Property based testing of Rust Bulletproofs using Hacspec

Rasmus Kirk Jakobsen – 201907084
Anders Wibrand Larsen – 201906147
Advisor: Bas Spitters

2022-07-07

Bachelor report (15 ECTS) in Computer Science
Department of Computer Science, Aarhus University

Abstract:

Contents

1 Introduction:

2 Notation:

We denote scalar in lowercase (x), vectors in bold lowercase ($\mathbf{v}$), sets are capitalized (S), as are Elliptic Curve Points (P). We also have the sum of scalar-point products, with a vector of scalars $\mathbf{v}$ and a vector of curve points $\mathbf{P}$, written as:

$$\mathbf{vP} = v_1P_1 + v_2P_2 + \cdots + v_nP_n$$

3 Prerequisites:

In this section we will be going over the necessary background knowledge to understand the results of the work in this project.

3.1 Finite Field Arithmetic:

We start with *fields* and the operations defined within them, as the operations we will later define that work on elliptic curves are built on fields.

Definition 3.1 (Field)**.** *A field is a set $\mathbb{F}$, along with the addition and multiplication operations. These two operations must uphold the so called field axioms:*

- *Associativity of addition and multiplication:* $\forall a, b, c \in \mathbb{F} : a + (b + c) = (a + b) + c \wedge a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- *Commutativity of addition and multiplication:* $\forall a, b \in \mathbb{F} : a + b = b + a \wedge a \cdot b = b \cdot a$
- *Additive and multiplicative identity:* $\exists 0, 1 \in \mathbb{F} : a + 0 = a \wedge a \cdot 1 = a$
- *Additive inverses:* $\forall a \in \mathbb{F},\ \exists -a \in \mathbb{F} : a + (-a) = 0$
- *Multiplicative inverses:* $\forall a \neq 0 \in \mathbb{F},\ \exists a^{-1} \in \mathbb{F} : a \cdot a^{-1} = 1$
- *Distributivity over addition:* $\forall a, b, c \in \mathbb{F} : a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

Note that this also means that subtraction and division is defined, due to the existance of the additive and multiplicative inverses respectively:

$$a - b = a + (-b)$$
$$\frac{a}{b} = a \cdot b^{-1}$$

Corrolarily, this leads us to *finite fields*:

Definition 3.2 (Finite Field)**.** *A finite field, is a field that contains a finite number of elements.*

The most common types of finite fields, and those used throughout this report are *Prime Fields:*

Definition 3.3 (Prime Field)**.** *A prime field $\mathbb{F}_p$ is a finite field with elements $[0, p-1]$ where each operation is performed over integers modulo p.*

Something important to note about this definition is that inverses in this kind of field are also positive whole numbers. This will come up again in section 4.2 with regards to the code.

3.2 Elliptic Curve:

To start our explanation of elliptic curves we add a few definitions:

Definition 3.4 (Elliptic Curves)**.** *An elliptic curve, E, is an algebraic curve defined over a prime field, $\mathbb{F}_p$, defined by the formula:*

$$y^2 = x^3 + ax + b$$

Definition 3.5 (EC Additive Identity)**.** *Let $\mathcal{O}$ be the point where $y = \infty$. For every point, P, on E the following property holds: $P + \mathcal{O} = \mathcal{O} + P = P$.*

Definition 3.6 (EC Negation)**.** *Let $P = (x, y)$ be a point on E. The negation of P, called $-P$ is defined as the mirror of P over the x-axis, or $-P = (x, -y)$.*

Definition 3.7 (EC Addition)**.** *Let ℓ be a line intersecting E at three points, P, Q and $-R$. The group operation: Addition defined on two points is defined as:*

$$P + Q = -R$$

Definition 3.8 (EC Doubling)**.** *let ℓ be the tangent to the point P on E, which intersects points P and $-R$. The group operation: point doubling on P is defined as:*

$$2P = P + P = R$$

An important thing to note about point addition. In the case where you add points P and Q where ℓ is a tangent to P then $P + Q = -P$. This is due to the fact that the *only* case in which this can happen is when $Q = -(P + P)$ and thus you get $P + Q = P + (-(P + P)) = -P$. Another similar case is using point doubling on any point $P = (x, 0)$ on E. The tangent of P do not intersect a second point. However note that the negation of P here is $-P = (x, -0) = (x, 0) = P$. Thus point doubling on these points is equivalent to the equation $P + (-P) = \mathcal{O}$.

From these definitions we can extrapolate two more definitions:

Definition 3.9 (EC Subtraction)**.** *Let P and Q be elliptic curve points. The group operation: Subtraction is defined by:*

$$P - Q = P + (-Q)$$

Definition 3.10 (EC Subtraction)**.** *Let m be a scalar and let P be a point on E. The group operation: Scalar multiplication is defined by adding P to itself m times and denoted a $m \cdot P$.*

Due to the absence of scalar division it is therefore impossible to multiply by anything other than integers as we cannot have something like $2.5 \cdot P = \frac{5 \cdot P}{2}$. Multiplying by 0 will naturally yield the identity element $\mathcal{O}$ by definition. Additionally multiplying by a negative integer, $-m$, is defined as $-m \cdot P = m \cdot (-P)$.

The particular curve used for our implementation, as well as the implementation of the Rust bulletproofs, is a special curve from a subset of curves known as Montgomery curves. These curves are defined by the formula: $By^2 = x^3 + Ax^2 + x$. Curves of this form have a birational equivalence with a different set of curves, known as Twisted Edwards curves. Our implementation, as well as the implementation we test against utilize this aspect in their internal representations of points on the curve.

For our implementation of bulletproofs, seeing as we needed to test it against an existing implementation we made use of the same elliptic curve, as well as gained an understanding of the basics of elliptic curve cryptography. The elliptic curve in question is Curve25519, which is widely used in encryption as is the case here. Curve25519, henceforth shortened to 25519, is defined by the following formula:

$$y^2 = x^3 + 48662x^2 + x$$

This curve is defined over the prime field $K = 2^{255} - 19$ and base point defined at $x = 9$ with the positive y value. This results in a sub-group of order $2^{252} + 27742317777372353535 85$, which has a co-factor of 8. This co-factor would make cryptocurrencies that use Curve25519 elliptic curves vulnerable to the so-called 'Double Spending Vulnerability'. However the implementation known as Ristretto circumvents this by eliminating the co-factor. For more detail see 3.5

3.3 Bulletproofs:

The ultimate goal of this project has been to implement bulletproofs in Hacspec, using property-based testing to ensure it is equivalent to the implementation done in Rust, using 25519. A bulletproof is a non-interactive aggregation of rangeproofs using Pedersen commitments. The following section will describe each of these terms in greater detail on a theoretic level.

3.3.1 Pedersen Commitment:

A Pedersen commitment is a commitment scheme defined by a certain property we will go into a little later. For our purposes we redefine the scheme to work with commitments to elliptic curve points rather than a value. First a single point G on the chosen elliptic curve, in this case 25519. This point will be our generator. for 25519 we use the point we previously defined as the base point.

Our adaptation of the Pedersen commitment scheme will involve point addition on the elliptic curve, rather than multiplication. This will have the desired effect of perfect hiding for Pedersen commitments. The points that we will add together to hide our commitment works using some amount of randomness which ensures the hiding property while also allowing for some decent binding. Not perfect, as this is impossible alongside perfect hiding.

Definition 3.11 (Pedersen Commitment)**.** *A Pedersen commitment, to some message, a, with a randomly chosen integer, r, is defined as:*

$$rH + aG$$

where G is a canonical generator for the curve and H is a public key where no one knows q such that $H = qG$.

Here r serves as our perfect hiding, G is our generator for the curve and H is another curve point where no one knows q such that $H = qG$. This ensures perfect hiding as there is a nearly infinite amount of possible combinations of two points that can add to any given point. However while that is true, it does not provide perfect binding by definition of binding and hiding. From here on we will refer to a commitment to a message, a, with random value, r, as $C(a, r)$.

The most important property of Pedersen commitments however is that pedersen commitments have the property of *Additive Homomorphism*:

Definition 3.12 (Additive Homomorphism)**.** *For any two Pedersen commitments $C(a_1, r_1), C(a_2, r_2)$ we have:*

$$C(a_1, r_1) + C(a_2, r_2) = C(a_1 + a_2, r_1 + r_2)$$

This is simple to show by simply applying the definition of Pedersen commitments:

$$\begin{aligned} C(a_1, r_1) + C(a_2, r_2) &= r_1H + a_1G + r_2H + a_2G \\ &= (r_1 + r_2)H + (a_1 + a_2)G \\ &= C(a_1 + a_2, r_1 + r_2) \end{aligned}$$

Which ensures a homomorphism between the sum of commitments and the commitment to the sum. This property will become vital when proving zero knowledgeness later.

Expanding on the *Vector Pederson Commitment*:

Definition 3.13 (Vector Pederson Commitment)**.** *A Vector Pedersen Commitment, to some vector of messages, $\boldsymbol{v}$, with a randomly chosen integer, r, is defined as:*

$$rH + \boldsymbol{vG}$$

where $\boldsymbol{G}$ is a vector of canonical generators for the curve and H is still a private key.

3.3.2 Inner Product Proof:

The inner product proof is a zero knowledge proof where the prover, $\mathcal{P}$, shows knowledge of two vectors $\mathbf{x}$ and $\mathbf{y}$, as well as their inner product, z, to the verifier, $\mathcal{V}$.

In order to do this the *Sigma Protocol*, which is a 3-phase protocol between verifier and prover of the form:

1. $\mathcal{P} \to \mathcal{V}$: Commitment
2. $\mathcal{V} \to \mathcal{P}$: Challenge
3. $\mathcal{P} \to \mathcal{V}$: Response

The prover sends a commitment to the verifier, the verifier responds with a randomly generated challenge, and finally the prover constructs a new commitment from the first commitment and the challenge...

We will now go through each step in order with the inner product proof in mind. The inner product, z, is the inner product of vectors $\mathbf{x}$ and $\mathbf{y}$. We assume that Pedersen commitments to these values are know:

$$
\begin{aligned}
C_z &= r_1 H + zG \\
C_x &= r_2 H + \mathbf{x}\mathbf{G} \\
C_y &= r_3 H + \mathbf{y}\mathbf{G}
\end{aligned}
$$

We define two random nonce vectors $\mathbf{d_x}, \mathbf{d_y}$

Step 1:

$$
\begin{aligned}
C_{d_x} &= r_{d_x} H + \mathbf{d_x}\mathbf{G} \\
C_{d_y} &= r_{d_y} H + \mathbf{d_y}\mathbf{G}
\end{aligned}
$$

$$
\begin{aligned}
C_0 &= r_0 H + (\mathbf{d_x} \cdot \mathbf{d_y})\, G \\
C_1 &= r_1 H + (\mathbf{x} \cdot \mathbf{d_x} + \mathbf{y} \cdot \mathbf{d_x})\, G
\end{aligned}
$$

So the prover sends:

$$\mathcal{P} \to \mathcal{V} : (C_{d_x}, C_{d_y}, C_0, C_1)$$

Step 2:

The verifier sends challenge e to prover

$$\mathcal{V} \to \mathcal{P} : e$$

Step 3:

$$
\begin{aligned}
\mathbf{f_x} &= e\mathbf{x} + \mathbf{d_x} \\
\mathbf{f_y} &= e\mathbf{y} + \mathbf{d_y} \\
r_x &= e r_2 + r_{d_x} \\
r_y &= e r_3 + r_{d_y} \\
r_z &= e r_4 + r_{d_y}
\end{aligned}
$$

$$\mathcal{V} \to \mathcal{P} : (\mathbf{f_x}, \mathbf{f_y}, r_x, r_y, r_z)$$

$$r_x H + \mathbf{f_x} \cdot \mathbf{G} =? eC_x + C_{d_x}$$

$$r_y H + \mathbf{f_y} \cdot \mathbf{G} =? eC_y + C_{d_y}$$

$$r_z H + (\mathbf{f_x} \cdot \mathbf{f_y})G =? e^2 C_z + eC_1 + C_0$$

$$\begin{aligned}
\mathbf{G_L} &= [G_1, \cdots, G_{n/2}] \\
\mathbf{G_R} &= [G_{n/2+1}, \cdots, G_n] \\
\mathbf{H_L} &= [H_1, \cdots, H_{n/2}] \\
\mathbf{H_R} &= [H_{n/2+1}, \cdots, H_n] \\
\mathbf{a_L} &= [a_1, \cdots, a_{n/2}] \\
\mathbf{a_R} &= [a_{n/2+1}, \cdots, a_n] \\
\mathbf{b_L} &= [b_1, \cdots, b_{n/2}] \\
\mathbf{b_R} &= [b_{n/2+1}, \cdots, b_n]
\end{aligned}$$

$$\begin{aligned}
L_a &= \mathbf{a_L G_R} \\
L_b &= \mathbf{b_L H_R} \\
L &= \mathbf{L_a} + \mathbf{L_b}
\end{aligned}$$

$$\begin{aligned}
R_a &= \mathbf{a_R G_L} \\
R_b &= \mathbf{b_R H_L} \\
R &= \mathbf{R_a} + \mathbf{R_b}
\end{aligned}$$

3.3.3 Range Proofs:

3.3.4 Bulletproofs:

3.4 Hacspec:

Hacspec is a subset of the programming language of Rust, designed in a way that makes it easy to compile into theorem solvers such as Coq or F*. The cost for having this property is a reduction in certain conventions available in Rust not being present in Hacspec. This is a double edged sword, both making the language simple, but also making it harder to express certain abstract ideas. The language however is still in development, balancing and adding features to the language.

All code written for this project was written to be Hacspec compliant. Our implementation is a specification meant to be simply understood compared to a more obfuscated, but highly optimized implementation. We will use property based testing with QuickCheck[1] to check our Bulletproofs implementation against the

[1] `https://github.com/BurntSushi/quickcheck`

Dalek-Cryptography Bulletproofs[2]. We have also created a minimal linear algebra library, in hacspec, since it was needed for the bulletproofs implementation. Property based testing using QuickCheck is also used for this library, testing it against the nalgebra[3].

At a later time our implementation could to be compiled to Coq or F* and proof-checked, leading to better guarantees about our implementation. This is however not in the scope of this paper and is left to future work.

3.5 Ristretto:

Ristretto is a specification of elliptic curve cryptography, created for the specific purpose of eliminating unwanted co-factors. This was done in order to circumvent the double-spending vulnerability. This is done using a so-called quotient group. Also appropriately called a Factor Group, a Quotient Group is a type of group, which takes elements from a larger group and, using an equivalence relation, maps elements that are 'similar' to the same element in the quotient group, while preserving most of the group structure. The remaining elements are factored out, leaving a group with the same operations, but fewer elements. What this means for Ristretto is that it takes points on an elliptic curve and eliminates the co-factor by simply mapping them down to their 'equivalent' elements. This is the reason for Ristretto's somewhat unorthodox method of doing operations, as it must ensure that each element is computed to the proper element in the quotient group. This is also done, in part, to allow the user to use points that are NOT in the quotient group without worry as any computation done with these points is automatically mapped to their proper element, thus eliminating the rather computationally expensive operation of checking if a point is on the proper curve.

Additionally the internal representation of the Montgomery curve we wish to eliminate the co-factor for, is its equivalent Twisted Edwards Curve, which is done to more easily factor points from the original curve into the quotient group. Additionally Ristretto's equality method ensures that equivalent representations of points on the curve are equal, even if they are not factored into the quotient group yet. Additionally from its encoding and decoding methods, equivalent points are encoded to the same bitstring, and are therefore decoded into their refactored quotient group element.

The exposed functions that were implemented are the encoding and decoding functions, the equality function to compare points, the addition function over points, point negation, and the derived functions from these, namely point doubling, point subtraction and scalar multiplication. And the final function is the One Way Map function. This function is not entirely necessary for our purposes, but having it allows outside users to more easily generate points from simple byte sequences, additionally makes generating random points for testing much easier.

4 Contributions:

4.1 Linear Algebra Library Specification:

- Vectors vs Matrices
- Describe formulas, with sources (Dot product)
- Generics :(
- Cloning
- Double indexing
- QuickCheck

[2] `https://github.com/dalek-cryptography/bulletproofs`
[3] `https://nalgebra.org/`

It was decided that the specification should of course consist of what we need, but also of some general linear algebra functions that could be used for others who might want to use it. The following functions was decided to be part of the specification:

- Instantiate matrix
- Instantiate zero filled matrix
- (Instantiate one filled matrix)
- Instantiate identity matrix
- Transposition
- Slicing
- (Scalar Multiplication)
- Addition
- Subtraction
- Hadamard product
- Matrix Multiplication

No generalized standard for Linear Algebra Specification.

4.2 Implementing Ristretto:

For our Ristretto implementation on 25519 there exists an IETF-standard specification of exactly this, which was used as a guide for our implementation. [4] This specification was very helpful and after inspecting the code we are testing against closely, it is clear that they used this standard as well.

The most important things to note about this standard, outside the explicit formulas used for its various methods are which functions and internal representations we are allowed to expose to the outside. Namely that the only functions we may expose were encoding of points, decoding of encoded points, the map which is used for creating points, point addition, point negation, point subtraction, scalar multiplication. The last two were allowed purely from being derivable from repeated application of point addition and/or negation. Some things that were not allowed to be exposed under any circumstances, are the internal representations of either points or its field elements or any functions that are not the ones mentioned above, which is as expected for something that is meant to be a thin layer of abstraction beneath an implementation of 25519.

Each internal point representation is composed of four field elements $(X : Y : Z : T)$. Field-elements, as defined by the standard are values modulo p, with p being the prime field for 25519, $2^{255} - 19$. This was achieved using the `public_nat_mod!` macro which defines fields over certain values which just so happen to accept hex values as modulo values, which was very convenient. With this we are now able to create field elements using various functions and properties that are native to the type created by `public_nat_mod!`. Most of the standard integer operators like addition, subtraction and multiplication are implemented for these field elements for example. An important thing to note is that the internal calculations for division is done using integer arithmetic rather than finite field arithmetic, however the few times where division is used directly it is ensured that the result in finite field arithmetic is equal to the integer arithmetic solution.

Additionally the standard, specified a series of constants. These constants were too large to be implemented as integers. Additionally hacspec did not allow for us to simply use the `from_hex()` method. As such, after converting each of these constants into their corresponding hex-values, we built them as byte sequences for which we had an equivalent `from_byte_seq_be` which created the correct field elements we wanted.

While it is impossible for a legal point to be encoded and then have the decoding on its encoding fail, the decoding method has several checks that ensure that the input given is legal. The primary reason for this is that decoding is a method available to the client and as such there is no guarantee that the byte-strings

[4] `https://www.ietf.org/archive/id/draft-irtf-cfrg-ristretto255-00.html`

they want to decode are necessarily legal points that they first encoded properly, but could be artificially constructed byte-strings. We want to avoid this and thus the standard has a list of properties the byte-string input and attempted decoding must fulfill in order to be canonical.

4.3 Implementing Bulletproofs:

Our bulletproofs implementation was implemented on the foundation of our implementations of Linear Algebra and Ristretto. The first and simplest step was implementing Pedersen commitments to single values as well as to vectors. From our implementation of Ristretto we can easily add points to each other and multiply points by a scalar. This is essentially the only thing a Pedersen commitment is. Additionally we needed it to support vector commitments. From our implementation of Linear Algebra we could easily adapt this into the code. This implementation was nearly trivial given the base we built it upon.

The next step in the process towards implementing Bulletproofs is the inner product proof. The implementation of which does not lean itself against our Ristretto implementation as our implementation of Pedersen commitments did, however it does utilize said Pedersen commitments.

4.4 Analysis of coding stuff:

4.5 Summary of coding stuff:

4.6 Conclusions on our work:

5 Future work:

6 Acknowledgements:

7 References