cdots + z_8 2^{408}$, can be obtained as follows. By exposing that $p = 2^{255} - 19 \longrightarrow 2^{255} \equiv (\,19 \mod p)$, Equation 5.1 is straightforward [1].

$$z = (\,z_0 + 19z_5)\; + (\,z_1 + 19z_6)\,2^{51} + (\,z_2 + 19z_7)\,2^{102} + (\,z_3 + 19z_8)\,2^{153} + z_4 2^{204} \tag{5.1}$$

$$\begin{aligned}
z_0 &= x_0y_0\\
z_1 &= x_0y_1 + x_1y_0\\
z_2 &= x_0y_2 + x_1y_1 + x_2y_0\\
z_3 &= x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0\\
z_4 &= x_0y_4 + x_1y_3 + x_2y_2 + x_3y_1 + x_4y_0\\
z_5 &= x_1y_4 + x_2y_3 + x_3y_2 + x_4y_1\\
z_6 &= x_2y_4 + x_3y_3 + x_4y_2\\
z_7 &= x_3y_4 + x_4y_3\\
z_8 &= x_4y_4
\end{aligned} \tag{5.2}$$

From Equations 5.1 and 5.2 we can deduce Equation 5.3.

$$\begin{aligned}
z_0 &= x_0y_0 + 19(\,x_0y_4 + x_2y_3 + x_3y_2 + x_4y_1)\\
z_1 &= x_0y_1 + x_1y_0 + 19(\,x_2y_4 + x_3y_3 + x_4y_2)\\
z_2 &= x_0y_2 + x_1y_1 + x_2y_0 + 19(\,x_3y_4 + x_4y_3)\\
z_3 &= x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 + 19(\,x_4y_4)\\
z_4 &= x_0y_4 + x_1y_3 + x_2y_2 + x_3y_1 + x_4y_0
\end{aligned} \tag{5.3}$$

In order to obtain coefficients that are slightly larger than 2^{51}, reduce (dealing with carries) is necessary after performing multiplications [51]. The same approach is applied to field element squaring (Section 5.1.1). Listing 5.2 shows the Rust code for field multiplication, which reflects Equation 5.3. It is important to note that inner field element ($u64$ limbs) multiplications are performed as $(f\ as\ u128) \cdot (g\ as\ u128)$, where both f and g are $u64$.

Listing 5.2: Field multiplication modulo p

```rust
let h0 = f0g0 + f1g4_19 + f2g3_19 + f3g2_19 + f4g1_19;
let mut h1 = f0g1 + f1g0 + f2g4_19 + f3g3_19 + f4g2_19;
let mut h2 = f0g2 + f1g1 + f2g0 + f3g4_19 + f4g3_19;
let mut h3 = f0g3 + f1g2 + f2g1 + f3g0 + f4g4_19;
let mut h4 = f0g4 + f1g3 + f2g2 + f3g1 + f4g0;
```

Squaring

Field element squaring is similar to its multiplication. We have $z = x \cdot x = z_0 + z_1 2^{51} + z_2 2^{102} + \cdots + z_8 2^{408}$ and $p = 2^{255} - 19 \longrightarrow 2^{255} \equiv (19 \mod p)$. Equation 5.4 describes the operation $x \cdot x$.

$$\begin{aligned}
z_0 &= 2 \cdot x_0 \\
z_1 &= x_0x_1 + x_1x_0 \\
z_2 &= x_0x_2 + 2 \cdot x_1 + x_2x_0 \\
z_3 &= x_0x_3 + x_1x_2 + x_2x_1 + x_3x_0 \\
z_4 &= x_0x_4 + x_1x_3 + 2 \cdot x_2 + x_3x_1 + x_4x_0 \\
z_5 &= x_1x_4 + x_2x_3 + x_3x_2 + x_4x_1 \\
z_6 &= x_2x_4 + 2 \cdot x_3 + x_4x_2 \\
z_7 &= x_3x_4 + x_4x_3 \\
z_8 &= 2 \cdot x_4
\end{aligned} \tag{5.4}$$

Equations 5.1 and 5.4 generates the limbs $z_0, \ldots, z_4$ illustrated in Equation 5.5.

$$\begin{aligned}
z_0 &= 2 \cdot x_0 + 19(2 \cdot x_1x_4 + 2 \cdot x_2x_3) \\
z_1 &= 2 \cdot x_0x_1 + 19(2 \cdot x_2x_4 + 2 \cdot x_3) \\
z_2 &= 2 \cdot x_0x_2 + 2 \cdot x_1 + 19(2 \cdot x_3x_4) \\
z_3 &= 2 \cdot x_0x_3 + 2 \cdot x_1x_2 + 19(2 \cdot x_4) \\
z_4 &= 2 \cdot x_0x_4 + 2 \cdot x_1x_3 + 2 \cdot x_2
\end{aligned} \tag{5.5}$$

Reduce is also necessary to obtain coefficients that are slightly larger than 2^{51}, similarly to field element multiplications.

Inversion

For field element inversion we have $a^p \cong a \mod p$ and therefore $a^{(p-2)} \cong a^{-1} \mod p$. More specifically we will simply compute $a^{2^{255}-21} \mod p$. This can be achieved by performing sequences of squaring and multiplications. For example:

- Square field element a: $b = a \cdot a = a^2$.
- Square field element b: $c = b \cdot b = a^4$.
- Multiply a and b: $d = a^2 \cdot a^4 = a^6$.
- Square field element d: $e = d \cdot d = (a^6)^2 = a^{2^4-2^2}$.

Listing 5.3 shows (partially) the code that performs the field element inversion.

Listing 5.3: Field element inversion.

```
...
...
// 1 * 2 = 2 -> self^2
let mut t0 = self.square();
// 2 * 2 = 4 -> self^4
let mut t1 = t0.square();
```

```
// 4 * 2 = 8 -> self^8
t1 = t1.square();
// 1 + 8 = 9 -> self^9
t1 = self.mul(t1);
// 2 + 9 = 11 -> self^11
t0 = t0.mul(t1);
// 2 * 11 = 22 -> self^22
let mut t2 = t0.square();
// 22 + 9 = 31 -> self^31
t1 = t1.mul(t2);
// 31 * 2 = 62 = 2^6 - 2^1 -> self^(2^6 - 2)
t2 = t1.square();
// 2^4 * (2^6 - 2^1) = 2^10 - 2^5 -> self^(2^10 - 32)
t2 = t2.square_times(4);
...
...
// Perform squarings and multiplications until self^(2^255 - 21)
```

5.1.2 Group Element Arithmetic

In this section we aim for expanding details about group arithmetic, i.e. coordinate systems, point addition, point doubling and scalar multiplication.

Coordinate Systems

Alternate coordinate systems are considered for performing point arithmetic.

The point (x, y) in the *extended* coordinate system (P^3) is represented as:

$$\begin{aligned} x &= X/Z \\ y &= Y/Z \\ xy &= T/Z, \end{aligned}$$

The identity element is $(0 : 1 : 1 : 0)$ and the negative element is $(-X : Y : Z : -T)$.

The point (x, y) in the *projective* coordinate system (P^2) is represented as:

$$x = X/Z$$
$$y = Y/Z,$$

where $Z \neq 0$.

The *precomputed* representation is defined as $(y + x, y - x, 2 * d * x * y)$, where d is the constant (Ed25519 parameter) $-121665/121666 \pmod{p}$. The precomputed values (*precomp.rs*) used in scalar multiplications are points in the precomputed representation.

The *cached* representation also uses precomputations, satisfying $(Y + X, Y - X, Z, 2 * d * T)$ (X, Y, Z and T are values extracted from the P^3 representation).

Finally, the *completed* ($P \times P$) representation $((X : Z), (Y : T))$ satisfies $x = X/Z$ and $y = Y/T$.

Addition and Subtraction

Addition and subtraction were implemented for points X and Y as $X \oplus Y$, where X is in the extended point representation and Y can either be in the precomputed or cached representations. The result of the operation is returned as $P \times P$ (completed representation). The code is available in the Appendix B.

Doubling

Doubling the point X requires performing $X + X$. The point X can either be in the extended or projective representation, and the result is returned as $P \times P$ (completed representation). The code is available in the Appendix B.

Fixed-window Scalar Multiplication

Two approaches were considered for scalar multiplications: one approach uses fixed windows and the other one uses sliding windows (Section 5.1.2). We will go through the constant-time fixed window approach in this section.

We consider a fixed window of size 4. The following needs to be computed:

$$S = a * P,$$

s.t. a is defined as $a_0 + a_1 \cdot 256^1 + a_2 \cdot 256^2 \cdots a_{31} \cdot 256^{31}$ and P is the base point.

For point P we precompute the following table:

	$\boldsymbol{P}$	$\boldsymbol{2 \cdot P}$	$\boldsymbol{3 \cdot P}$	$\boldsymbol{\cdots}$	$\boldsymbol{8 \cdot P}$
$\mathbf{16^2}$	$256 \cdot P$	$2 \cdot 256 \cdot P$	$3 \cdot 256 \cdot P$	$\cdots$	$8 \cdot 256 \cdot P$
$\mathbf{16^4}$	$256^2 \cdot P$	$2 \cdot 256^2 \cdot P$	$3 \cdot 256^2 \cdot P$	$\cdots$	$8 \cdot 256^2 \cdot P$
$\mathbf{16^6}$	$256^3 \cdot P$	$2 \cdot 256^3 \cdot P$	$3 \cdot 256^3 \cdot P$	$\cdots$	$8 \cdot 256^3 \cdot P$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	
$\mathbf{16^{62}}$	$256^{31} \cdot P$	$2 \cdot 256^{31} \cdot P$	$3 \cdot 256^{31} \cdot P$	$\cdots$	$8 \cdot 256^{31} \cdot P$

Then, compute the radix-2^4 representation for the scalar a:

$$a = a_0 + a_1 \cdot 16^1 + a_2 \cdot 16^2 + \cdots + a_{63} \cdot 16^{63},$$

where $-8 \leq a_i < 8$ and $0 \leq |a_i| \leq 8$. Therefore, instead of doing precomputations for $P \ldots 16P$, only $P \ldots 8P$ precomputations were generated. This is possible by using signed values for a_i and transforming the resulting point $(a_i \cdot 16^j \cdot P)$ into its negative whenever a_i is negative.

The scalar multiplication can be solved by iterating through a and accumulating each $a_i \cdot 16^j \cdot P$. It is important to note that we accumulate odd indexes first and multiply the result by 16. Although performing the scalar multiplication with the lookup table uses more memory, it is much faster. The code for the fixed-window scalar multiplication can be found in Appendix C.

Sliding-window Scalar Multiplication

The goal is to compute the following:

$$S = a \cdot P,$$

where $a = a_0 + a_1 \cdot 256 + a_2 \cdot 256^2 + \cdots + a_{31} \cdot 256^{31}$ and P is a point in the curve.

In contrast to the fixed-window scalar multiplication, the sliding-window scalar multiplication is used for verifying signatures. It uses less memory but is not constant-time.

The idea is to "slide" the window over the scalar [52]. The following steps were implemented:

1. First, a is passed to the *slide* function. The function *slide* takes the parameter a, which has 32 8-bit unsigned elements $(a_1, a_2, ..., a_{31})$, and returns r.
2. r has 256 8-bit signed elements and is defined as: $r_0 + r_1 \cdot 2^1 + r_2 \cdot 2^2 + \cdots + r_{255} \cdot 2^{255}$. Each element $r_i \in \{-15, -13, -11, -9, -7, -5, -3, -1, 0, 1, 3, 5, 7, 9, 11, 13, 15\}$.
3. Then, since r_i is always odd, we precompute values for the point P: $\{P$, $3P$, $5P$, $7P$, $9P$, $11P$, $13P$, $15P\}$.
4. Lastly, perform the chain of point doublings and additions in r, and accumulate the result after each operation.

Although the sliding-window method is more efficient, it is not constant-time and is only used for verifying signatures. The code can be found in Appendix D.

5.1.3 Interface

Fundamentally, the interface should have at least three operations: key generation, signature generation and signature verification. These operations are illustrated in Figures 5.1 and 5.2. Both diagrams reflect the steps described in Section 2.4. It is important to observe a few things about the design illustrated in both figures: Ed25519 performs *secret key clamping* (*set/clear bits*); signing also depends on the public key; verification also depends on checks and point decoding validations [39].

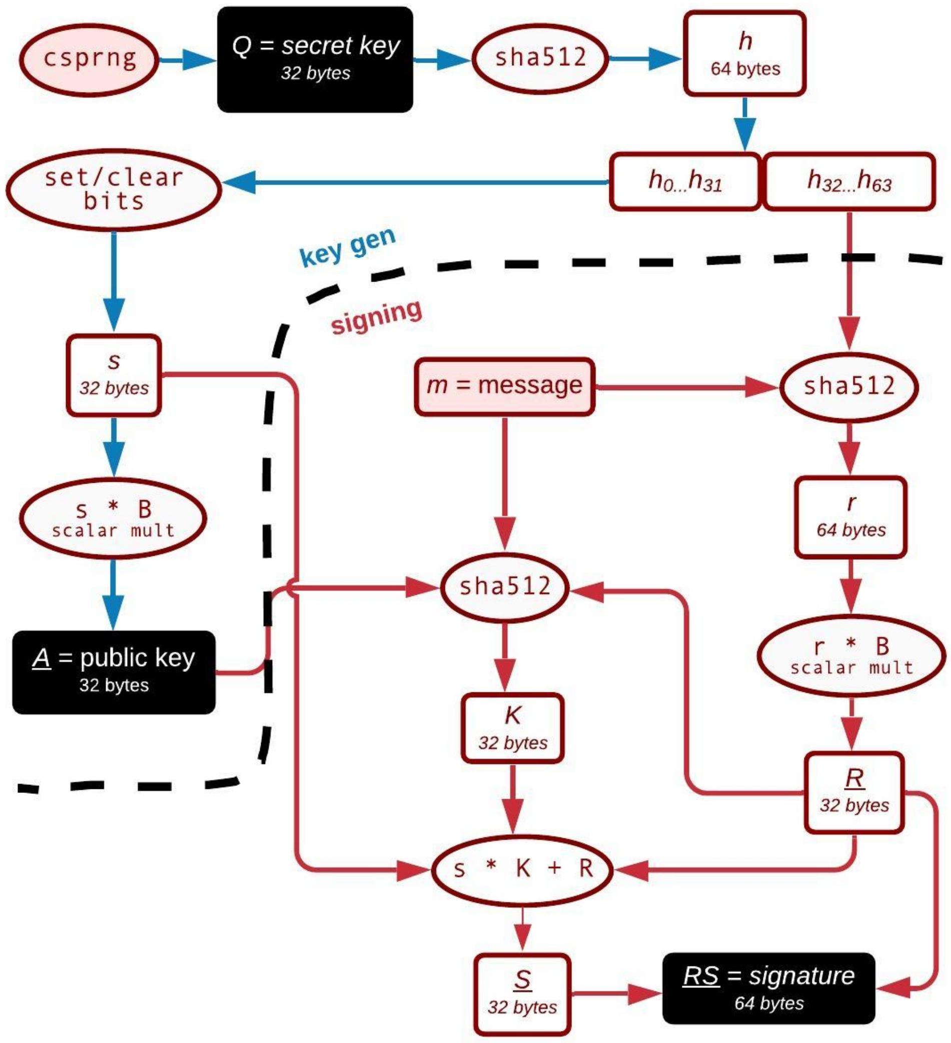

Figure 5.1: Overview of Ed25519's key generation and signature generation.

Further details about the usage of the library API (code examples) and a description of the Ed25519 algorithm (showing the key generation, signing and verification processes) were given in Sections 5.4 and 2.4, respectively.

5.2 Implementation Properties

The properties considered for the implementation were three: modularity, readability and ease of use.

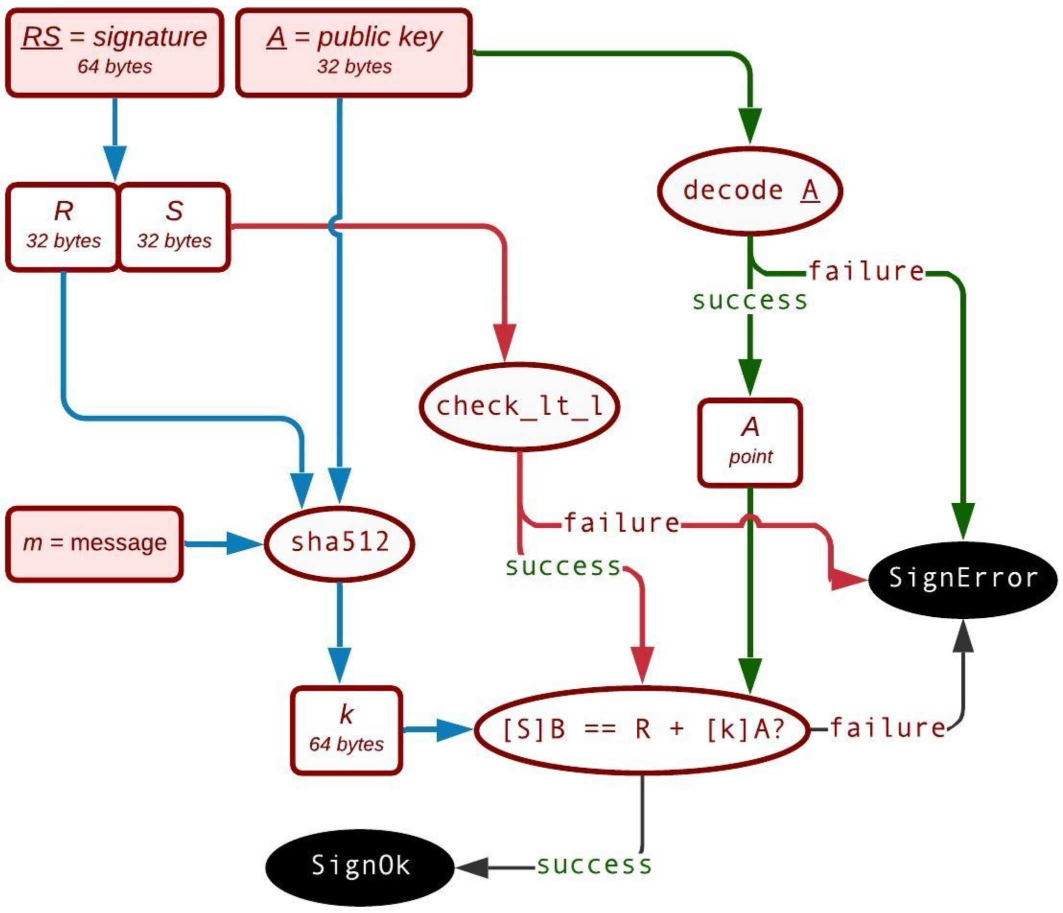

Figure 5.2: Overview of Ed25519's signature verification.

5.2.1 Modularity

The implementation was organized as follows:

```
/ed25519-fun
 ├─ /benches
 ├─ /src
 │   ├─ /curve25519
 │   │   ├─ constants.rs
 │   │   ├─ field_element.rs
 │   │   ├─ group_element.rs
 │   │   ├─ mod.rs
 │   │   ├─ precomp.rs
 │   │   ├─ scalar_ops.rs
 │   │   └─ utils.rs
 │   ├─ constants.rs
 │   ├─ errors.rs
 │   ├─ keypair.rs
 │   ├─ lib.rs
 │   ├─ public.rs
 │   ├─ secret.rs
 │   └─ signature.rs
 ├─ /tests
 │   ├─ ed25519_testing.rs
 │   ├─ sign.input
 │   └─ verify.input
 └─ Cargo.toml
```

The implementation was organized similarly to the *ed25519-java* implementation. The library was divided in three parts. (i) Built-in benchmarks for key generation, signing and verification were included in the */benches* folder. (ii) */src* comprises the Ed255519 implementation. (iii) */tests* includes the regression testing. The */src* folder was organized as follows:

Curve25519. The module */curve25519* houses the implementation of the field element arithmetic, group element arithmetic and scalar operations.

Ed25519. The modules *keypair.rs*, *public.rs*, *secret.rs* and *signature.rs* implements the specification for the secret key, public key and signature, as well as the implementation of the Ed25519 protocol. To do so, it uses the */curve25519* module.

5.2.2 Readability

In this case, we refer to readability simply as the appropriate attribution of names. To exemplify, the code in Listing 5.4, which returns a point in P^3 representation given a 32 byte encoding *enc*, illustrates two main characteristics. First, names given to functions are easily associated with the arithmetic applied to the field element. Second, names given to variables follows the same pattern introduced by the RFC 8032 and articles.

Listing 5.4: Point decoding.

```rust
pub fn decode(enc: [u8; 32]) -> Option<P3> {
    let y = FieldElement::decode(enc);
    let yy = y.square();
    let u = yy - FieldOne;
    let v = (yy * D) + FieldOne;
    let v3 = v.square() * v;
    let v7 = v3.square() * v;
    let mut x = u * v7;
    x = x.pow22523();
    x = x * u * v3;
    let vxx = x.square() * v;
    let mut check = vxx - u;
    if check.is_zero().unwrap_u8() == 0u8 {
        check = vxx + u;
        if check.is_zero().unwrap_u8() == 0u8 {
            return None;
        }
        x = x * I;
    }

    if x.is_negative().unwrap_u8() == enc[31] >> 7 {
        x = x.negate();
    }

    Some(P3 {
        X: x,
        Y: y,
        Z: FieldOne,
        T: x * y,
    })
}
```

5.2.3 Ease of Use

We associate ease of use with the library interface and code examples.

Code examples. Proper usage of the library were exposed in this documentation and on the root of the GitHub repository. Also, complete examples were written as Rust comments (for data types and functions).

Library API. As for the library interface, we provided distinct Rust types for each element: *SecretKey* for secret keys, *PublicKey* for public keys, *Signature* for signatures, and *Keypair* for both secret and public keys. The interface has similarities with the *ed25519-dalek* implementation. In this context, the *Keypair* type implements both signing and verification functions, *SecretKey* implements the signing function, and *PublicKey* implements the verification function. Therefore, using distinct types to represent different elements (even though these elements are all byte arrays), and functions implemented under the correct types, are essential for the usability of the library and to increase the difficulty of misusing the API.

5.3 Tests

Tests were applied for internal functions and Ed25519 operations. We classified them as:

Unit testing. Written mainly for field elements (*field_element.rs*), group elements (*group_element.rs*), scalar operations (*scalar_ops.rs*) and utilitarian functions (*utils.rs*). Tests were done using random inputs. The goal was to ensure basic functionality by generating the correct output.

Regression testing. Regression tests were written for key generation, signing and verification functions. Tests were taken from [42]. Essentially, the test vectors have 1024 instances, each containing a message, secret and public keys, and the corresponding signature. Therefore, the key generation, signature generation and signature verification functions were tested. The output from each operation was matched against the test vectors.

5.4 Library API

Key generation. The secret key and public key are generated using the *generation* function from *Keypair*, as shown in Listing 5.5.

Listing 5.5: Example of secret key and public key generation.

```
let keypair: Keypair = Keypair::generate();
```

Signature generation. Signing is done through the *sign* function from *Keypair*, as shown in Listing 5.6.

Listing 5.6: Example of signature generation.

```
let message: &[u8] = b"";
let signature: Signature = keypair.sign(message);
```

Signature verification. In order to verify signatures we use the *verify* function from *Keypair*, as shown in Listing 5.7.

Listing 5.7: Example of signature verification.

```
let signok: bool = keypair.verify(message, signature);
```

Listing 5.8 provides a full example using our library.

Listing 5.8: Example using the library API.

```
extern crate ed25519_fun;
use ed25519_fun::{Keypair, Signature, SecretKey, PublicKey};

```

```rust
fn main() {
    // Keypair generation
    let keypair: Keypair = Keypair::generate();

    // Signing
    let message: &[u8] = b"";
    let signature: Signature = keypair.sign(message);

    // Verification
    let signok: bool = keypair.verify(message, signature);

    // The Keypair can also be generated with the secret key
    let secret_key: SecretKey = keypair.secret;
    let keypair_from_secret_key = Keypair::generate_public_key(secret_key);

    // Signing can also be performed with SecretKey
    let public_key: PublicKey = Keypair.public;
    let secret_key_signing: Signature = secret_key.sign(public_key, message);

    // Verification can also be performed with PublicKey
    let public_key_signok = public_key.verify(message, signature);

    // Transform Keypair, SecretKey, PublicKey and Signature into byte arrays
    let keypair_bytes: [u8; 64] = keypair.as_bytes();
    let secret_bytes: [u8; 32] = secret_key.as_bytes();
    let public_bytes: [u8; 32] = public_key.as_bytes();
    let signature_bytes: [u8; 64] = signature.as_bytes();

    // Transform byte arrays into its corresponding types
    let keypair_from_bytes: Keypair = Keypair::from_bytes(&keypair_bytes);
    let secret_key_from_bytes: SecretKey = SecretKey::from_bytes(&secret_bytes);
    let public_key_from_bytes: PublicKey = PublicKey::from_bytes(&public_bytes);
    let signature_from_bytes: Signature = Signature::from_bytes(&signature_bytes);
}
```

5.5 Conclusions

This chapter described the implementation of the signature scheme. For simplicity, we have only explored the main ideas and functions surrounding the underlying arithmetic. Then, we attempted to explain the properties we chose to consider in the implementation: modularity, readability and ease of use. They are simple concepts that resulted in writing code that is easier to read and maintain, and in a usable interface. Lastly, we described the tests written for the internal functions and API, as well as code examples using the library.

Chapter 6

Performance and Security Analysis

This chapter focuses on the aspects of performance and security in two perspectives: Rust programming language and Ed25519 implementation and algorithm. Section 6.1 evaluates Ed25519 implementations by measuring the execution time of the key generation, signing and verification functions, as well as the memory footprint. Next, Section 6.2 explores Rust's and Ed25519's security.

6.1 Evaluation

We observed the impact of precomputed values in scalar multiplications, as well as 32-bit and 64-bit implementations in Sections 6.1.1 and 6.1.2, respectively. Then, we analyzed our benchmark results for various implementations in Section 6.1.3. Lastly, Section 6.1.4 shows memory consumption. The code can be found in GitHub[1].

6.1.1 Scalar Multiplication

First, we take look at the impact of precomputed values, explained at Section 5.1.2. As shown in Table 6.1, the use of precomputed values in scalar multiplications can drastically improve performance in key and signature generation, seeing that scalar multiplications

[1]https://github.com/yuzonightly/ed25519-bench

are the most expensive operation in ECC arithmetic. Our results indicated an increase in performance by 81.77% and 78.87% in key and signature generation, respectively.

Table 6.1: Performance impact of 30 KB of precomputed values (with our implementation).

ed25519-fun	**With Precomputation (μs)**	**Without Precomputation (μs)**
Key Generation	21.065	115.99
Signature Generation	25.294	120.05

By using precomputed values, we perform 128 table lookups and 128 point additions. Otherwise, 512 point additions are necessary.

6.1.2 Architecture

This time, we consider the difference in performance for 32-bit and 64-bit implementations of Ed25519. Essentially, 64-bit implementations breaks field elements into 5 limbs using *radix* 2^{51}. 32-bit implementations, in the other hand, breaks them into 10 limbs using the radix-$2^{\{25,26\}}$ representation. Figures 6.1, 6.2 and 6.3 show that performance halves (performance decreases by a factor of 2) for key generation, signing and verification when building the library for 32-bit targets. Which is expected, considering that the field element arithmetic is optimized for specific radix-β number representations. Because we wanted to emphasize the difference in performance for radix-β representations, we decided to exclude our implementation, which only supports the 64-bit version.

6.1.3 Overall

We now focus on performance comparisons between Ed25519 implementations in the following languages: Rust, C, Python, Java and Go.

Rust's performance. The C language is known to be fast and efficient. Thus, we set *ed25519-donna* as a performance baseline. For all Ed25519 operations, our implementation (*ed25519-fun*) has performed as well as the *ed25519-donna* implementation with similar optimization configurations. Also, it outperformed both Java implementations, Go's *crypto* library and *ed25519 (orlp)*, likely due to different field element representations and optimizations choices.

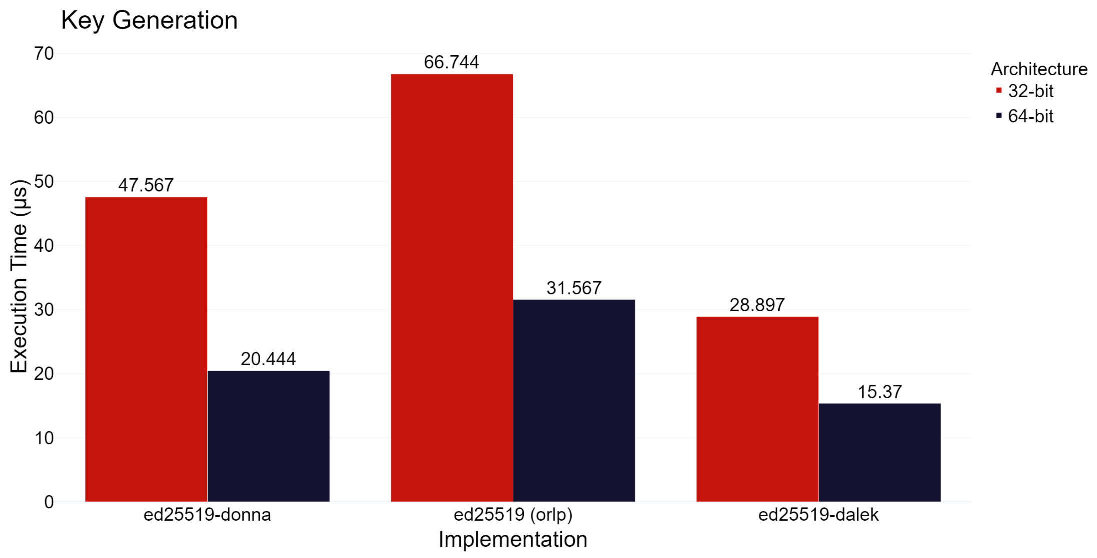

Figure 6.1: Key generation benchmarks involving 32-bit and 64-bit builds of three Ed25519 implementations.

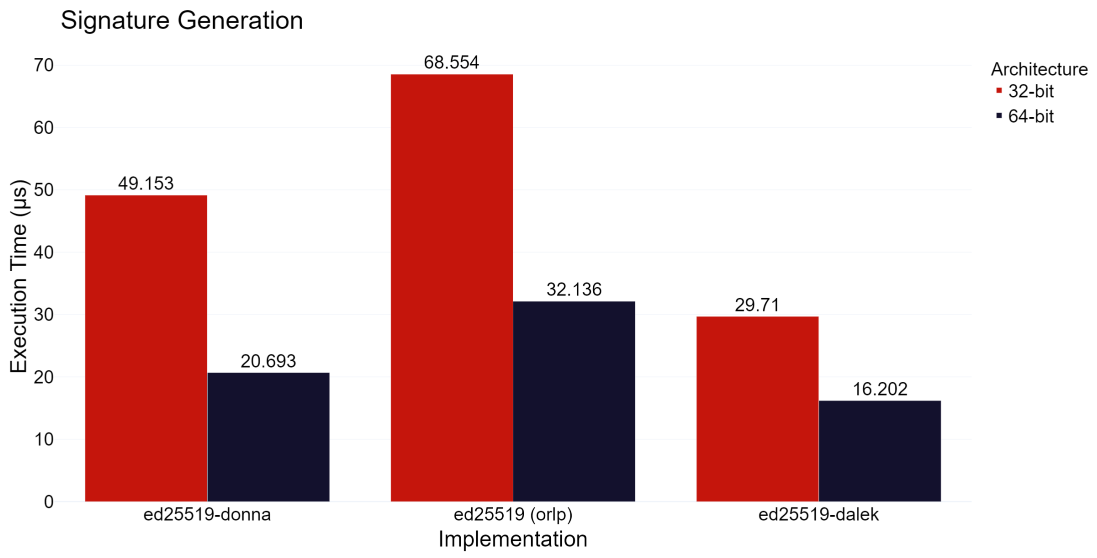

Figure 6.2: Signature generation benchmarks involving 32-bit and 64-bit builds of three Ed25519 implementations.

The following observations derived from Figures 6.4, 6.5 and 6.6, which illustrates the execution time (in microseconds) for the selected implementations:

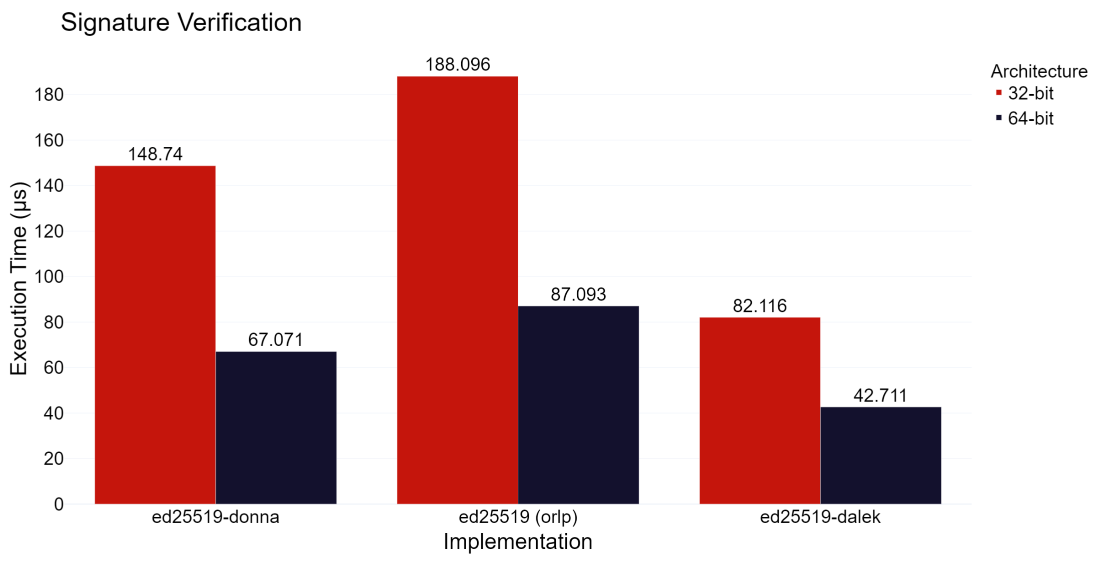

Figure 6.3: Signature verification benchmarks involving 32-bit and 64-bit builds of three Ed25519 implementations.

- *ed25519-dalek*'s experimental SIMD backend has the best signature verification performance.

- *ed25519-donna* with SSE2 instruction sets offers the fastest key generation and signature generation.

- Rust is capable of matching C's performance. *ed25519-fun* and *ed25519-donna*, for instance, shows similar performance of key generation and signature generation. The same can be applied for implementations written in Go, which is known for being efficient and reliable. *ed25519 (oasis)*, for example, has great performance for key generation (14.068 µs) and signature generation (14.921 µs).

- *pynacl* is a Python binding to *libsodium*, and *pyca*, in the other hand, is a Python binding to OpenSSL. We assume that the added overhead (binding) is insignificant, since *pyca* and *pynacl* have similar execution times in comparison to other implementations in C and Rust.

- The *Bouncy Castle* library (in Java), *ed25519-java* and the *crypto* library (in Go) placed last in our tests likely due to field element representations, which can increase performance considerably (as seen in Section 6.1.2).

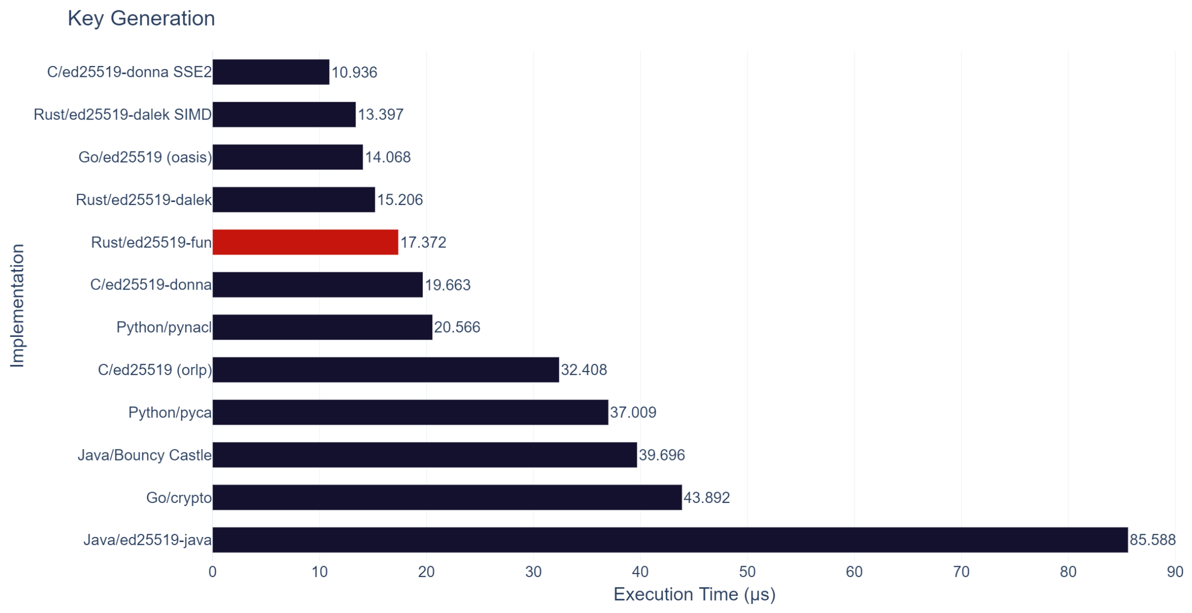

Figure 6.4: Key generation benchmarks for Ed25519 implementations in various languages.

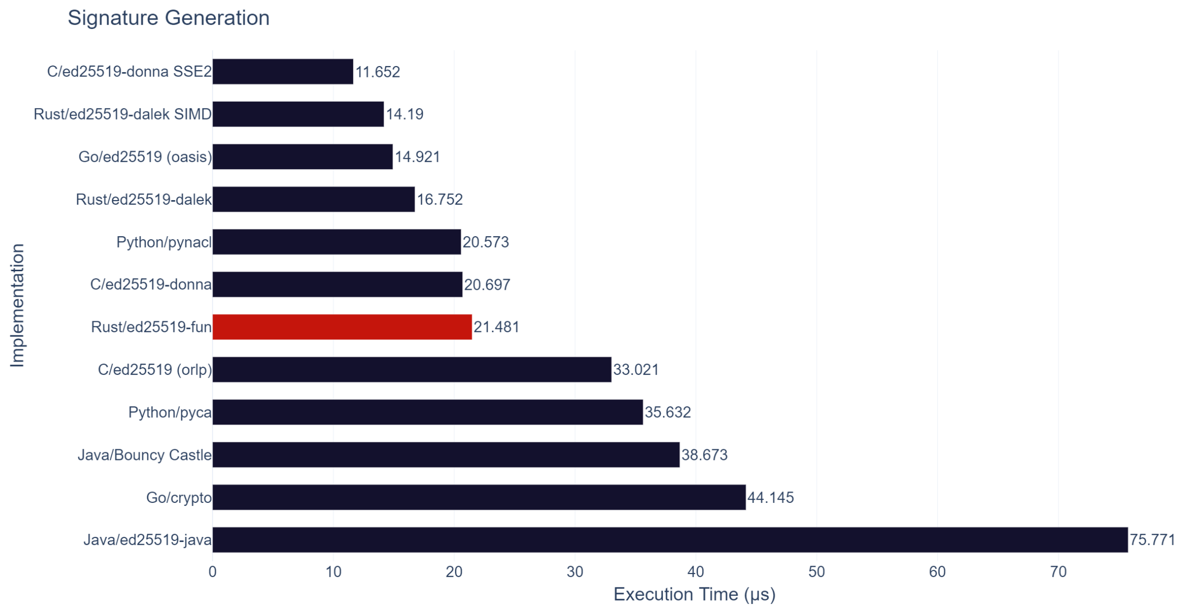

Figure 6.5: Signature generation benchmarks for Ed25519 implementations in various languages.

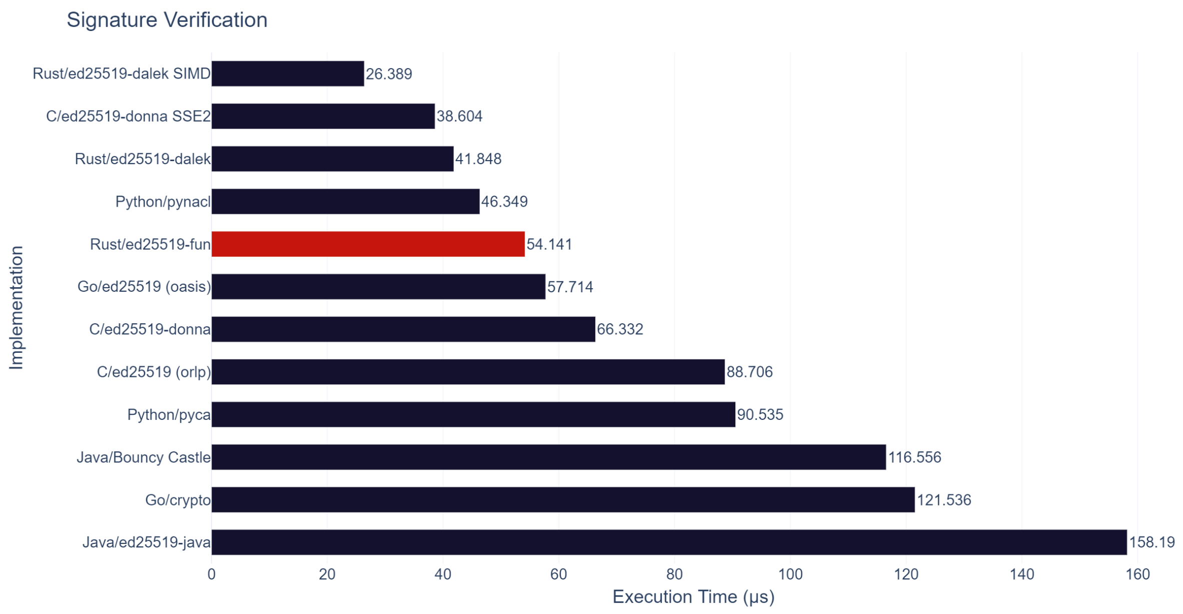

Figure 6.6: Signature verification benchmarks for Ed25519 implementations in various languages.

As seen in our benchmarks of Ed25519 implementations, Rust offers high performance, while providing desirable features for cryptographic applications, such as memory safety.

6.1.4 Memory Usage

Table 6.2 shows peak memory consumption reported by the Massif tool, which encouraged the following observations:

- The implementations in Java clearly consumes more memory.
- *ed25519 (orlp)* has the lowest memory consumption.
- Although the *total memory* consumed ($\approx$ 750 MiB) by Go implementations was $\approx 15\times$ higher than implementations in Python, its *stack* + *heap* consumption ($\approx$ 85 KiB) was only $\approx 3\times$ higher than implementations in Rust.
- *ed25519-donna*'s *total memory* consumption is 9.7 MiB, which surpasses *ed25519-fun* and *ed25519-dalek* by $\approx$ 1.7 MiB. Regarding *stack* + *heap*, however, Rust (*ed25519-dalek* and *ed25519-fun*) showed $\approx 4\times$ more memory usage than C.

- *ed25519-dalek* with the *simd_backend* feature enabled is worth considering: *ed25519-dalek SIMD* showed an increase in *total memory* usage of 0.2 MiB and significant improvement in signature verification speed. Whereas *ed25519-donna* did not show significant changes in regards to peak memory usage after enabling *SSE2* instructions.

Table 6.2: Peak memory usage reported by Massif for various implementations.

Implementation	**Total Memory** (*--pages-as-heap*)	**Stack + Heap** (*--stacks*)
Rust/ed25519-fun	8 MiB	29.7 KiB
Rust/ed25519-dalek	8 MiB	29.1 KiB
Rust/ed25519-dalek SIMD	8.2 MiB	90.7 KiB
C/ed25519-donna	9.7 MiB	7.5 KiB
C/ed25519-donna SSE2	9.7 MiB	7.5 KiB
C/ed25519 (orlp)	4.2 MiB	7.2 KiB
Python/pynacl	37.3 MiB	912.1 KiB
Python/pyca	52.7 MiB	2700 KiB
Go/crypto	750.2 MiB	85.4 KiB
Go/ed25519 (oasis)	750.3 MiB	85.2 KiB
Java/ed25519-java	15200 MiB	25400 KiB
Java/Bouncy Castle	15200 MiB	25600 KiB

6.2 Security

The Rust language has features that are appealing for cryptographic implementations, such as:

Expressiveness. Rust's type system allows for writing expressive programs. In this context, creating layers of abstraction and traits converges into lines of code that are easier to write and read.

Memory safety. *Ownership* is the main Rust feature, which enables memory management [53]. The Rust documentation states three ownership rules: (i) each value has an *owner*, (ii) there can only be one owner at a time, and lastly, (iii) the value is dropped once the owner goes out of scope [54]. These rules implies the following:

- There is no pointer aliasing in Rust (one owner at a time).
- Variables can only be either copied or moved (*borrow*).
- When the value is dropped its memory is no longer accessible.

The compiler knows when the allocated memory should be freed (*lifetime* construct). Use-after-free, double-free and dangling pointers are prevented at compile time [53]. Also, buffer overflow is prevented with built-in bounds checking.

Security measures considered for the Ed25519 implementation in Rust included the following:

Constant-time operations. Making sure cryptographic implementations are isochronous is important. Underlying field arithmetic and scalar multiplications should operate in constant-time to avoid side-channel attacks [55]. Our implementation approaches constant-time operations for key generation and message signing. An example of this approach is the use of fixed-window scalar multiplications (Section 5.1.2), which attempts constant-time execution. The *subtle* library [56], for instance, is another attempt that provides constant-time implementations of bitwise operations.

Zeroization. As an exploit mitigation technique, we used the *zeroize* library [57] to ensure that secret data is no longer accessible. This library performs memory zeroing while avoiding compiler optimizations.

Usability. The usability of cryptographic libraries is essential for preventing API misuse. Simplicity, easy to comprehend documentation and secure code examples were emphasized as important usability qualities for cryptographic APIs [58]–[60]. Therefore, we adopted the documentation of a simple and structured interface, that provides distinct types for the public key, secret key and signature.

Lastly, we tried classifying our implementation based on security properties exposed by Chalkias et al. [34] and Brendel et al. [39]:

EUF-CMA (existential unforgeability under chosen message attacks). For a number of selected messages and its respective signatures, the attacker should not be able to efficiently output a signature $\sigma*$ for a new message $m*$ that can be verified under the same public key, s.t. $m* \notin \{m_i\}_N^{i=1}$.

SUF-CMA (strong unforgeability under chosen message attacks). This definition is stronger than the EUF-CMA property. For a number of selected (message m_i, signature σ_i) pairs, the attacker should not be able to output ($m*$, $\sigma*$), s.t. $(m*, \sigma*) \notin \{m_i, \sigma_i\}_N^{i=1}$. This implies that a valid signature for an old message cannot be generated, except with negligible probability. This is useful for blockchain applications.

BS (binding signatures). The signer cannot generate two messages (m, $m*$) that can be verified under the same signature σ and public key pk, except with negligible probability. This implies that the signature is bound to the message and the non-repudiation property is held.

SBS (strongly binding signatures). The attacker cannot output valid (m_1, pk_1, $\sigma*$) and (m_2, pk_2, $\sigma*$), s.t. either $m_1 \neq m_2$ or $pk_1 \neq pk_2$ is true. This is an stronger notion than BS, and implies that the signature is also bound to the public key.

Table 6.3: Security properties satisfied by Ed25519 implementations. Adapted from: Chalkias et al. [34]. The test vectors, together with the explanation, can found in Appendix A.

Library	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**SUF-CMA**	**SBS**	**cofactored**
Bouncy Castle	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
ed25519-dalek	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
ed25519-donna	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗
go/crypto	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
libsodium	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗
pyca	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
ed25519-fun	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗

Appendix A gives an explanation for the test vectors. The following analysis can be drawn from Table 6.3 [34]:

- All libraries implements the cofactorless variant of EdDSA. We also considered the cofactorless version.
- Most implementations offer SUF-CMA security, with the exception of our implementation (*ed25519-fun*) and *ed25519-donna*.
- *libsodium* checks vectors 0, 1 and 2, which leads to SBS security.

Although the implementation that provides the *strongest notion of security* satisfies the SUF-CMA and SBS properties (e.g. *libsodium*), it is sufficient for a signature scheme to only satisfy the EUF-CMA property [34]. Therefore, our implementation meets the standard EUF-CMA security property.

6.3 Conclusions

This chapter has demonstrated that Rust is competitive considering its performance and security properties. Our Ed25519 implementation in Rust has respectable execution times and memory consumption, which matched Ed25519 libraries implemented in C. We have also performed benchmarks for comparing 32-bit and 64-bit implementations, and showed the impact of precomputed values in scalar multiplications. Further, we discussed security with regard to the implementation (i.e. constant-time operations, usability), Ed25519 algorithm (i.e. SUF-CMA, SBS) and Rust (i.e. memory safety).

In order to achieve memory safety, programs can be written in Java, C# or F#, which performs runtime checks and garbage collection. However, the consequential overhead of such memory management is undesired for programs that aims for efficiency. Rust is not a language for rapid scripting or tasks that does not require performance, such as Python. Though its features are desirable for cryptographic implementations, where memory safety, efficiency and expressiveness are important.

Chapter 7

Conclusions

Taking the RFC 8032 and existing Curve25519 and Ed25519 implementations in C, Java and Rust and implementing our own is straightforward. However, focusing on the bigger picture and understanding the overall functionality of the Ed25519 system was not the only goal. As well as describing the internal functions of the Ed25519 library, we also presented attributes related to modularity, readability and ease of use.

In addition to the implementation, an evaluation from a security/performance perspective was also a part of the contribution. Performance wise, Rust presented execution times and memory consumption comparable to implementations written in C. We have also investigated the use of basepoint precomputed table and field element representations tailored for 32-bit and 64-bit platforms. This investigation showed significant impact in performance: $\approx 80\%$ performance increase for key generation and signing - using the base point precomputed table; $\approx 2\times$ performance increase for 64-bit representations. Security wise, we mainly discussed the unforgeability and non-repudiation (binding signatures) properties, constant-time operations, zeroization and memory-safety. Thanks to Rust the implementation is type and memory safe and does not compromise on performance. Consequently, Rust is suitable for writing cryptographic applications and expressive code that runs efficiently.

Therefore, we were able to explore the understanding of individual functions, optimizations and security properties for Ed25519. This exploration showed the correlation between security, efficiency and cryptographic applications. Furthermore, the discrepancy between specification and the actual implementation was emphasized, where knowledge about cryptographic implementations is important.

There are two remarks worth raising, especially because we believe they would better the execution of this work. (i) Understanding the underlying mathematical details and design helps with the creation of layers of abstractions within the implementation, and also the writing of better documentations/comments and tests. (ii) Simply let the idea of *cryptographic implementation* dictate the quality and focus of the evaluation process: security, correctness and efficiency are essential.

7.1 Future Works

The Ed25519 implementation can be expanded in the following ways: (i) include batch verification functions; (ii) implement layers of abstraction for internal mathematical representations; (iii) include optimization configurations for scalar multiplications and support for 32-bit and 64-bit platforms; (iv) include the necessary modifications and verification for signature malleability and security notions, such as strong unforgeability and strongly binding signatures; (v) optimize arithmetic operations by using a vectorized SIMD implementation. Beyond that, we glimpse the idea of implementing other instances of EdDSA (e.g. Ed448) and analyzing its contrasting properties in comparison to Ed25519; also, the implementation of *ristretto255*[1] for Ed25519; and lastly, we consider the addition of support for other environments (resource-limited devices) and the *no_std* feature, plus benchmarks considering energy and memory consumption and speed.

[1]https://ristretto.group/ristretto.html

Bibliography

[1] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[2] N. Koblitz, "Elliptic Curve Cryptosystems," en, *Mathematics of Computation*, vol. 48, pp. 203–209, 1987.

[3] A. Menezes, P. Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[4] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," en, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas Texas USA: ACM, Oct. 2017, pp. 1789–1806. [Online]. Available: `https://dl.acm.org/doi/10.1145/3133956.3134043` (visited on 05/20/2021).

[5] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren, "Practical Realisation and Elimination of an ECC-Related Software Bug Attack," en, in *Topics in Cryptology – CT-RSA 2012*, O. Dunkelman, Ed., vol. 7178, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 171–186. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-27954-6_11` (visited on 05/27/2021).

[6] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why does cryptographic software fail? a case study and open problems," in *Proceedings of 5th Asia-Pacific Workshop*

on Systems, ser. APSys '14, New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: `https://doi.org/10.1145/2637166.2637237`.

[7] A. Moon, *ed25519-donna*. GitHub, 2015. [Online]. Available: `https://github.com/floodyberry/ed25519-donna` (visited on 11/24/2019).

[8] I. Lovecruft and H. de Valence, *ed25519-dalek*. [Online]. Available: `https://github.com/dalek-cryptography/ed25519-dalek` (visited on 05/20/2021).

[9] str4d, *Ed25519-java*. [Online]. Available: `https://github.com/str4d/ed25519-java` (visited on 05/20/2021).

[10] *Zero Cost Abstractions*. [Online]. Available: `https://doc.rust-lang.org/stable/embedded-book/static-guarantees/zero-cost-abstractions.html` (visited on 05/24/2021).

[11] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the Rust programming language," en, *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–34, Jan. 2018. [Online]. Available: `https://dl.acm.org/doi/10.1145/3158154` (visited on 05/17/2021).

[12] E. Reed, "Patina: A Formalization of the Rust Programming Language," en, p. 37, 2015.

[13] *Rust By Example*. [Online]. Available: `https://doc.rust-lang.org/stable/rust-by-example/` (visited on 05/19/2021).

[14] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*, en, 2nd. New York: Wiley, 1996.

[15] W. Diffie and M. Hellman, "New directions in cryptography," en, *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[16] A. Kerckhoffs, "Le Cryptographie Militaire," *Journal des sciences militaires*, vol. IX, pp. 161–191, 1883.

[17] N. P. Smart, *Cryptography Made Simple*, en, 1st, ser. Information security and cryptography. Cham Heidelberg New York Dordrecht London: Springer, 2016.

[18] C. E. Shannon, “Communication Theory of Secrecy Systems,” en, *Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[19] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: `http://doi.acm.org/10.1145/359340.359342` (visited on 05/19/2021).

[20] R. C. Merkle, “Secure communications over insecure channels,” en, *Communications of the ACM*, vol. 21, no. 4, pp. 294–299, Apr. 1978. [Online]. Available: `https://dl.acm.org/doi/10.1145/359460.359473` (visited on 12/02/2020).

[21] D. Stinson and M. Paterson, *Cryptography: Theory and Practice*, 4th. CRC Press, Taylor & Francis Group, 2018.

[22] S. Azad and A.-S. K. Pathan, *Practical Cryptography: Algorithms and Implementations Using C++*, en, 1st. Auerbach Publications, 2014.

[23] A. H. Koblitz, N. Koblitz, and A. Menezes, “Elliptic curve cryptography: The serpentine course of a paradigm shift,” en, *Journal of Number Theory*, vol. 131, no. 5, pp. 781–814, 2011.

[24] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*, en, 1st. Heidelberg; New York: Springer, 2010.

[25] L. Prokop, “Elliptic Curve Cryptography: Theory for EdDSA,” en, p. 31, 2016.

[26] J. Wellens, “A FRIENDLY INTRODUCTION TO GROUP THEORY,” en, p. 16, [Online]. Available: `https://math.mit.edu/~jwellens/Group%20Theory%20Forum.pdf` (visited on 05/24/2021).

[27] L. Finotti, “A GENTLE INTRODUCTION TO NUMBER THEORY AND CRYPTOGRAPHY,” en, p. 60, 2009.

[28] D. R. Hankerson, S. A. Vanstone, and A. J. Menezes, *Guide to elliptic curve cryptography*, en, 1st. New York: Springer, 2004.

[29] R. Afreen and S. Mehrotra, "A Review on Elliptic Curve Cryptography for Embedded Systems," en, *International Journal of Computer Science and Information Technology*, vol. 3, no. 3, pp. 84–103, Jun. 2011. [Online]. Available: `http://www.airccse.org/journal/jcsit/0611csit07.pdf` (visited on 04/16/2021).

[30] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, and C. Peters, "Twisted Edwards Curves," en, in *Progress in Cryptology – AFRICACRYPT 2008*, S. Vaudenay, Ed., vol. 5023, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 389–405. [Online]. Available: `http://link.springer.com/10.1007/978-3-540-68164-9_26` (visited on 11/03/2019).

[31] H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson, "Twisted Edwards Curves Revisited," en, in *Advances in Cryptology - ASIACRYPT 2008*, J. Pieprzyk, Ed., vol. 5350, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 326–343. [Online]. Available: `http://link.springer.com/10.1007/978-3-540-89255-7_20` (visited on 11/03/2019).

[32] J. L. Vagle, "A Gentle Introduction to Elliptic Curve Cryptography," en, p. 25, 2000.

[33] S. Josefsson and I. Liusvaara, *RFC 8032 - Edwards-Curve Digital Signature Algorithm (EdDSA)*, en, 2017. [Online]. Available: `https://tools.ietf.org/html/rfc8032` (visited on 05/20/2021).

[34] K. Chalkias, F. Garillot, and V. Nikolaenko, "Taming the Many EdDSAs," en, in *Security Standardisation Research*, T. van der Merwe, C. Mitchell, and M. Mehrnezhad, Eds., vol. 12529, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 67–90. [Online]. Available: `http://link.springer.com/10.1007/978-3-030-64357-7_4` (visited on 04/02/2021).

[35] *Go cryptography.* [Online]. Available: `https://github.com/golang/crypto` (visited on 05/24/2021).

[36] *Pynacl: Python binding to the libsodium library.* [Online]. Available: `https://github.com/pyca/pynacl` (visited on 05/23/2021).

[37] *Libsodium.* [Online]. Available: `https://github.com/jedisct1/libsodium` (visited on 05/24/2021).

[38] A. Liu and P. Ning, "TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks," en, in *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, St. Louis, MO, USA: IEEE, Apr. 2008, pp. 245–256. [Online]. Available: `http://ieeexplore.ieee.org/document/4505478/` (visited on 04/05/2021).

[39] J. Brendel, C. Cremers, D. Jackson, and M. Zhao, "The Provable Security of Ed25519: Theory and Practice," en, p. 26, Oct. 2020.

[40] D. J. Bernstein, T. Lange, and P. Schwabe, "The Security Impact of a New Cryptographic Library," en, in *Progress in Cryptology – LATINCRYPT 2012*, vol. 7533, Series Title: Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 159–176. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-33481-8_9` (visited on 05/20/2021).

[41] *The rustonomicon.* [Online]. Available: `https://doc.rust-lang.org/nomicon/` (visited on 05/15/2021).

[42] *Test vectors.* [Online]. Available: `https://ed25519.cr.yp.to/python/sign.input` (visited on 05/24/2021).

[43] K.-I. D. Kyriakou, "Improving C/C++ Open Source Software Discoverability by Utilizing Rust and Node.js Ecosystems," en, p. 12, Jun. 2018.

[44] M. Baranowski, S. He, and Z. Rakamarić, "Verifying Rust Programs with SMACK," en, in *Automated Technology for Verification and Analysis*, S. K. Lahiri and C. Wang, Eds., vol. 11138, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2018, pp. 528–535. [Online]. Available: `http://link.springer.com/10.1007/978-3-030-01090-4_32` (visited on 05/16/2021).

[45] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards Memory Safe Enclave Programming with Rust-SGX," en, in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, London United Kingdom: ACM, Nov. 2019, pp. 2333–2350. [Online]. Available: `https://dl.acm.org/doi/10.1145/3319535.3354241` (visited on 05/16/2021).

[46] *Ed25519-fun*. [Online]. Available: `https://github.com/yuzonightly/ed25519-fun` (visited on 05/29/2021).

[47] *Oasisprotocol/ed25519*. [Online]. Available: `https://github.com/oasisprotocol/ed25519` (visited on 02/16/2021).

[48] *Bouncy Castle*. [Online]. Available: `https://www.bouncycastle.org/` (visited on 05/29/2021).

[49] *Ed25519*. [Online]. Available: `https://github.com/orlp/ed25519` (visited on 05/29/2021).

[50] *Pyca/cryptography*. [Online]. Available: `https://github.com/pyca/cryptography` (visited on 05/29/2021).

[51] Y.-F. Chen, C.-H. Hsu, H.-H. Lin, P. Schwabe, M.-H. Tsai, B.-Y. Wang, B.-Y. Yang, and S.-Y. Yang, "Verifying Curve25519 Software," en, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, Scottsdale Arizona USA: ACM, Nov. 2014, pp. 299–309. [Online]. Available: `https://dl.acm.org/doi/10.1145/2660267.2660370` (visited on 03/31/2021).

[52] D. J. Bernstein, C. Chuengsatiansup, and T. Lange, "Double-base scalar multiplication revisited," en, p. 38, 2017.

[53] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto, "Ownership is theft: Experiences building an embedded OS in rust," en, in *Proceedings of the 8th Workshop on Programming Languages and Operating Systems - PLOS '15*, Monterey, California: ACM Press, 2015, pp. 21–26. [Online].

Available: `http://dl.acm.org/citation.cfm?doid=2818302.2818306` (visited on 03/31/2021).

[54] *The Rust Programming Language*. [Online]. Available: `https://doc.rust-lang.org/book/` (visited on 05/19/2021).

[55] S. Bhunia and M. Tehranipoor, *Hardware Security: A Hands-on Learning Approach*. Elsevier Science, 2018. [Online]. Available: `https://books.google.pt/books?id=wIp1DwAAQBAJ` (visited on 05/24/2021).

[56] *Crate subtle*. [Online]. Available: `https://docs.rs/subtle/2.4.0/subtle/index.html` (visited on 05/15/2021).

[57] *Crate zeroize*. [Online]. Available: `https://docs.rs/zeroize/1.3.0/zeroize/` (visited on 05/15/2021).

[58] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the Usability of Cryptographic APIs," en, in *2017 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA: IEEE, May 2017, pp. 154–171. [Online]. Available: `http://ieeexplore.ieee.org/document/7958576/` (visited on 04/10/2021).

[59] K. Mindermann, P. Keck, and S. Wagner, "How Usable Are Rust Cryptography APIs?" en, in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Lisbon: IEEE, Jul. 2018, pp. 143–154. [Online]. Available: `https://ieeexplore.ieee.org/document/8424966/` (visited on 04/10/2021).

[60] K. Mindermann and S. Wagner, "Usability and Security Effects of Code Examples on Crypto APIs," en, in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, Belfast: IEEE, Aug. 2018, pp. 1–2. [Online]. Available: `https://ieeexplore.ieee.org/document/8514203/` (visited on 04/10/2021).

Appendix A

Test Vectors

Listing A.1 shows the test vectors used in the security evaluation, found in [34]. There are 12 test vectors in total:

- *Notation: public key A; signature (R, S); hashed secret key h; group order L.*
- Tests vectors 0-3 passes both cofactored and cofactorless verification. Tests 0-2 have small R, A or both, whereas test 3 has mixed-order A and R.
- Test vector 4 passes cofactored but fails in cofactorless verification.
- Test 5 is rejected in cofactored verification that erroneously performs $(8h \ mod \ L)A$ instead of $8(hA)$.
- Tests 6-7 passes for libraries that accept non-canonical S or performs an incomplete check.
- Tests 8-9 have a non-canonical R; vector 8 passes if the library reduces R before hashing; vector 9 passes if the library does not reduce R before hashing.
- Vector 10-11 have a non-canonical A; test 10 passes if the library reduces A before hashing; test 11 passes if the library does not reduce A before hashing.

Listing A.1: Test vectors.

```
{
    message: 8c93255d71dcab10e8f379c26200f3c7bd5f09d9bc3068d3ef4edeb4853022b6
    public_key: c7176a703d4dd84fba3c0b760d10670f2a2053fa2c39ccc64ec7fd7792ac03fa
    signature: c7176a703d4dd84fba3c0b760d10670f2a2053fa2c39ccc64ec7fd7792ac037a0
    000000000000000000000000000000000000000000000000000000000000000
}
{
    message: 9bd9f44f4dcc75bd531b56b2cd280b0bb38fc1cd6d1230e14861d861de092e79
    public_key: c7176a703d4dd84fba3c0b760d10670f2a2053fa2c39ccc64ec7fd7792ac03fa
    signature: f7badec5b8abeaf699583992219b7b223f1df3fbbea919844e3f7c554a43dd43a
    5bb704786be79fc476f91d3f3f89b03984d8068dcf1bb7dfc6637b45450ac04
}
{
    message: aebf3f2601a0c8c5d39cc7d8911642f740b78168218da8471772b35f9d35b9ab
    public_key: f7badec5b8abeaf699583992219b7b223f1df3fbbea919844e3f7c554a43dd43
    signature: c7176a703d4dd84fba3c0b760d10670f2a2053fa2c39ccc64ec7fd7792ac03fa8
    c4bd45aecaca5b24fb97bc10ac27ac8751a7dfe1baff8b953ec9f5833ca260e
}
{
    message: 9bd9f44f4dcc75bd531b56b2cd280b0bb38fc1cd6d1230e14861d861de092e79
    public_key: cdb267ce40c5cd45306fa5d2f29731459387dbf9eb933b7bd5aed9a765b88d4d
    signature: 9046a64750444938de19f227bb80485e92b83fdb4b6506c160484c016cc1852f8
    7909e14428a7a1d62e9f22f3d3ad7802db02eb2e688b6c52fcd6648a98bd009
}
{
    message: e47d62c63f830dc7a6851a0b1f33ae4bb2f507fb6cffec4011eaccd55b53f56c
    public_key: cdb267ce40c5cd45306fa5d2f29731459387dbf9eb933b7bd5aed9a765b88d4d
    signature: 160a1cb0dc9c0258cd0a7d23e94d8fa878bcb1925f2c64246b2dee1796bed5125
    ec6bc982a269b723e0668e540911a9a6a58921d6925e434ab10aa7940551a09
}
{
    message: e47d62c63f830dc7a6851a0b1f33ae4bb2f507fb6cffec4011eaccd55b53f56c
    public_key: cdb267ce40c5cd45306fa5d2f29731459387dbf9eb933b7bd5aed9a765b88d4d
    signature: 21122a84e0b5fca4052f5b1235c80a537878b38f3142356b2c2384ebad4668b7e
    40bc836dac0f71076f9abe3a53f9c03c1ceeeddb658d0030494ace586687405
}
{
    message: 85e241a07d148b41e47d62c63f830dc7a6851a0b1f33ae4bb2f507fb6cffec40
    public_key: 442aad9f089ad9e14647b1ef9099a1ff4798d78589e66f28eca69c11f582a623
    signature: e96f66be976d82e60150baecff9906684aebb1ef181f67a7189ac78ea23b6c0e5
    47f7690a0e2ddcd04d87dbc3490dc19b3b3052f7ff0538cb68afb369ba3a514
}
{
    message: 85e241a07d148b41e47d62c63f830dc7a6851a0b1f33ae4bb2f507fb6cffec40
    public_key: 442aad9f089ad9e14647b1ef9099a1ff4798d78589e66f28eca69c11f582a623
    signature: 8ce5b96c8f26d0ab6c47958c9e68b937104cd36e13c33566acd2fe8d38aa19427
    e71f98a4734e74f2f13f06f97c20d58cc3f54b8bd0d272f42b695dd7e89a8c2
}
{
    message: 9bedc267423725d473888631ebf45988bad3db83851ee85c85e241a07d148b41
    public_key: f7badec5b8abeaf699583992219b7b223f1df3fbbea919844e3f7c554a43dd43
    signature: ecffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff0
    3be9678ac102edcd92b0210bb34d7428d12ffc5df5f37e359941266a4e35f0f
}
{
    message: 9bedc267423725d473888631ebf45988bad3db83851ee85c85e241a07d148b41
    public_key: f7badec5b8abeaf699583992219b7b223f1df3fbbea919844e3f7c554a43dd43
    signature: ecffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffc
    a8c5b64cd208982aa38d4936621a4775aa233aa0505711d8fdcfdaa943d4908
}
{
    message: e96b7021eb39c1a163b6da4e3093dcd3f21387da4cc4572be588fafae23c155b
    public_key: ecffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
    signature: a9d55260f765261eb9b84e106f665e00b867287a761990d7135963ee0a7d59dca
    5bb704786be79fc476f91d3f3f89b03984d8068dcf1bb7dfc6637b45450ac04
}
{
    message: 39a591f5321bbe07fd5a23dc2f39d025d74526615746727ceefd6e82ae65c06f
    public_key: ecffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
    signature: a9d55260f765261eb9b84e106f665e00b867287a761990d7135963ee0a7d59dca
    5bb704786be79fc476f91d3f3f89b03984d8068dcf1bb7dfc6637b45450ac04
}
```

Appendix B

Group Element Arithmetic

Listings B.1, B.2 and B.3 shows the code that performs point addition, subtraction and doubling, respectively.

Listing B.1: Implementation of addition for extended points (P^3).

```rust
impl Add<Cached> for P3 {
    type Output = P1P1;

    fn add(self, p: Cached) -> P1P1 {
        let YpX = self.Y + self.X; // Y1 + X1
        let YmX = self.Y - self.X; // Y1 - X1
        let B = YpX * p.YpX; // (Y1 + X1) * (Y2 + X2)
        let b = YmX * p.YmX; // (Y1 - X1) * (Y2 - X2)
        let c = p.T2d * self.T;
        let d = self.Z * p.Z;
        let e = d + d;
        let x = B - b;
        let y = B + b;
        let z = e + c;
        let t = e - c;

        P1P1 {
            X: x,
            Y: y,
            Z: z,
            T: t,
        }
    }
}

```

```rust
impl Add<Precomp> for P3 {
    type Output = P1P1;

    fn add(self, p: Precomp) -> P1P1 {
        let YpX = self.Y + self.X; // Y1 + X1
        let YmX = self.Y - self.X; // Y1 - X1
        let a = YpX * p.YpX; // (Y1 + X1) * (Y2 - X2)
        let b = YmX * p.YmX; // (Y1 - X1) * (Y2 - X2)
        let c = p.XY2d * self.T; // D * 2 * (X2 * Y2) * T
        let d = self.Z + self.Z; // Z1 + Z1
        let x = a - b;
        let y = a + b;
        let z = d + c;
        let t = d - c;

        P1P1 {
            X: x,
            Y: y,
            Z: z,
            T: t,
        }
    }
}
```

Listing B.2: Implementation of subtraction for extended points (P^3).

```rust
impl Sub<Cached> for P3 {
    type Output = P1P1;

    fn sub(self, p: Cached) -> P1P1 {
        let YpX = self.Y + self.X;
        let YmX = self.Y - self.X;
        let a = YpX * p.YmX;
        let b = YmX * p.YpX;
        let c = p.T2d * self.T;
        let d = self.Z * p.Z;
        let e = d + d;
        let x = a - b;
        let y = a + b;
        let z = e - c;
        let t = e + c;

        P1P1 {
            X: x,
            Y: y,
            Z: z,
                T: t,
            }
        }
}

impl Sub<Precomp> for P3 {
    type Output = P1P1;

    fn sub(self, p: Precomp) -> P1P1 {
        let YpX = self.Y + self.X; // Y1 + X1
        let YmX = self.Y - self.X; // Y1 - X1
        let a = YpX * p.YmX; // (Y1 + X1) * (Y2 - X2)
        let b = YmX * p.YpX; // (Y1 - X1) * (Y2 - X2)
        let c = p.XY2d * self.T; // 2 * D * (X2 * Y2) * T
        let d = self.Z + self.Z; // Z1 + Z1
        let x = a - b;
        let y = a + b;
        let z = d - c;
        let t = d + c;

        P1P1 {
            X: x,
            Y: y,
            Z: z,
            T: t,
        }
    }
}
```

Listing B.3: Implementation of doubling for projective points (P^2).

```
pub fn double(&self) -> P1P1 {
    let A = self.X.square();
    let B = self.Y.square();
    let C = self.Z.double_square();
    let Y_plus_X = self.X + self.Y;
    let a = Y_plus_X.square();
    let y = A + B;
    let z = B - A;
    let x = a - y;
    let t = C - z;

    P1P1 {
        X: x,
        Y: y,
        Z: z,
            T: t,
        }
}
```

Appendix C

Scalar Multiplication

Listing C.1 shows the code that performs $h = a \cdot B$, s.t. a is a scalar and B is the base point (Ed25519 parameter).

Listing C.1: Scalar multiplication in constant-time.

```rust
pub fn select(pos: usize, b: i8) -> Precomp {
    // Check if b is negative (1u8: true, 0u8: false)
    let negative = (b as u8) >> 7;

    // If b is negative:
    // we have b - (b << 1), which results in its absolute value.
    // If b is positive:
    // we have b - 0x00 = b.
    let absolute: u8 = (b - (((-(negative as i8)) & b) << 1)) as u8;
    let mut t = Precomp::zero();

    // Assign value based on pos (exponent of base 256) and
    // absolute ([1, 8]).
    // Ex.: if pos = 1 and absolute = 8, t is assigned (8 * 256^{1} * B).
    t.conditional_assign(&PRECOMP_BASE[pos][0], equal(absolute, 1u8).into());
    t.conditional_assign(&PRECOMP_BASE[pos][1], equal(absolute, 2u8).into());
    t.conditional_assign(&PRECOMP_BASE[pos][2], equal(absolute, 3u8).into());
    t.conditional_assign(&PRECOMP_BASE[pos][3], equal(absolute, 4u8).into());
    t.conditional_assign(&PRECOMP_BASE[pos][4], equal(absolute, 5u8).into());
    t.conditional_assign(&PRECOMP_BASE[pos][5], equal(absolute, 6u8).into());
    t.conditional_assign(&PRECOMP_BASE[pos][6], equal(absolute, 7u8).into());
    t.conditional_assign(&PRECOMP_BASE[pos][7], equal(absolute, 8u8).into());

    // Negative of t.
    let negative_t = Precomp {
        YpX: t.YmX,
        YmX: t.YpX,
        XY2d: t.XY2d.negate(),
    };

    // Assign negative of t if b is negative.
    t.conditional_assign(&negative_t, negative.into());

    t
}

/// Converts a to radix 16 representation.
/// a: a[0] + 256 * a[1] + 256^{2} * a[2] + ...
/// + 256^{31} * a[31].
fn radix16(a: &[u8]) -> [i8; 64] {
    let mut e = [0i8; 64];

    // Split each byte into two 4-bit values.
    // [e[0]..e[62]] values ranges between 0 and 15.
    // e[63] ranges between 0 and 7.
    for i in 0..32 {
        e[2 * i + 0] = (a[i] & 15) as i8;
        e[2 * i + 1] = ((a[i] >> 4) & 15) as i8;
    }

    // Convert each value from e to [-8..7].
    let mut carry: i8 = 0;
    // 10 -> -6, 9 -> -7, 8 -> -8...
    for i in 0..63 {
        e[i] += carry;
        carry = e[i] + 8;
        carry >>= 4;
        e[i] -= carry << 4;
    }
    e[63] += carry;

    e
}

/// Performs scalar multiplication h = a * B.
/// a: a[0] + 256 * a[1] + 256^{2} * a[2] + ...
/// + 256^{31} * a[31].
/// B: Ed25519 base point (x, 4/5) with positive x.
/// Uses precomputed values.
pub fn scalar_multiply(a: &[u8]) -> P3 {
    let e: [i8; 64] = Precomp::radix16(a);
    let mut t: Precomp;

    let mut h = P3::zero();
    // 64 table lookups
    // 64 point additions
    for i in (1..64).step_by(2) {
        t = Precomp::select(i / 2, e[i]);
        h = (h + t).to_P3();
    }

    // 4 doublings
    h = h
        .double()
        .to_P2()
        .double()
        .to_P2()
        .double()
        .to_P2()
        .double()
        .to_P3();

    // 64 point lookups
    // 64 point additions
    for i in (0..64).step_by(2) {
        t = Precomp::select(i / 2, e[i]);
        h = (h + t).to_P3();
    }
    h
}
```

Appendix D

Sliding-window Scalar Multiplication

Listing D.1 shows Code that performs $r = a \cdot A + b \cdot B$, s.t. a and b are scalars and A and B (base point) are points.

Listing D.1: Sliding-window scalar multiplication.

```
pub fn slide(a: &[u8]) -> [i8; 256] {
    let mut r = [0i8; 256];

    // Each bit in a has its own position in r.
    for i in 0..256 {
        r[i] = (1 & (a[i >> 3] >> (i & 7))) as i8;
    }

    for i in 0..256 {
        if r[i] != 0 {
            for b in 1..min(7, 256 - i) {
                if r[i + b] != 0 {
                    if r[i] + (r[i + b] << b) <= 15 {
                        r[i] += r[i + b] << b;
                        r[i + b] = 0;
                    } else if r[i] - (r[i + b] << b) >= -15 {
                        r[i] -= r[i + b] << b;
                        for k in i + b..256 {
                            if r[k] == 0 {
                                r[k] = 1;
                                break;
                            }
                            r[k] = 0;
                        }
                    } else {
                        break;
                    }
                }
            }
        }
    }

    r
}

pub fn double_scalar_multiply_vartime(a: &[u8], b: &[u8], A: P3) -> P2 {
    let aslide = P2::slide(a);
    let bslide = P2::slide(b);

    // A * I precomputation.
    // {A, 3A, 5A, 7A, 9A, 11A, 13A, 15A}.
    let mut AI = [Cached {
        YpX: FieldZero,
        YmX: FieldZero,
        Z: FieldZero,
        T2d: FieldZero,
    }; 8];
    AI[0] = A.to_Cached(); // A
    let A2 = A.double().to_P3(); // 2A
    for i in 1..8 {
        // 3A, 5A, 7A, ..., 15A
        AI[i] = (A2.add(AI[i - 1])).to_P3().to_Cached();
    }

    let mut r = P2::zero();
    let mut i: usize = 255;

    loop {
        if aslide[i] != 0 || bslide[i] != 0 {
            break;
        }

        if i == 0 {
            return r;
        }

        i -= 1;
    }

    loop {
        // Doubling chain.
        let mut t = r.double();

        if aslide[i] > 0 {
            t = t.to_P3() + AI[(aslide[i] / 2) as usize];
        } else if aslide[i] < 0 {
            t = t.to_P3() - AI[(-aslide[i] / 2) as usize];
        }

        if bslide[i] > 0 {
            t = t.to_P3() + BI[(bslide[i] / 2) as usize];
        } else if bslide[i] < 0 {
            t = t.to_P3() - BI[(-bslide[i] / 2) as usize];
        }

        r = t.to_P2();

        if i == 0 {
            return r;
        }

        i -= 1;
    }
}
```

Appendix E

Ed25519 Protocol

Listings E.1, E.2 and E.3 shows the implementation of the key generation, signature generation and signature verification functions, in that sequence.

Listing E.1: Secret key and public key generation functions.

```rust
/// Generates the secret key: 32 octets of cryptographically
/// secure random data.
pub(crate) fn generate_key() -> SecretKey {
    let mut sk = [0u8; 32];
    let mut csprng: ThreadRng = thread_rng();
    csprng.fill_bytes(&mut sk);
    SecretKey(sk)
}

/// Generates asymmetric keys: both public and secret,
/// as described in RFC 8032.
pub fn generate() -> Keypair {
    let secret = SecretKey::generate_key();

    // Hash the 32-byte private key using SHA-512, storing the digest in
    // a 64-octet large buffer h. Only the lower 32 bytes are
    // used for generating the public key.
    let h = {
        let mut hash = Sha512::default();
        hash.input(secret.0);
        let mut output = hash.result();
        // Lowest 3 bits of the first octet are cleared
        output[0] &= 248;
        // Highest bit of the last octet is cleared
        output[31] &= 63;
        // Second highest bit of the last octet is set
        output[31] |= 64;
        output
    };

    // Scalar multiplication: h * B.
    let point = Precomp::scalar_multiply(&h[0..32]);
    // Encode P2 point y coordinate.
    let public = PublicKey(point.encode());

    Keypair { secret, public }
}
```

Listing E.2: Signature generation function.

```
/// RFC 8032.
/// Generates the signature.
pub fn sign(&self, public: &PublicKey, message: &[u8]) -> Signature {
    // Hash the secret key using SHA-512.
    let h = {
        let mut hash = Sha512::new();
        hash.input(self.0);
        let mut output = hash.result();
        output[0] &= 248;
        output[31] &= 63;
        output[31] |= 64;
        output
    };

    // Compute SHA-512(prefix || PH(M)), where M is the
    // message to be signed and prefix is the second half of h.
    // Interpret the 64-octet digest as a little-endian integer r.
    let mut r = {
        let mut hash = Sha512::default();
        hash.input(&h[32..64]);
        hash.input(message);
        hash.result()
    };

    // Compute the point [r]B.  For efficiency, do this by first
    // reducing r modulo L, the group order of B.
    reduce(&mut r[..]);
    let R: P3 = Precomp::scalar_multiply(&r[0..32]);

    // Compute SHA512(enc(R) || A || PH(M)), and interpret the
    // 64-octet digest as a little-endian integer k.
    let mut k = {
        let mut hash = Sha512::default();
        hash.input(&R.encode());
        hash.input(public.0);
        hash.input(&message);
        hash.result()
    };
    reduce(&mut k[..]);

    // The signature.
    let mut signature = [0u8; 64];
    // Populate the second half of the signature with the
    // result of (r + k * s) mod L.
    multiply_add(&mut signature[32..64], &k[0..32], &h[0..32], &r);

    // Populate the first half of the signature with the
    // encoding of R.
    for (result_byte, source_byte) in &mut signature[0..32].iter_mut().zip(R.encode().iter(
        *result_byte = *source_byte;
    }
    Signature(signature)
}
```

Listing E.3: Signature verification function.

```
/// Verifies the signature.
pub fn verify(&self, message: &[u8], sig: &Signature) -> Result<(), Error> {
    let signature = sig.as_bytes();
    let s = &signature[32..64];

    if check_lt_l(s) {
        return Err(Error::InvalidSignature);
    }

    // Try to decode the public key into a P3 point.
    // Verification fails if decoding fails.
    let A = match P3::decode(self.0) {
        Some(point) => point,
        None => {
            return Err(Error::InvalidSignature);
        }
    };

    // Compute SHA512(R || A || PH(M)), and interpret the
    // 64-octet digest as a little-endian integer k.
    let mut k = {
        let mut hash = Sha512::default();
        hash.input(&signature[0..32]);
        hash.input(&self.0);
        hash.input(&message);
        hash.result()
    };
    reduce(&mut k);

    // Check the group equation [s]B = R + [k]A'.
    // Perform [s]B + [k]A'.
    let eq = P2::double_scalar_multiply_vartime(&k[..], s, A);
    // Check [s]B + [k]A' == R?
    if eq
        .encode()
        .as_ref()
        .iter()
        .zip(signature.iter())
        .fold(0, |acc, (x, y)| acc | (x ^ y))
        == 0
    {
        Ok(())
    } else {
        return Err(Error::SignatureMismatch);
    }
}
```