Verifying an elliptic curve cryptographic algorithm using Coq and the Ssreflect extension

Master's thesis
Mathematics

Timmy Weerwag

31 August 2016

Supervisor
dr. F. Wiedijk

Second reader
dr. P. Schwabe

Radboud University

Abstract

An important element of modern cryptography are key exchange methods, which are used by two parties to obtain a shared secret through an insecure channel. Bernstein introduced a new method for key exchange, based on the simultaneously introduced Curve25519 function, which uses arithmetic on a specific Montgomery curve. Montgomery curves are a form of elliptic curves which allow for fast computations of scalar multiples. Using Coq and Ssreflect, we verify the functional correctness of an algorithm computing Curve25519. For this purpose, we extended the elliptic curve library by Bartzia and Strub to support Montgomery curves. The verified algorithm is used, for example, in TweetNaCl, a small, concise cryptographic library.

Acknowledgements

I'd like to thank Robbert Krebbers, Freek Wiedijk and Herman Geuvers for all ideas for possible researches. It was one of those ideas that led to this research.

Thanks to Cyril Cohen, Laurence Rideau, Enrico Tassi and Laurent Théry for organizing the winter school in Sophia-Antipolis on Ssreflect and the Mathematical Components library. Although I could not attend, the uploaded material from this week comprises a good tutorial into this matter.

Another thanks to Laurent Théry for responding so quickly after an inquiry about proving primality of large numbers in Coq.

A big thanks to Peter Schwabe for reading and commenting on this thesis in a short time frame.

Last but certainly not least, I would like to thank Freek Wiedijk, my thesis supervisor. His expertise on formalizations has been helpful for devising strategies to tackle problems which arose during formalizing. Besides that, his never-ending enthusiasm and optimism during the weekly meetings have been a big factor.

Contents

Chapter 1

Introduction

Cryptography is widely used on the internet nowadays. Probably most visible is the increasing usage of “https” by websites. Besides secure HTTP traffic, an increasing number of DNS records is signed, people working at home connect to their company’s network with secure VPN connections, etc.

First of all, it is important that the cryptographic primitives and protocols are safe theoretically. For example, the MD5 hashing function, developed over two decades ago, aims for collision resistance, which means it is very hard to find two inputs which generate the same hash. However, feasible attacks have been found. This enabled the construction of a CA certificate which appeared to be signed by a trusted certificate authority [14].

A well-studied unbroken cryptographic primitive is not enough though. Uncareful implementations can leak secret data, for example, by spending a different amount of time on a computation depending on a secret datum. These so called timing attacks are also feasible in the real world, see for example [10].

There are some other related attacks. Implementing cryptographic software thus requires certain programming techniques to prevent these kinds of attacks. Unfortunately, this could also introduce extra bugs and faulty behaviour.

The classical approach to prevent bugs and faulty behaviour is to rely on an automated test suite, and careful reviewing of every piece of committed code by other team members. However, one cannot garantuee the absence of bugs this way. Recently, with the Heartbleed bug, we have seen that even clear, simple mistakes can slip through this system [6].

It should be clear that a better method for verifying software is needed. This is where formal verification jumps in, using computers to provide mathematical proofs that a program cannot exhibit undefined behaviour, that all bounds of arrays are checked, that a program satisfies a specification, etc.

In this thesis we will try to formalize and verify a part of the algorithm used by TweetNaCl for key exchange. TweetNaCl is a small high-security cryptographic library written in C. The aim of TweetNaCl was to produce small, concise, and readable code. Although TweetNacl supports a wide range of cryptographic applications, it is distributed as 100 messages on a well-known social network supporting messages of at most 140 characters. A result of the conciseness of the code is the high auditability of TweetNaCl.

We will use the proof assistant Coq with the Ssreflect extension to verify that TweetNaCl's algorithm for Curve25519, a method for key exchange, matches the mathematical specification of Curve25519. The devolopment in Coq can be found at

`http://www.timmyweerwag.nl/research/`

1.1 Related work

The paper that introduced TweetNaCl [5] already mentioned the verification of two properties of TweetNaCl's code: all array accesses for inputs of a *certain* (not arbitrary) length are within bounds, and all array elements are initialized before being read. They used C++ overloading to achieve this.

In [8], the functional correctness of two highly optimized implementations of Curve25519 was verified. They annotated `qhasm` (a high-level assembler) files with conditions, and used an SMT solver (and a bit of Coq in hard cases) to verify these conditions. The translation from `qhasm` to normal assembler files was assumed correct. Interestingly, they were able to detect a known bug in an old version of one of these implementations, which only occurs with low probability, and probably will not be found by extensive testing.

In [1], Appel used the Verified Software Toolchain (VST) to verify the functional correctness of the SHA-256 implementation of OpenSSL. The VST contains the Verifiable C logic, which is proved sound with respect to the operational semantics of CompCert. CompCert is an optimizing C compiler where the translation upto assembler code is proved correct. By composition, this yields correct assembler code of this implementation of SHA-256. However, it also assures us that compilation with other, major C compilers probably yields correct code.

Lennart Beringer also uses the Verified Software Toolchain to prove the functional correctness of TweetNaCl's Salsa20 code. At the time of writing, the results were not published yet. Salsa20 consists of a core function and an accompanying stream cipher based on the core function. TweetNaCl uses Salsa20 for its symmetric-key encryption.

Chapter 2

Key exchange

Historically, two parties who wished to exchange a message in secret, had to agree in advance on a key to use. For example, Ceasar's cipher works by shifting the letters of the message by a fixed amount along the alphabet. Encrypting a message with a shift of 3 involves changing A's into D's, B's into E's, C's into F's, etc. In this case, the key consists of the shift amount.

Of course, Ceasar's cipher has an extremely limited amount of possible keys, and is extremely easy to crack. An example with a much bigger key space is the Enigma. These machines used by the German army in WWII had room for three rotors which could be chosen from a set of five. Hence, there are 60 different ways of inserting the rotors. Each rotor could be set to one of its 26 positions, which leads to 17576 different initial positions of the rotors. On each key press the right-most rotor always turned one position. However, the point on which the other two rotors would turn, could be altered with a ring around the rotors. This leads to 676 different ring settings.

However, most contributing to the number of configurations was the stecker board on the front of the machine. On the board were 26 sockets corresponding to each of the 26 letters. Ten cables were used to swap ten pairs of letters on the board. With a little bit of combinatorics, we see that this yields

$$\frac{26!}{(26-20)! \cdot 10! \cdot 2^{10}} = 150738274937250$$

configurations of the stecker board. All in all, the Enigma machine as used in WWII by the German army had approximately 1.07×10^{23} (over 100 sextillion) different initial configurations.

However, everyone in the German army used the same key at the same moment. Since the keys were distributed on printed form, it would be impracticable to use different keys for any pair of operators who wished to communicate. In addition, the sheets contained keys for the whole month. So despite of the tremendous amount of keys, if one could capture just one of these sheets, all communication of the whole German army for that period could be deciphered.

Today, still a lot of encryption schemes work with a common key between the two parties. However, instead of just being used by centralized institutions like armies and intelligence agencies, cryptography is nowadays also widely used by the casual internet user to pay its bills, buy stuff online, secretly google his boss, etc. It should be clear that it is a bit cumbersome that if the casual internet user wants to secretly google his boss, he has to drive to the Googleplex[1] merely to agree on a key to use. For that reason, so called key exchange methods were devised. Instead of relying on a secure channel (meeting someone in private, a government courier, etc.), key exchange methods rely on mathematics to securely agree on a common key over an insecure channel (e.g. the internet).

An immensely popular method for key exchange is named after Diffie and Hellman. In this system, both parties (let's call them Alice and Bob) have a key pair consisting of a private and a public key. It should be noted that not every set of a private and public key forms a valid key pair (in fact, the public key is generated from the private key). The private key is shared with nobody. On the other hand, the public key may be given to anybody without compromising security. The heart of Diffie-Hellman key exchange is an algorithm which computes a new key from somebody's private and somebody else's public key. The important property of this algorithm is that if Alice feeds the algorithm her private key and Bob's public key, she obtains the same output as when Bob feeds the algorithm his private key and Alice's public key. However, no other (easy findable) combination of a private and public key yields the same output, so an attacker needs to have either Alice's or Bob's private key (assuming the public keys are readily available), or needs to solve the mathematical problem on which the algorithm relies.

The classical Diffie-Hellman key exchange is as follows. Alice and Bob first agree on a (large) prime p, and a primitive root modulo p, g. Both Alice and Bob randomly pick a private key $a \in \{1, \dots, p-1\}$, respectively $b \in \{1, \dots, p-1\}$. They compute their public keys $A = g^a \bmod p$, $B = g^b \bmod p$ and send it to each other. Alice computes their shared secret as $S_A = B^a \bmod p$, and Bob computes $S_B = A^b \bmod p$. It is clear that $S_A = S_B$.

Note that modular exponentiation can be done fast by repeated squaring. On the other hand, finding a such that $g^a \equiv A \pmod{p}$ can be extremely difficult for large p. This problem is known as the discrete logarithm problem and prevents the attacker from obtaining the secret. Unfortunately, although efficient algorithms for discrete logarithms haven't been found, still no-one was able to prove these algorithms don't exist. Hence, the secureness of this method is only conjectured, not proved.

Note that the preceding method uses the finite group $(\mathbf{Z}/p\mathbf{Z})^\times$, the multiplicative group of integers modulo p. This is not the only type of group for which the discrete logarithm problem is hard. A notable other type of group, and our subject of interest, are the elliptic curves over finite fields. We can adapt the method by replacing exponentiation modulo p with scalar multiplication on a cyclic subgroup of the

[1]Corporate headquarters of Google, located at 1600 Amphitheatre Parkway in Mountain View, Santa Clara County, California, United States.

elliptic curve group. (Note that elliptic curve groups are usually used additively, i.e the group operation is written as addition.)

This leads to the elliptic curve version of Diffie-Hellman. Alice and Bob first agree on an elliptic curve E, a positive integer n, and a base point G on E of order n. Both Alice and Bob randomly pick a private key $a \in \{0, \dots, n-1\}$, respectively $b \in \{0, \dots, n-1\}$. They compute their public keys $A = aG$ and $B = bG$ and send it to each other. Alice computes their shared secret as $S_A = aB$, and Bob computes $S_B = bA$. Again, clearly, $S_A = S_B$.

One should take care when choosing the elliptic curve E and base point G. For example, it is desirable that the order of G is a large prime. There have been many studies on this subject, and since we are not trying to invent a new key exchange method, we won't delve into this subject.

Curve25519 is a function which can be used for key exchange in the elliptic curve Diffie-Hellman style. It uses scalar multiplication on the elliptic curve $E : y^2 = x^3 + 486662x^2 + x$ over the finite field $\mathbf{F}_{2^{255}-19}$. In Section 2.2 we will see that for these types of curves, it is sufficient to only store the x-coordinates of points on the elliptic curve. A point with the x-coordinate equalling 9 is used as base point. The cyclic subgroup generated by this point has order $2^{252} + 27742317777372353535851937790883648493$, which is a prime [3, §2].

More concrete, Curve25519 is defined as a function $\{\text{Curve25519 private keys}\} \times \{\text{Curve25519 public keys}\} \to \{\text{Curve25519 public keys}\}$. The set of Curve25519 public keys is just $\{0, 1, \dots, 255\}^{32}$, the set of all values of 32 bytes. The set of Curve25519 private keys is $\{0, 8, 16, \dots, 248\} \times \{0, 1, \dots, 255\}^{30} \times \{64, 65, \dots, 127\}$. These 32-tuples of bytes represent numbers in little-endian form in the finite field $\mathbf{F}_{2^{255}-19}$. Given a private key n and a public key q, Curve25519(n, q) is the public key s such that for all points Q on E with the x-coordinate equalling q mod $2^{255} - 19$, the x-coordinate of nQ equals s.

2.1 Weierstraß curves

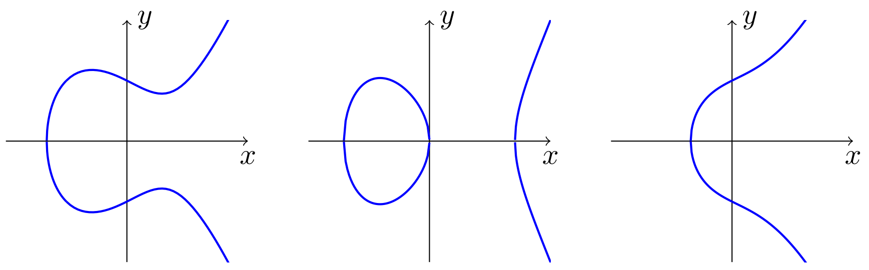
In this section, we consider elliptic curves over a field K. We assume that the characteristic of K is neither 2 nor 3.

We begin with a definition of elliptic curves in a general form, the Weierstraß form. A well-known result [13, §III.3] is that every non-singular algebraic curve of genus one can be "written" in the Weierstraß form, i.e. for each non-singular algebraic curve of genus one there exists a Weierstraß curve whose curve group is isomorphic to the curve group of the other algebraic curve. A slightly more general form exists which allows the characteristic of K to be 2 or 3. However, for us the following form is sufficient.

Definition 2.1. Let $a, b \in K$ such that $-16(4a^3 + 27b^2) \neq 0$. The *elliptic curve* $E_{a,b}$ is defined by the set of all points $(x, y) \in K^2$ satisfying the equation

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

Figure 2.1: Plots of three elliptic curves of the Weierstraß form over $\mathbf{R}$. From left to right: $E_{-1,1}$, $E_{-2,0}$, $E_{1,1}$.



along with an additional formal point $\mathcal{O}$, called *the point at infinity*. We call this form of elliptic curves *the Weierstraß form*.

The quantity $-16(4a^3 + 27b^2)$ is the *discriminant* of the elliptic curve. The curve is smooth if and only if the discrimant is nonzero, which explains the requirement in Definition 2.1. Here, the factor -16 is irrelevant. However, it is needed when the discriminant is used in some other situations. We leave it in for familiarity.

A very useful property of elliptic curves is that we can define a group on it. Since the operation we will define will be commutative, we'll call this operation addition, and use the usual symbol for it. Addition on elliptic curves rests on the fact that any line through two points on an elliptic curve will generally intersect the curve in a unique third point. Some corner cases remain, but we can cope with them as well. See Figure 2.2 for a visualization.

Definition 2.2. Let (x, y) be a point on an elliptic curve E. The *opposite of a point on E* is defined as

$$-\mathcal{O} = \mathcal{O} \qquad \text{and} \qquad -(x, y) = (x, -y).$$

Definition 2.3. Let P_1 and P_2 be points on an elliptic curve E.

i) If $P_1 \neq P_2$ and the line through P_1 and P_2 intersects the curve in a distinct third point P_3, define $P_1 + P_2 = -P_3$.

ii) If $P_1 \neq P_2$ and the line through P_1 and P_2 is tangential to the curve at P_1, define $P_1 + P_2 = -P_1$. Symmetrically, define $P_1 + P_2 = -P_2$ if the line through P_1 and P_2 is tangential to the curve at P_2.

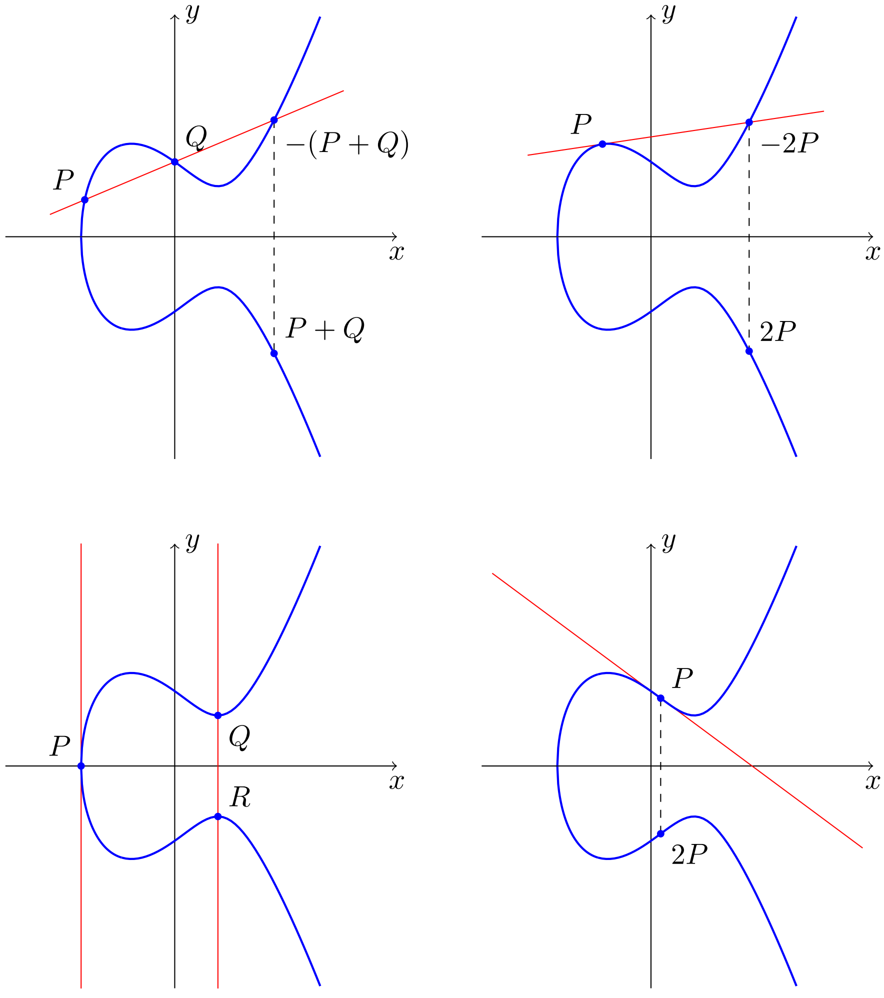
iii) If $P_1 \neq P_2$, but the two other cases above don't apply, define $P_1 + P_2 = \mathcal{O}$.

iv) If $P_1 = P_2$ and the tangent to the curve at P_1 intersects the curve in a distinct second point P_3, define $P_1 + P_2 = -P_3$.

v) If $P_1 = P_2$ and P_1 is an inflection point of the curve, define $P_1 + P_2 = -P_1$.

vi) If $P_1 = P_2$, but the two other cases above don't apply, define $P_1 + P_2 = \mathcal{O}$.

Figure 2.2: Addition on the Weierstraß curve $E_{-2,2}$ over $\mathbf{R}$. Note that in the lower left figure we have $2P = \mathcal{O}$ and $Q+R = \mathcal{O}$. In the lower right figure we have $2P = -P$.



vii) If $P_1 = \mathcal{O}$, define $P_1 + P_2 = P_2$.

viii) if $P_2 = \mathcal{O}$, define $P_1 + P_2 = P_1$.

Note, in case (iii) the line through P_1 and P_2 will be a vertical line, and thus, “intersects the curve at infinity”. Therefore, we define $P_1 + P_2 = \mathcal{O}$ in that case. Similarly, in case (vi) the tangent at P_1 will be a vertical line.

We do have to check that these operations uphold the group laws. However, this geometric definition isn't very handy for computations. We can transform this definition into algebraic formulas.

Lemma 2.4. *Let $E_{a,b}$ be an elliptic curve. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be* *points on $E_{a,b}$, such that $P_1 \neq -P_2$. If $x_1 \neq x_2$, define*

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

On the other hand, if $x_1 = x_2$ (and thus, $y_1 = y_2$, since not $y_1 = -y_2$), define

$$\lambda = \frac{3x_1^2 + a}{2y_1}. \tag{2.1}$$

Now define $P_3 = (x_3, y_3)$ as

$$x_3 = -x_1 - x_2 + \lambda^2 \qquad \textit{and} \qquad y_3 = -y_1 - \lambda(x_3 - x_1). \tag{2.2}$$

Then, $P_1 + P_2 = P_3$.

Remark 2.5. If we say "Let $E_{a,b}$ be an elliptic curve," we actually mean, "Let $a, b \in K$ such that the discriminant of $E_{a,b}$ is nonzero, i.e. $-16(4a^3 + 26b^2) \neq 0$."

Proof of Lemma 2.4. Observe that λ in Lemma 2.4 is the slope of line through P_1 and P_2, if $P_1 \neq P_2$. On the other hand, if $P_1 = P_2$, we take the tangent line. The slope of that line can be determined by differentiating the elliptic curve equation. One then obtains $2y\frac{dy}{dx} = 3x^2 + a$, which explains λ in that case.

So we obtain the line $y = \lambda x + \mu$ for another constant μ. (We could compute that constant with $\mu = y_1 - \lambda x_1$, but that isn't necessary.) Points on the intersection of that line with E satisfy the equation

$$(\lambda x + \mu)^2 = x^3 + ax + b.$$

After writing out the terms and rewriting a bit we get the cubic equation

$$x^3 - \lambda^2 x^2 + (a - 2\lambda\mu)x + (b - \mu^2) = 0. \tag{2.3}$$

We already know that x_1 and x_2 are solutions. Let x_3 be another solution, so we can factorize the equation as

$$(x - x_1)(x - x_2)(x - x_3) = 0.$$

The polynomial on the left-hand side contains the term $(-x_1 - x_2 - x_3)x^2$. This coefficient should be equal to the corresponding coefficient in (2.3), i.e. $-x_1 - x_2 - x_3 = -\lambda^2$. (Note that this is an instance of Vieta's formulas.) Hence,

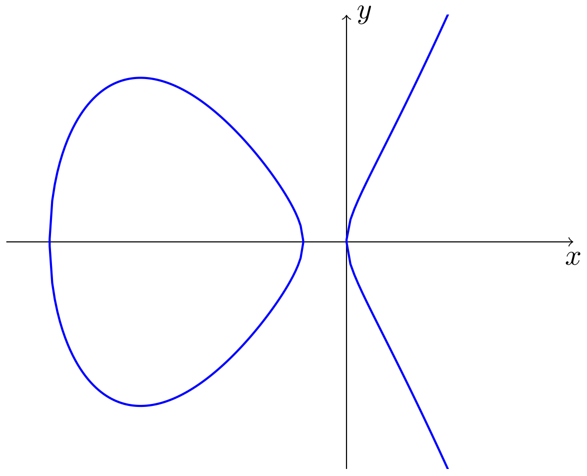
$$x_3 = \lambda^2 - x_1 - x_2.$$

Now we can obtain y_3 pretty quickly by looking at x_3 on the line $y = \lambda x + \mu$. Note that we want take the opposite, so this yields $y_3 = -\lambda x_3 - \mu$. Filling in $\mu = y_1 - \lambda x_1$ then gives

$$y_3 = -\lambda(x_3 - x_1) - y_1.$$

□

Figure 2.3: A plot of a typical Montgomery curve ($M_{1,3}$) over $\mathbf{R}$.



Commutativity and most of the group laws are easily verified. However, proving associativity directly with these equations is extremely tedious. For an indirect elementary proof, see [7, sections 1-6]. We'll state the result without a proof.

Theorem 2.6. *An elliptic curve $E_{a,b}$ together with the operations $-$ and $+$ forms a commutative group. The triple $(E_{a,b}, -, +)$ will be called the elliptic curve group of $E_{a,b}$.*

2.2 Montgomery curves

By now, it should be clear that computations on elliptic curves are hard. Speedups can be obtained by using homogeneous coordinates and other forms than the Weierstraß form. We'll consider the Montgomery form, named so since they were introduced by Montgomery [12].

Definition 2.7. Let $a \in K \setminus \{-2, 2\}$, and $b \in K \setminus \{0\}$. The *Montgomery curve* $M_{a,b}$ is the set of all points $(x, y) \in K^2$ satisfying the equation

$$by^2 = x^3 + ax^2 + x, \tag{2.4}$$

along with an additional formal point, $\mathcal{O}$, "at infinity".

We can define addition on Montgomery curves the same way as we did with the Weierstraß form. However, equations (2.1) and (2.2) become slightly different.

Lemma 2.8. *Let $M_{a,b}$ be a Montgomery curve. The same formulas for addition with a curve in the Weierstraß form (Lemma 2.4) can be used for addition on $M_{a,b}$, except that we'll use*

$$\lambda = \frac{3x_1^2 + 2ax_1 + 1}{2by_1}$$

when $P_1 = P_2$, *and*

$$x_3 = \lambda^2 b - a - x_1 - x_2.$$

Proof. With implicit differentiation of equation (2.4) we find

$$2by\frac{dy}{dx} = 3x^2 + 2ax + 1.$$

Equation (2.3) becomes

$$x^3 + (a - \lambda^2 b)x^2 + (1 - 2\lambda\mu b)x - \mu^2 b = 0.$$

□

Remark 2.9. Again, if we say "Let $M_{a,b}$ be a Montgomery curve," we actually mean, "Let $a, b \in K$ such that $M_{a,b}$ is a Montgomery curve, i.e. $a^2 \neq 4$ and $b \neq 0$."

Of course, we are also interested in the correspondence between the Weierstraß and Montgomery forms. It happens that every Montgomery curve can be transformed into a Weierstraß curve, but not vice versa. This transformation actually is a linear coordinate transformation and acts as an isomorphism on the elliptic curve groups.

Lemma 2.10. *Let* $M_{a,b}$ *be a Montgomery curve. Define*

$$a' = \frac{3 - a^2}{3b^2} \qquad \textit{and} \qquad b' = \frac{2a^3 - 9a}{27b^3}.$$

Then, $E_{a',b'}$ *is an elliptic curve, and the mapping* $\phi : M_{a,b} \to E_{a',b'}$ *defined as*

$$\begin{aligned} \phi(\mathcal{O}_M) &= \mathcal{O}_E \\ \phi((x, y)) &= \left(\frac{x}{b} + \frac{a}{3b}, \frac{y}{b}\right) \end{aligned}$$

is an isomorphism between groups.

Proof. First, we'll want to check that $E_{a',b'}$ has a nonzero discriminant. By expanding a' and b' we get

$$-16(4a'^3 + 27b'^2) = -16\left(4\left(\frac{3 - a^2}{3b^2}\right)^3 + 27\left(\frac{2a^3 - 9a}{27b^3}\right)^2\right) = -16\left(\frac{-a^2 + 4}{b^6}\right).$$

Since $M_{a,b}$ is a Montgomery curve, we know $a \in K \setminus \{-2, 2\}$ and $b \in K \setminus \{0\}$, and hence the discriminant of $E_{a',b'}$ is nonzero.

Now we have to prove that ϕ is additive, i.e. for all P and Q on $M_{a,b}$, $\phi(P + Q) = \phi(P) + \phi(Q)$. We could proceed with an elegant argument as in [13, §III.4], which can be applied to other forms of elliptic curves and maps between them as well. However, since we only deal with Weierstraß and Montgomery curves, we choose a simpler, less elegant path.

Let P and Q be points on $M_{a,b}$. By writing out the definitions of ϕ and of addition on $M_{a,b}$ and $E_{a',b'}$, one needs merely elementary algebra to verify that $\phi(P + Q) = \phi(P) + \phi(Q)$. This argument is fully checked in the formalization and therefore omitted here. □

As a remark, there is also a simple geometric argument that the mapping from Lemma 2.10 is a group isomorphism. The geometric definition of addition on a Montgomery curve transfers very nicely to a geometric definition of addition on a Weierstraß curve, since the mapping is just an affine transformation.

We don't give an explicit counterexample of a Weierstraß curve which doesn't have a corresponding Montgomery curve. However, we do give a simple counting example. We consider elliptic curves over the finite field $\mathbf{F}_5$. Observe that the only valid choices for a and b for a Montgomery curve $M_{a,b}$ are 0, 1 and 4 for a, and 1, 2, 3 and 4 for b. This leads to 12 different Montgomery curves over $\mathbf{F}_5$. On the other hand, computing the discriminant $-16(4a^3 + 27b^2)$ for all $a, b \in \mathbf{F}_5$ yields 20 different pairs (a, b) such that $E_{a,b}$ is non-singular. Hence, there are some Weierstraß curves over $\mathbf{F}_5$ without a corresponding Montgomery curve over $\mathbf{F}_5$.

Our goal is to define an efficient algorithm for computing scalar multiples on an elliptic curve. For these computations we only use the x-coordinates of points. This relies on the fact that the x-coordinate of a point already gives us a lot of information. In fact, there are at most two points with the same x-coordinate.

Lemma 2.11. *Let $M_{a,b}$ be a Montgomery curve, and let $x \in K$. Suppose there is an $y \in K$ such that (x, y) is a point on $M_{a,b}$, then for all points (x', y') on $M_{a,b}$ such that $x' = x$, either $y' = y$ or $y' = -y$.*

Proof. Observe that for any $y' \in K$, the point (x, y') lies on $M_{a,b}$ if and only if $by'^2 = x^3 + ax^2 + x$. It is clear that there are at most two solutions for y' for a given x. Moreover, if a solution y' exists, then y' and $-y'$ are the only solutions. □

The Montgomery form also allows for some elegant formulas for addition using only the x-coordinates of the points involved. In general, we don't have $P+Q = P-Q$ for points P and Q on a Montgomery curve $M_{a,b}$. Hence, if we only know the x-coordinates of P and Q, we cannot compute the x-coordinate of $P + Q$. We need an extra bit of information than just the x-coordinates of the two summands.

It happens it suffices to take the x-coordinate of $P - Q$ into account as well. It also happens that this is not a disastrous restriction. Later, we use these x-coordinate addition formulas in a scalar multiplication algorithm. There, the summands are of the form $P + Q$ and P. The x-coordinates of $P + Q$ and P are a result from earlier computations, and the x-coordinate of $(P + Q) - P = Q$ is an input to the algorithm.

Lemma 2.12. *Let $M_{a,b}$ be a Montgomery curve. Let $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, $P_3 = P_1 + P_2 = (x_3, y_3)$ and $P_4 = P_1 - P_2 = (x_4, y_4)$ be finite points on M. If $P_1 \neq P_2$, then*

$$x_3x_4(x_1 - x_2)^2 = (x_1x_2 - 1)^2 \tag{2.5}$$

holds. On the other hand, if $P_1 = P_2$, then

$$4x_1x_3(x_1^2 + ax_1 + 1) = (x_1^2 - 1)^2 \tag{2.6}$$

holds.

Proof. We'll first handle the case $P_1 \neq P_2$. Note that $x_1 \neq x_2$, since if $P_1 = -P_2$, then P_3 is not finite. We assume $x_1 x_2 \neq 0$. From Lemma 2.8 we have

$$x_3 = b\left(\frac{y_1 - y_2}{x_1 - x_2}\right)^2 - a - x_1 - x_2.$$

Multiplying both sides with $(x_1 - x_2)^2$ yields

$$x_3(x_1 - x_2)^2 = b(y_1 - y_2)^2 - (a + x_1 + x_2)(x_1 - x_2)^2.$$

Writing out all terms on the right-hand side gives us

$$\begin{aligned} x_3(x_1 - x_2)^2 &= by_1^2 - x_1^3 - ax_1^2 \\ &+ by_2^2 - x_2^3 - ax_2^2 \\ &- 2by_1y_2 + 2ax_1x_2 + x_1^2x_2 + x_1x_2^2. \end{aligned}$$

Remember that $by^2 = x^3 + ax^2 + x$ for any point (x, y) on the Montgomery curve $M_{a,b}$. Using this fact we get

$$x_3(x_1 - x_2)^2 = x_1 + x_2 - 2by_1y_2 + x_1x_2(2a + x_1 + x_2).$$

We can rewrite this right-hand side again so that we can use the defining equation of the curve again:

$$\begin{aligned} x_3(x_1 - x_2)^2 &= \frac{x_1^2(x_2^3 + ax_2^2 + x_2) - 2bx_1x_2y_1y_2 + x_2^2(x_1^3 + ax_1^2 + x_1)}{x_1x_2} \\ &= \frac{bx_1^2y_2^2 - 2bx_1x_2y_1y_2 + bx_2^2y_1^2}{x_1x_2} \\ &= \frac{b(x_1y_2 - x_2y_1)^2}{x_1x_2}. \end{aligned}$$

Similarly, we have

$$x_4 = b\left(\frac{y_1 + y_2}{x_1 - x_2}\right)^2 - a - x_1 - x_2.$$

The only difference with the previous computation is the term $-2by_1y_2$, which in this case becomes positive, i.e. $2by_1y_2$. It is not hard to see we get

$$x_4(x_1 - x_2)^2 = \frac{b(x_1y_2 + x_2y_1)^2}{x_1x_2}.$$

Multiplying this with our previous result yields

$$\begin{aligned}
x_3x_4(x_1-x_2)^4 &= \frac{b^2(x_1y_2-x_2y_1)^2(x_1y_2+x_2y_1)^2}{x_1^2x_2^2} \\
&= \frac{b^2(x_1^2y_2^2-x_2^2y_1^2)^2}{x_1^2x_2^2} \\
&= \frac{(x_1^2(x_2^3+ax_2^2+x_2)-x_2^2(x_1^3+ax_1^2+x_1))^2}{x_1^2x_2^2} \\
&= (x_1x_2^2+x_1-x_1^2x_2-x_2)^2 \\
&= ((x_1x_2-1)(x_2-x_1))^2 \\
&= (x_1x_2-1)^2(x_1-x_2)^2.
\end{aligned}$$

Dividing both sides by $(x_1-x_2)^2$ gives the desired result.

This compution only works under the assumption $x_1x_2 \neq 0$, so we need to check that the formula is also correct when $x_1x_2 = 0$. So, suppose $x_1x_2 = 0$. Without loss of generality we assume $x_1 = 0$. Remember that we begun the proof with establishing the fact that $x_1 \neq x_2$. Hence, it follows that $x_2 \neq 0$. From Lemma 2.8 we have

$$x_3 = x_4 = b\left(\frac{y_2}{x_2}\right)^2 - a - x_2.$$

Substituting this in the left-hand side of our desired equation:

$$\begin{aligned}
x_3x_4(x_1-x_2)^2 &= \left(b\left(\frac{y_2}{x_2}\right)^2 - a - x_2\right)^2 x_2^2 \\
&= (x_2^2+ax_2+1-ax_2-x_2^2)^2 \\
&= 1 = (x_1x_2-1)^2.
\end{aligned}$$

The formula for when $P_1 = P_2$ can be obtained slightly easier. Again, we'll start Lemma 2.8:

$$x_3 = b\left(\frac{3x_1^2+2ax_1+1}{2by_1}\right)^2 - a - 2x_1.$$

Multiplying both sides with $4by_1^2$:

$$4x_3by_1^2 = (3x_1^2+2ax_1+1)^2 - 4by_1^2(a+2x_1).$$

Substituting $by_1^2 = x_1^3 + ax_1^2 + x_1$ in both sides:

$$4x_3(x_1^3+ax_1^2+x_1) = (3x_1^2+2ax_1+1)^2 - 4(x_1^3+ax_1^2+x_1)(a+2x_1).$$

After writing out all terms on the right-hand side, we get

$$\begin{aligned}
4x_3(x_1^3+ax_1^2+x_1) &= x_1^4 - 2x_1^2 + 1 \\
&= (x_1^2-1)^2.
\end{aligned}$$

□

From equation (2.5) we could for example obtain the formula

$$x_3 = \frac{(x_1x_2 - 1)^2}{x_4(x_1 - x_2)^2}.$$

However, we'll perform these additions a couple of hundred times in the scalar multiplication algorithms. Then we would have to perform division in a finite field at every step, which is quite expensive. We can postpone this division by remembering the denominator as long as possible, or, in other words, by using homogeneous coordinates.

Homogeneous coordinates are a coordinate system devised for projective geometry. We will only consider the projective plane, the projective brother of the Euclidean plane. Points on the projective plane are represented with a triple (X, Y, Z). Any triple except $(0, 0, 0)$ defines a point on the projective plane, although a scalar multiple of a point defines the same point, i.e. (X, Y, Z) and $(\alpha X, \alpha Y, \alpha Z)$ for any $\alpha \neq 0$ define the same point.

We think of the point (X, Y, Z) on the projective plane as the point $(X/Z, Y/Z)$ on the Euclidean plane. This also explains why we identify scalar multiples with each other. Suppose we have points (X, Y, Z) and $(\alpha X, \alpha Y, \alpha Z)$, then $(X/Z, Y/Z) = (\alpha X/\alpha Z, \alpha Y/\alpha Z)$. Of course, this only works if $Z \neq 0$. The points with $Z = 0$ can be thought of as points at infinity.

We can also write the elliptic curve equations using homogeneous coordinates. We replace x with X/Z and y with Y/Z. For example, the equation for a Montgomery curve $M_{a,b}$ becomes

$$b\left(\frac{Y}{Z}\right)^2 = \left(\frac{X}{Z}\right)^3 + a\left(\frac{X}{Z}\right)^2 + \frac{X}{Z}.$$

Multiplying both sides by Z^3 yields the more pleasant equation

$$bY^2Z = X^3 + aX^2Z + XZ^2. \tag{2.7}$$

We would like to use homogeneous coordinates to also represent the formal point at infinity, $\mathcal{O}$, of the curve $M_{a,b}$. However, there are many different points at infinity on the projective plane, and only one point at infinity on our curve. Luckily, if we set $Z = 0$, we see from (2.7) that $X = 0$. Hence, only the "infinite points" $(0, Y, 0)$ with $Y \neq 0$ lie on $M_{a,b}$. Since we identify scalar multiples with each other, these actually represent just one point at infinity.

We will give homogeneous versions of Lemma 2.12. Beside the aforementioned performance gains of homogeneous coordinates, these formulas will also play nicely with $\mathcal{O}$.

It happens that we need to impose an extra condition on the parameter a of a Montgomery curve $M_{a,b}$ though. We will restrict ourselves to Montgomery curves $M_{a,b}$, where $a^2 - 4$ is not a square in the field the curve is defined over. This condition is enough to ensure us that $(0, 0)$ is the only point with a y-coordinate of 0.

Lemma 2.13. *Let $M_{a,b}$ a Montgomery curve such that $a^2 - 4$ is not a square, and let $x \in K$. If $x \neq 0$, then $x^2 + ax + 1 \neq 0$.*

Proof. Suppose $x \neq 0$ and $x^2 + ax + 1 = 0$. We rewrite a in terms of x:

$$a = \frac{-x^2 - 1}{x} = -x - \frac{1}{x}.$$

Then we look at $a^2 - 4$:

$$a^2 - 4 = \left(-x - \frac{1}{x}\right)^2 - 4 = \left(x - \frac{1}{x}\right)^2.$$

Hence, $a^2 - 4$ is a square. Contradiction. □

Lemma 2.14. *Define the function $\mathcal{X} : M_{a,b} \to K \cup \{\infty\}$ as $\mathcal{X}(\mathcal{O}) = \infty$ and $\mathcal{X}((x, y)) = x$. Let $M_{a,b}$ be a Montgomery curve such that $a^2 - 4$ is not a square, and let $X_1, Z_1, X_2, Z_2, X_4, Z_4 \in K$ such that $(X_1, Z_1) \neq (0, 0)$, $(X_2, Z_2) \neq (0, 0)$, $X_4 \neq 0$, and $Z_4 \neq 0$. Define*

$$X_3 = Z_4((X_1 - Z_1)(X_2 + Z_2) + (X_1 + Z_1)(X_2 - Z_2))^2 \quad \textit{and}$$
$$Z_3 = X_4((X_1 - Z_1)(X_2 + Z_2) - (X_1 + Z_1)(X_2 - Z_2))^2,$$

then for any points P_1 and P_2 on $M_{a,b}$ such that $X_1/Z_1 = \mathcal{X}(P_1)$, $X_2/Z_2 = \mathcal{X}(P_2)$, and $X_4/Z_4 = \mathcal{X}(P_1 - P_2)$, we have $X_3/Z_3 = \mathcal{X}(P_1 + P_2)$.

Remark 2.15. For any $x \in K \setminus \{0\}$, $\frac{x}{0}$ should be understood as ∞.

Observe that computing these formulas requires few operations on the field elements. First, we avoid the expensive divisions entirely. We can also reuse $(X_1 - Z_1)(X_2 + Z_2)$ and $(X_1 + Z_1)(X_2 - Z_2)$ in the computations of X_3 and Z_3. All in all, this leads to a mere 3 additions, 3 subtractions, 4 multiplications, and 2 squarings in K to compute both X_3 and Z_3.

Proof of Lemma 2.14. Let P_1 and P_2 points on $M_{a,b}$ such that $X_1/Z_1 = \mathcal{X}(P_1)$, $X_2/Z_2 = \mathcal{X}(P_2)$, and $X_4/Z_4 = \mathcal{X}(P_1 - P_2)$.

Suppose $P_1 = P_2$, then $P_1 - P_2 = \mathcal{O}$, and hence, $Z_4 = 0$. Contradiction.

Suppose $P_1 = \mathcal{O}$. We then have $Z_1 = 0$, and $X_1 \neq 0$. Also, $\mathcal{X}(P_1 - P_2) = \mathcal{X}(-P_2) = \mathcal{X}(P_2)$, so, $X_4/Z_4 = X_2/Z_2$. Observe that since $X_4/Z_4 \neq \infty$ and $X_4/Z_4 \neq 0$, we have $Z_2 \neq 0$ and $X_2 \neq 0$ too. We compute

$$\begin{aligned} Z_3 &= X_4(X_1(X_2 + Z_2) - X_1(X_2 - Z_2))^2 \\ &= X_4(X_1X_2 + X_1Z_2 - X_1X_2 + X_1Z_2)^2 \\ &= X_4(2X_1Z_2)^2. \end{aligned}$$

Since the characteristic of K is not 2, we see $Z_3 \neq 0$. Likewise, we compute

$$X_3 = Z_4(2X_1X_2)^2.$$

Then,

$$\frac{X_3}{Z_3} = \frac{Z_4(X_2)^2}{X_4(Z_2)^2}.$$

Using $X_4/Z_4 = X_2/Z_2$, we get

$$\frac{X_3}{Z_3} = \frac{X_2}{Z_2} = \mathcal{X}(P_2) = \mathcal{X}(P_1 + P_2).$$

Suppose $P_2 = \mathcal{O}$. The previous argument is symmetric in whether P_1 or P_2 is $\mathcal{O}$.

Suppose $P_1 = -P_2$. First note $P_3 = P_1 + P_2 = -P_2 + P_2 = \mathcal{O}$. Hence, we need to prove $X_3 \neq 0$ and $Z_3 = 0$. Observe, that $X_1/Z_1 = \mathcal{X}(P_1) = \mathcal{X}(P_2) = X_2/Z_2$, and hence, $X_1Z_2 = X_2Z_1$. Using this, we find

$$(X_1 - Z_1)(X_2 + Z_2) = X_1X_2 + X_1Z_2 - X_2Z_1 - Z_1Z_2 = X_1X_2 - Z_1Z_2, \text{and}$$
$$(X_1 + Z_1)(X_2 - Z_2) = X_1X_2 - X_1Z_2 + X_2Z_1 - Z_1Z_2 = X_1X_2 - Z_1Z_2.$$

Clearly, $Z_3 = 0$. On the other hand, X_3 simplifies to

$$Z_4(2X_1X_2 - 2Z_1Z_2)^2.$$

Suppose $X_3 = 0$, then, since $Z_4 \neq 0$, $X_1X_2 = Z_1Z_2$. Using $X_1/Z_1 = X_2/Z_2$, we get $(X_1)^2 = (Z_1)^2$ and $(X_2)^2 = (Z_2)^2$. We also assumed $P_1 = -P_2$, hence, either $X_1/Z_1 = X_2/Z_2 = -1$ or $X_1/Z_1 = X_2/Z_2 = 1$. In both cases, using equation (2.6), we get

$$\mathcal{X}(P_1 - P_2) = \mathcal{X}(2P_1) = \frac{(\mathcal{X}(P_1)^2 - 1)^2}{4\mathcal{X}(P_1)(\mathcal{X}(P_1)^2 + a\mathcal{X}(P_1) + 1)} = 0.$$

Note that the denominator is nonzero by Lemma 2.13 and since $\mathcal{X}(P_1) \neq 0$. Hence, $X_4 = 0$. Contradiction.

We conclude, $X_3 \neq 0$ and $Z_3 = 0$, and hence, $\mathcal{X}(P_3) = X_3/Z_3$.

Suppose $P_1 \neq P_2$, $P_1 \neq \mathcal{O}$, $P_2 \neq \mathcal{O}$, *and* $P_1 \neq -P_2$. Since $P_1 \neq -P_2$, also $P_1 + P_2 \neq \mathcal{O}$. Therefore, we want to prove $Z_3 \neq 0$. By assumption, $X_4 \neq 0$, so remains to prove $(X_1 - Z_1)(X_2 + Z_2) - (X_1 + Z_1)(X_2 - Z_2) \neq 0$. Note that $(X_1 - Z_1)(X_2 + Z_2) - (X_1 + Z_1)(X_2 - Z_2) = 2X_1Z_2 - 2X_2Z_1$. Suppose $X_1Z_2 - X_2Z_1 = 0$, then $X_1/Z_1 = X_2/Z_2$. We conclude $\mathcal{X}(P_1) = \mathcal{X}(P_2)$. This is a contradiction by Lemma 2.11.

Using equation (2.5) from Lemma 2.12, we get

$$\mathcal{X}(P_1 + P_2) \cdot \frac{X_4}{Z_4} \cdot \left(\frac{X_1}{Z_1} - \frac{X_2}{Z_2}\right)^2 = \left(\frac{X_1}{Z_1} \cdot \frac{X_2}{Z_2} - 1\right)^2$$

Multiplying both sides by $4Z_1^2Z_2^2$ yields

$$\mathcal{X}(P_1 + P_2) \cdot \frac{X_4}{Z_4} \cdot (2X_1Z_2 - 2X_2Z_1)^2 = (2X_1X_2 - 2Z_1Z_2)^2.$$

Observe that $(X_1 - Z_1)(X_2 + Z_2) + (X_1 + Z_1)(X_2 - Z_2) = 2X_1X_2 - 2Z_1Z_2$ and $(X_1 - Z_1)(X_2 + Z_2) - (X_1 + Z_1)(X_2 - Z_2) = 2X_1Z_2 - 2X_2Z_1$. Hence, we get the equation

$$\mathcal{X}(P_1 + P_2) = \frac{Z_4((X_1 - Z_1)(X_2 + Z_2) + (X_1 + Z_1)(X_2 - Z_2))^2}{X_4((X_1 - Z_1)(X_2 + Z_2) - (X_1 + Z_1)(X_2 - Z_2))^2} = \frac{X_3}{Z_4}.$$

□

The preceding proof is heavily inspired by the proof Bernstein gives in [4, Theorem B.2]. For doubling a point we can prove a similar statement, whose proof is also considerably inspired by Bernstein's [4, Theorem B.1].

Lemma 2.16. *Define the function $\mathcal{X} : M_{a,b} \to K \cup \{\infty\}$ as $\mathcal{X}(\mathcal{O}) = \infty$ and $\mathcal{X}((x, y)) = x$. Let $M_{a,b}$ be a Montgomery curve such that $a^2 - 4$ is not a square, and let $X_1, Z_1 \in K$ such that $(X_1, Z_1) \neq (0, 0)$. Define*

$$\begin{aligned} c &= (X_1 + Z_1)^2 - (X_1 - Z_1)^2, \\ X_3 &= (X_1 + Z_1)^2(X_1 - Z_1)^2, \quad \textit{and} \\ Z_3 &= c\Big((X_1 + Z_1)^2 + \frac{a-2}{4} \cdot c\Big), \end{aligned}$$

then for any point P_1 on $M_{a,b}$ such that $X_1/Z_1 = \mathcal{X}(P_1)$, we have $X_3/Z_3 = \mathcal{X}(2P_1)$.

Proof. Suppose $Z_1 = 0$. By assumption, $(X_1, Z_1) \neq (0, 0)$, hence, $X_1 \neq 0$. So $X_3 = X_1^4 \neq 0$. Also, $c = 0$, and thus, $Z_3 = 0$. Note that if $\mathcal{X}(P_1) = X_1/Z_1$, then $P_1 = \mathcal{O}$, and $2P_1 = \mathcal{O}$. Indeed $X_3/Z_3 = \infty = \mathcal{X}(2P_1)$.

Suppose $Z_1 \neq 0$ and $X_1 = 0$. Again, $X_3 \neq 0$ and $c = 0$, so also $Z_3 = 0$. If $\mathcal{X}(P_1) = X_1/Z_1 = 0$, then $P_1 = (0, 0)$. For this point, we have $2P_1 = \mathcal{O}$. Indeed $X_3/Z_3 = \infty$.

Suppose $Z_1 \neq 0$ and $X_1 \neq 0$. Let's start by determining whether $Z_3 \neq 0$. First observe that $c = 4X_1Z_1$, and hence, $c \neq 0$. Now note that $(X_1 + Z_1)^2 + \frac{a-2}{4} \cdot c = X_1^2 + aX_1Z_1 + Z_1^2$. By assumption, we have $\frac{X_1}{Z_1} \neq 0$. Lemma 2.13 tells us then, that $\big(\frac{X_1}{Z_1}\big)^2 + a\frac{X_1}{Z_1} + 1 \neq 0$, and hence $X_1^2 + aX_1Z_1 + Z_1^2 \neq 0$. Therefore $Z_3 \neq 0$.

Let P_1 a point on $M_{a,b}$ such that $\mathcal{X}(P_1) = X_1/Z_1$. We have by assumption $Z_1 \neq 0$, and thus, $P_1 \neq \mathcal{O}$. Morover, since $X_1 \neq 0$, we have $\mathcal{X}(P_1) \neq 0$. Suppose $P_1 = (x, 0)$ for some $x \in K$, then $0 = x^3 + ax^2 + x = x(x^2 + ax^2 + x)$ (since P_1 lies on $M_{a,b}$). Lemma 2.13 implies $x = 0$. Hence, we conclude $P_1 \neq \mathcal{O}$ and $P_1 \neq (0, 0)$. This also means that $P_1 + P_1 \neq \mathcal{O}$.

Using equation (2.6) from Lemma 2.12 we get

$$4 \cdot \frac{X_1}{Z_1} \cdot \mathcal{X}(2P_1) \cdot \left(\left(\frac{X_1}{Z_1}\right)^2 + a\frac{X_1}{Z_1} + 1\right) = \left(\left(\frac{X_1}{Z_1}\right)^2 - 1\right)^2.$$

Now, we multiply both sides by Z_1^4 to get

$$4X_1Z_1 \cdot \mathcal{X}(2P_1) \cdot (X_1^2 + aX_1Z_1 + Z_1^2) = (X_1^2 - Z_1^2)^2.$$

Note that $X_1^2 - Z_1^2 = (X_1 + Z_1)(X_1 - Z_1)$ and $(X_1 - Z_1)^2 = X_1^2 - 2X_1Z_1 + Z_1^2$. Hence, $X_1^2 + aX_1Z_1 + Z_1^2 = (X_1 + Z_1)^2 + (a-2)X_1Z_1$. This leads to

$$\mathcal{X}(2P_1) = \frac{(X_1 + Z_1)^2(X_1 - Z_1)^2}{4X_1Z_1\left((X_1 + Z_1)^2 + \frac{a-2}{4}(4X_1Z_1)\right)}.$$

Remember we had already verified the quantity in the denominator is nonzero. We conclude $\mathcal{X}(2P_1) = X_3/Z_3$. □

Observe that as with the formulas from Lemma 2.14, computing these formulas can be done with a few operations. Note that we can precompute $\frac{a+2}{4}$. We are then left with 2 additions, 2 subtractions, 3 multiplications, and 2 squarings in K to compute X_3 and Z_3.

Soon we will see that in our use case we want to perform both types of computations (adding different points and doubling a point) simultaneously. Suppose we have points P and Q, and we want to compute $2P$ and $P + Q$. We can then spare ourselves an addition and a subtraction by reusing $(X_1 + Z_1)$ and $(X_1 - Z_1)$ during doubling of P. After all, we also needed to compute those values during the computation of $P + Q$. We can shave of an extra multiplication by always choosing $Z_4 = 1$. This leads to a total of 4 additions, 4 subtractions, 6 multiplications, and 4 squarings to compute $2P$ and $P + Q$, given we choose $Z_4 = 1$.

2.3 Scalar multiplication algorithms

We now have the necessary baggage for the scalar multiplication algorithms. Suppose we have a scalar n and a point P on some curve. It doesn't matter whether the curve is in Weierstraß or Montgomery form for now. The most straightforward way to compute nP is to repetitively add P, i.e. computing $P + \cdots + P$. However, that would cost us unnecessarily much work.

It is not hard to come up with a much better algorithm by combining doubling and adding. The basic idea to compute nP is to start with $\mathcal{O}$, repetitively double the result, while sometimes adding P to that result. More precisely, we use the binary representation of n to determine whether to add P inbetween. For example, when $n = 11$, we compute $2(2(2(2\mathcal{O} + P)) + P) + P$. See Algorithm 2.17 for pseudocode.

Although the correctness of this algorithm should be clear, we give a proof of it, which the formalization will follow.

Lemma 2.18. *Algorithm 2.17 is correct, i.e. it respects its output conditions given the input conditions.*

Proof. We prove this by induction on m on the generalized statement "for any scalar n such that $n < 2^m$, we have ...".

For the base case $m = 0$, note that only $n = 0$ satisfies $n < 2^m$. All bits of n are 0 in this case, and we will never perform $Q \leftarrow Q + P$. Hence, Q will remain $\mathcal{O}$ at all times. This is correct, since $0P = \mathcal{O}$.

Algorithm 2.17 Double-and-add scalar multiplication

```
Input: Point $P$, scalars $n$ and $m$, $n < 2^m$
Output: $Q = nP$
1: $Q \leftarrow \mathcal{O}$
2: for $k := m$ downto 1 do
3:     $Q \leftarrow 2Q$
4:     if $k$-th bit of $n$ is 1 then
5:         $Q \leftarrow Q + P$
6:     end if
7: end for
```

Suppose our statement holds for m, and let n be a scalar such that $n < 2^{m+1}$. If $n < 2^m$, the $(m+1)$-th bit of n is 0. Hence, the first iteration is just $Q \leftarrow 2Q = 2\mathcal{O} = \mathcal{O}$. The correctness of the remaining iterations (m down to 1) follows from the induction hypothesis.

Assume $2^m \leq n < 2^{m+1}$. Then the $(m+1)$-th bit of n is 1. Observe that $n - 2^m \geq 0$ and $n - 2^m < 2^m$. From the induction hypothesis follows that the algorithm is correct for $n - 2^m$, i.e. $Q = (n - 2^m)P$ at the end. Running the algorithm on $n = (n - 2^m) + 2^m$ results in $Q = P$ after the first iteration. With another induction on m, we see that after the remaining m iterations, this results in an extra $2^m P$ compared with the result of running the algorithm on $n - 2^m$. Hence, at the end, $Q = (n - 2^m)P + 2^m P = nP$. $\square$

Unfortunately, Algorithm 2.17 is not always suitable for cryptographic applications. The time needed for the computations depends on the individual bits of n. With careful timing, an attacker could reconstruct n. In the object of our interest, Curve25519, n actually is a private key.

This is where Montgomery's ladder jumps in. With slightly more computation and an extra variable, we can prevent an attacker from obtaining useful information by timing. See Algorithm 2.19.

Although normal timing attacks on Algorithm 2.19 are pretty much impossible, there are other types of side-channel attacks which can be succesful. For example, in [17] Yarom and Benger succesfully attacked an implementation of this algorithm in the OpenSSL cryptographic library. They exploited a weakness in the X86 architecture, which allowed them to detect accesses of targeted cache lines. If the code of both branches of the if statement happen to lie on different cache lines, this can be used to detect the branch taken during each iteration.

Observe that we can avoid the if statement by swapping Q and R both at the beginning and the end of each iteration if k-th bit of n is 1. This conditional swapping can be implemented entirely arithmetically.

Suppose we want to swap to words x and y depending on the value of $b \in \{0, 1\}$. In two's complement, $b - 1$ will be the word with all bits set if $b = 0$, and the word with all bits cleared when $b = 1$. We will use the boolean operations "not", "and"

Algorithm 2.19 Montgomery ladder for scalar multiplication.

```
Input: Point $P$, scalars $n$ and $m$, $n < 2^m$
Output: $Q = nP$
 1: $Q \leftarrow \mathcal{O}$
 2: $R \leftarrow P$
 3: for $k := m$ downto 1 do
 4:     if $k$-th bit of $n$ is 0 then
 5:         $R \leftarrow Q + R$
 6:         $Q \leftarrow 2Q$
 7:     else
 8:         $Q \leftarrow Q + R$
 9:         $R \leftarrow 2R$
10:     end if
11: end for
```

and "xor" as their bitwise functions on words here. Let c be the bitwise NOT of $b-1$, i.e. $\mathrm{NOT}(b-1)$, and let t be $c \,\mathrm{AND}\, (x \,\mathrm{XOR}\, y)$. Observe that $t = 0$ if $b = 0$, and $t = x \,\mathrm{XOR}\, y$ if $b = 1$. It follows, if $b = 0$, then $x \,\mathrm{XOR}\, t = x$ and $y \,\mathrm{XOR}\, t = y$, and if $b = 1$, then $x \,\mathrm{XOR}\, t = x \,\mathrm{XOR}\, (x \,\mathrm{XOR}\, y) = y$ and likewise $y \,\mathrm{XOR}\, t = x$.

Of course, we also prove the correctness of Algorithm 2.19.

Lemma 2.20. *Algorithm 2.19 is correct, i.e. it respects its output conditions given the input conditions.*

Proof. Notice that at the beginning of every iteration in Algorithm 2.19, we have $R = Q + P$. The first branch computes $Q = 2Q$ and $R = 2Q + P$, and the second $Q = 2Q + P$ and $R = 2Q + 2P$. We see that the value of Q after each iteration coincides with the value of Q in Algorithm 2.17. With this knowledge it is easy to see that Algorithm 2.19 is correct, i.e. $Q = nP$ at the end. □

In Curve25519 we are actually only interested in the x-coordinates of the points. This is where the hard work we have done to obtain Lemma 2.14 and Lemma 2.16 pays off. Using these formulas, we get a version of the Montgomery ladder which resembles the function `crypto_scalarmult` in TweetNaCl. See Algorithm 2.21.

Here we have replaced the large if statement in Algorithm 2.19 with two smaller conditional swappings at the beginning and the end of each iteration. In actual code, these conditional swappings should be implemented arithmetically as described earlier.

Finally, we also have a correctness lemma for Algorithm 2.21.

Lemma 2.22. *Algorithm 2.21 is correct, i.e. it respects its output conditions given the input conditions.*

Proof. Let a_0, b_0, c_0, d_0 be the values of a, b, c, d at the beginning of an iteration, after the conditional swap. At the end of the iteration, before the other conditional swap,

Algorithm 2.21 Montgomery ladder for scalar multiplication on $M_{A,B}$ with optimizations.

Input: $x \in K$ such that $x \neq 0$, and scalars n and m such that $n < 2^m$

Output: $a/c = \mathcal{X}(nP)$ for any P such that $\mathcal{X}(P) = x$

```
1: $(a, b, c, d) \leftarrow (1, x, 0, 1)$
2: for $k := m$ downto 1 do
3:     if $k$-th bit of $n$ is 1 then
4:         $(a, b) \leftarrow (b, a)$
5:         $(c, d) \leftarrow (d, c)$
6:     end if
7:     $e \leftarrow a + c$
8:     $a \leftarrow a - c$
9:     $c \leftarrow b + d$
10:    $b \leftarrow b - d$
11:    $d \leftarrow e^2$
12:    $f \leftarrow a^2$
13:    $a \leftarrow c \times a$
14:    $c \leftarrow b \times e$
15:    $e \leftarrow a + c$
16:    $a \leftarrow a - c$
17:    $b \leftarrow a^2$
18:    $c \leftarrow d - f$
19:    $a \leftarrow c \times \frac{A-2}{4}$
20:    $a \leftarrow a + d$
21:    $c \leftarrow c \times a$
22:    $a \leftarrow d \times f$
23:    $d \leftarrow b \times x$
24:    $b \leftarrow e^2$
25:    if $k$-th bit of $n$ is 1 then
26:        $(a, b) \leftarrow (b, a)$
27:        $(c, d) \leftarrow (d, c)$
28:    end if
29: end for
```

we have computed

$$\begin{aligned}
a &= (a_0 + c_0)^2(a_0 - c_0)^2 \\
c &= \left((a_0 + c_0)^2 - (a_0 - c_0)^2\right)\left(\left((a_0 + c_0)^2 - (a_0 - c_0)^2\right)\frac{A-2}{4} + (a_0 + c_0)^2\right) \\
b &= ((b_0 + d_0)(a_0 - c_0) + (b_0 - d_0)(a_0 + c_0))^2 \\
d &= x((b_0 + d_0)(a_0 - c_0) - (b_0 - d_0)(a_0 + c_0))^2.
\end{aligned}$$

Note that these a and c coincide with respectively X_3 and Z_3 from Lemma 2.16 if we let $X_1 = a_0$ and $Z_1 = c_0$. Similarly, these b and d coincide with respectively X_3 and Z_3 from Lemma 2.14 if we let $X_1 = a_0$, $Z_1 = c_0$, $X_2 = b_0$, $Z_2 = d_0$, $X_4 = x$ and $Z_4 = 1$.

Suppose Q and P are points on $M_{A,B}$ such that $\mathcal{X}(P) = x$, $\mathcal{X}(Q) = a_0/c_0$ and $\mathcal{X}(Q + P) = b_0/d_0$, then $a/c = \mathcal{X}(2Q)$ and $b/d = \mathcal{X}(Q + (Q + P)) = \mathcal{X}(2Q + P)$ before the last conditional swap. Taking the conditional swaps at the beginning and end of each iteration into account, we see a strong resemblence to Algorithm 2.19. The correctness proof of that algorithm, Lemma 2.20, implies the correctness of this algorithm. □

2.4 Curve25519

We turn our attention to Curve25519. Here we will define an algorithm strongly resembling the Curve25519 code in TweetNaCl. We begin with some preliminaries.

Curve25519 uses a finite field $\mathbf{F}_p$ with p a specific odd prime. In addition to the finite field $\mathbf{F}_p$, we also construct the extension $\mathbf{F}_{p^2}$. We do this by choosing a non-square $\delta \in \mathbf{F}_p$, and constructing $\mathbf{F}_p(\sqrt{\delta})$.

Lemma 2.23. *Let p be an odd prime, and $\delta \in \mathbf{F}_p$ a non-square in $\mathbf{F}_p$. Every $c \in \mathbf{F}_p$ has square root in $\mathbf{F}_p(\sqrt{\delta})$.*

Proof. We assume that $c \in \mathbf{F}_p$ is a non-square in $\mathbf{F}_p$, otherwise, c already has a square root in $\mathbf{F}_p$. First note that there are exactly $\frac{p-1}{2}$ different non-squares in $\mathbf{F}_p$, and there are also exactly $\frac{p-1}{2}$ different nonzero squares in $\mathbf{F}_p$. It is easy to verify that multiplying a non-square by a nonzero square yields another non-square. Hence,

$$\{cd \mid d \in \mathbf{F}_p \text{ is a nonzero square}\} \subseteq \{\text{non-squares in } \mathbf{F}_p\}. \tag{2.8}$$

Note that multiplication by c is injective, i.e. for all $d, e \in \mathbf{F}_p$, if $d \neq e$, then $cd \neq ce$. Hence, $\#\{cd \mid d \in \mathbf{F}_p \text{ is a nonzero square}\} = \#\{\text{nonzero squares in } \mathbf{F}_p\}$. Note that we have $\#\{\text{nonzero squares in } \mathbf{F}_p\} = \#\{\text{non-squares in } \mathbf{F}_p\}$. Hence, we can improve (2.8) to

$$\{cd \mid d \in \mathbf{F}_p \text{ is a nonzero square}\} = \{\text{non-squares in } \mathbf{F}_p\}.$$

By a counting argument, multiplying c by a non-square in $\mathbf{F}_p$ yields a nonzero square in $\mathbf{F}_p$. It is easy to verify that the multiplicative inverse of δ is also a non-square. Hence, c/δ is a nonzero square in $\mathbf{F}_p$. Choose a square root $r \in \mathbf{F}_p$ of c/δ. Then, $r\sqrt{\delta} \in \mathbf{F}_p(\sqrt{\delta})$, and

$$(r\sqrt{\delta})^2 = \frac{c}{\delta} \cdot \delta = c.$$

□

Curve25519 only uses the x-coordinates as input and output. However, there can be more than one point on a Montgomery curve with the given x-coordinate. So we want to assure ourselves that the particular choice of the point doesn't affect the outcome. The following theorem states just that.

Theorem 2.24. *Let p be a prime number with $p \geq 5$. Let a be an integer such that a^2-4 is not a square in $\mathbf{F}_p$. Let δ a non-square in $\mathbf{F}_p$ and construct $\mathbf{F}_{p^2} = \mathbf{F}_p(\sqrt{\delta})$. We consider the Montgomery curve $M_{a,1}$ defined over $\mathbf{F}_{p^2}$. Define the function $\mathcal{X}_0 : M_{a,1} \to \mathbf{F}_{p^2}$ as $\mathcal{X}_0(\mathcal{O}) = 0$ and $\mathcal{X}_0((x,y)) = x$. Let k be an integer, and let q be an element of $\mathbf{F}_p$, then there exists a unique $s \in \mathbf{F}_p$ such that $\mathcal{X}_0(kQ) = s$ for all Q on $M_{a,1}$ such that $\mathcal{X}_0(Q) = q$.*

Proof. First observe that $M_{a,1}$ has the subgroups $\{\mathcal{O}, (0,0)\}$, $\{\mathcal{O}\} \cup (M_{a,1} \cap (\mathbf{F}_p \times \mathbf{F}_p))$ and $\{\mathcal{O}\} \cup (M_{a,1} \cap (\mathbf{F}_p \times \sqrt{\delta}\mathbf{F}_p))$. Only the last of these three may not be entirely evident. Suppose we want to add the points $(x_1, y_1), (x_2, y_2) \in M_{a,1} \cap (\mathbf{F}_p \times \sqrt{\delta}\mathbf{F}_p)$. Let $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$. Whether $(x_1, y_1) = (x_2, y_2)$ or not, in both cases, λ in the addition formulas from Lemma 2.8 will be an element of $\sqrt{\delta}\mathbf{F}_p$. It then readily follows that $x_3 = \lambda^2 b - a - x_1 - x_2 \in \mathbf{F}_p$, and $y_3 = -y_1 - \lambda(x_3 - x_1) \in \sqrt{\delta}\mathbf{F}_p$.

Let k be an integer, and let q be an element of $\mathbf{F}_p$. We will show that there exist exactly two Q on $M_{a,1}$ such that $\mathcal{X}_0(Q) = q$. Also, both these Q have the same value $\mathcal{X}_0(kQ)$. Define $\alpha = q^3 + aq^2 + q$.

Case $\alpha = 0$. Note that by Lemma 2.13, we have $q = 0$. Clearly, $\mathcal{X}_0(Q) = 0$ if and only if $Q = \mathcal{O}$ or $Q = (0,0)$. Since $\{\mathcal{O}, (0,0)\}$ forms a subgroup, we have $kQ \in \{\mathcal{O}, (0,0)\}$, and hence, $\mathcal{X}_0(kQ) = 0$ for both Q.

Case α is a nonzero square in $\mathbf{F}_p$. Choose a square root $r \in \mathbf{F}_p$ of α. Note that there are at most two square roots in of α in $\mathbf{F}_{p^2}$, and we know that r and $-r$ are two different square roots. Hence, the only Q on $M_{a,1}$ with $\mathcal{X}_0(Q) = q$ are $Q = (q, r)$ and $Q = (q, -r)$. Remember that $\{\mathcal{O}\} \cup (M_{a,1} \cap (\mathbf{F}_p \times \mathbf{F}_p))$ is a subgroup of $M_{a,1}$. Hence, $\mathcal{X}_0(kQ) \in \mathbf{F}_p$ for both Q. Also, $\mathcal{X}_0(k(q,-r)) = \mathcal{X}_0(k(-(q,r))) = \mathcal{X}_0(-k(q,r)) = \mathcal{X}_0(k(q,r))$.

Case α is a non-square in $\mathbf{F}_p$. By Lemma 2.23 there exists a square root $r \in \mathbf{F}_{p^2}$. Since α is a non-square in $\mathbf{F}_p$, necessarily, $r \in \sqrt{\delta}\mathbf{F}_p$. Now, the only two Q on $M_{a,1}$ with $\mathcal{X}_0(Q) = q$ are $Q = (q, r)$ and $Q = (q, -r)$. Note that both lie on the subgroup $\{\mathcal{O}\} \cup (M_{a,1} \cap (\mathbf{F}_p \times \sqrt{\delta}\mathbf{F}_p))$. The argument is now the same as in the case when α is a nonzero square in $\mathbf{F}_p$. □

Having the preliminaries in the pocket, we can look at the concrete definition of Curve25519. For Curve25519, we consider the finite field $\mathbf{F}_{2^{255}-19}$ and the Montgomery curve $M_{486662,1}$. First to prove is the primality of $2^{255} - 19$. We state this as a full-blown lemma mainly to refer to it conveniently later on.

Lemma 2.25. *The integer* $2^{255} - 19$ *is a prime.*

Proof. This can be easily checked by many programs available on the Internet. □

A second condition we often need, is the non-squareness of $a^2 - 4$.

Lemma 2.26. *Let* $p = 2^{255} - 19$ *and* $a = 486662 \in \mathbf{F}_p$*, then* a *is not a square in* $\mathbf{F}_p$*.*

Proof. We use Euler's criterion to prove this. By Fermat's little theorem, we have $c^{p-1} \equiv 1 \pmod{p}$ for any integer c which isn't divisible by p. Hence, we have $c^{(p-1)/2} \equiv -1 \pmod{p}$ or $c^{(p-1)/2} \equiv 1 \pmod{p}$.

Suppose c is a nonzero square in $\mathbf{F}_p$, i.e. there exists a $r \in \mathbf{F}_p$ such that $r^2 = c$, then we have $c^{(p-1)/2} = (r^2)^{(p-1)/2} = r^{p-1} = 1$. Note that there are at most $\frac{p-1}{2}$ roots of the polynomial $X^{(p-1)/2} - 1 \in \mathbf{F}_p[X]$, and since there exist $\frac{p-1}{2}$ different nonzero squares in $\mathbf{F}_p$, we know all roots of $X^{(p-1)/2} - 1$. So, if c is a non-square in $\mathbf{F}_p$, then $c^{(p-1)/2} \equiv -1 \pmod{p}$, otherwise c is a root of $X^{(p-1)/2} - 1$.

Now, we actually compute $a^{(p-1)/2}$ for $p = 2^{255} - 19$ and $a = 486662$. Note that exponentiation modulo p can be done blazingly fast. The reader can verify that $a^{(p-1)/2} \equiv -1 \pmod{p}$ using any popular computer algebra system. □

From now on, let $a = 486662$ and $b = 1$. We will look at the Curve25519 curve, $M_{a,b}$, defined over $\mathbf{F}_{p^2}$. Thus, points (x, y) on the Curve25519 curve satisfy the equation $y^2 = x^3 + 486662x^2 + x$. By Lemma 2.23, for any $x \in \mathbf{F}_p$, there exists an $y \in \mathbf{F}_{p^2}$, such that (x, y) lies on $M_{a,b}$.

Remember from Chapter 2 that not all strings of 32 bytes are valid private keys. The set of valid Curve25519 private keys are $\{0, 8, 16, \ldots, 248\} \times \{0, 1, \ldots, 255\}^{30} \times \{64, 65, \ldots, 127\}$. Choosing a private key can be done by randomly choosing a 32-byte string, unsetting the three least significant bits and the most significant bit, and setting the second most significant bit. However, we should never trust user input, so perform this unsetting and setting of specific bits anyway.

Definition 2.27. The function Clamp : $\{0, \ldots, 255\}^{32} \to \{0, \ldots, 255\}^{32}$ is defined by unsetting the three least significant bits of the last byte, and unsetting the most significant bit and setting the second most significant bit of the first byte. Note that the image of Clamp is the set of Curve25519 private keys.

We also define a function to clarify how we interpret these strings of 32 bytes.

Definition 2.28. Let LittleEndian : $\{0, \ldots, 255\}^{32} \to \{0, \ldots, 2^{256} - 1\}$ be the little-endian interpretation of 32-byte strings. Note that LittleEndian is a bijective function, and hence, has an inverse.

We can now define the actual Curve25519 function and give an algorithm computing this function. Since this algorithm is based on Algorithm 2.21, the correctness of it readily follows.

Definition 2.29. We define the function Curve25519 : $\{$Curve25519 private keys$\} \times \{$Curve25519 public keys$\} \rightarrow \{$Curve25519 public keys$\}$ s.t. Curve25519$(N, X) =$ LittleEndian$^{-1}(s)$, where $n =$ LittleEndian(N), $x =$ LittleEndian(X), and $s = \mathcal{X}_0(nQ)$ for all Q on $M_{486662,1}$ such that $\mathcal{X}_0(Q) = x$. We consider $M_{486661,1}$ to be defined over $\mathbf{F}_{(2^{255}-19)^2}$. Note that this s is unique by Theorem 2.24. The function $\mathcal{X}_0 : M_{486662,1} \rightarrow \mathbf{F}_{2^{255}-19}$ is defined as $\mathcal{X}_0((x, y)) = x$ and $\mathcal{X}_0(\mathcal{O}) = 0$.

Algorithm 2.30 An algorithm computing the Curve25519 function using the Montgomery ladder.

```
Input: $X, N \in \{0, \ldots, 255\}^{32}$, variables $n \in \mathbf{N}$, $x, a, b, c, d \in \mathbf{F}_{2^{255}-19}$
Output: $Y = \text{Curve25519}(\text{Clamp}(N), X)$
 1: $n \leftarrow \text{LittleEndian}(\text{Clamp}(N))$
 2: $x \leftarrow \text{LittleEndian}(X) \bmod 2^{255} - 19$
 3: $(a, b, c, d) \leftarrow (1, x, 0, 1)$
 4: for $k := 255$ downto 1 do
 5:     if $k$-th bit of $n$ is 1 then
 6:         $(a, b) \leftarrow (b, a)$
 7:         $(c, d) \leftarrow (d, c)$
 8:     end if
 9:     $e \leftarrow a + c$
10:     $a \leftarrow a - c$
11:     $c \leftarrow b + d$
12:     $b \leftarrow b - d$
13:     $d \leftarrow e^2$
14:     $f \leftarrow a^2$
15:     $a \leftarrow c \times a$
16:     $c \leftarrow b \times e$
17:     $e \leftarrow a + c$
18:     $a \leftarrow a - c$
19:     $b \leftarrow a^2$
20:     $c \leftarrow d - f$
21:     $a \leftarrow c \times 121665$
22:     $a \leftarrow a + d$
23:     $c \leftarrow c \times a$
24:     $a \leftarrow d \times f$
25:     $d \leftarrow b \times x$
26:     $b \leftarrow e^2$
27:     if $k$-th bit of $n$ is 1 then
28:         $(a, b) \leftarrow (b, a)$
29:         $(c, d) \leftarrow (d, c)$
30:     end if
31: end for
32: $Y \leftarrow \text{LittleEndian}^{-1}(a/c)$
```

Chapter 3

Formalization

The main mathematical statements in the preceding chapter are all formally verified, except for the last section about Curve25519. An overview of where to find the corresponding part of the formalization can be found in Table 3.1. We used the proof assistant Coq 8.5pl1 along with the Mathematical Components (including Ssreflect) 1.6 library.

We estimate that the actual formalization cost us 120 hours. Besides these 120 hours, considerable time went into getting acquainted with Ssreflect and the Mathematical Components library. The formalization comprises 1628 lines of code (excluding blank lines and comments) in total. Table 3.2 contains a breakdown of the amount of lines of code amongst major parts of the formalization

3.1 Coq and Ssreflect

Coq is a well-known mature interactive proof assistant based on the Calculus of Inductive Constructions. Definitions and statements are written in a dependently typed lambda calculus. As such, Coq is very powerful functional language, though with the property that all programs must terminate.

The correspondence between programs and proofs is given by the Curry-Howard isomorphism. It establishes a correspondence between types and logical statements. Whether a proof exists for a statement then boils down to whether the corresponding type is inhabited. Hence, proving a statement in Coq requires constructing a term of a certain type. The user is heavily assisted with constructing terms by a powerful, extendable tactic system.

Ssreflect provides the user with an alternative set of tactics to prove goals. One of the other key features of Ssreflect is the increased control on bookkeeping of the proof context. Ssreflect considers a goal as a stack of hypotheses and a conclusion. Specialized tacticals help popping hypotheses from the stack into the proof context, and vice versa. These tacticals (`=>` and `:`) can be combined with most of the Ssreflect tactics. In addition, most of these tactics work on the top of the stack instead of taking a hypothesis from the context.

Table 3.1: Mapping between the mathematical content of this document and the formalization in Coq.

This document	File	Coq identifier
Definition 2.7	`mc.v`	`oncurve` and `mc`
Lemma 2.8	`mc.v`	`MCGroup.add`
Lemma 2.10	`mcgroup.v`	`ec_of_mc_bij`
		`ec_of_mc_additive`
Lemma 2.11	`montgomery.v`	`same_x_opposite_points`
Lemma 2.12	`montgomery.v`	`montgomery_neq`
		`montgomery_eq`
Lemma 2.13	`montgomery.v`	`aux_no_square`
Lemma 2.14	`montgomery.v`	`montgomery_hom_neq`
Lemma 2.16	`montgomery.v`	`montgomery_hom_eq_ok`
		`montgomery_hom_eq`
Algorithm 2.17	`ladder.v`	`doubleadd`
Lemma 2.18	`ladder.v`	`doubleadd_ok`
Algorithm 2.19	`ladder.v`	`montgomery`
Lemma 2.20	`ladder.v`	`montgomery_ok`
Algorithm 2.21	`opt_ladder.v`	`opt_montgomery`
Lemma 2.22	`ladder.v`	`opt_montgomery_ok`
Lemma 2.25	`curve25519_prime.v`	`curve25519_prime`

Table 3.2: Lines of code (excluding blank lines and comments) of the formalization. Note that a part of the formalization of Montgomery curves is an adaption of the library by Bartzia and Strub [2], and that a large part of the code for the primality of $2^{255} - 19$ is generated.

Part of formalization	Sections	LOC
Montgomery curves	Section 2.2	1119
Scalar multiplication algorithms	Section 2.3	247
Primality of $2^{255} - 19$	Section 2.4	262

An interesting consequence is that one of the most used tactics in Ssreflect, `move`, is almost useless on its own. According to the Ssreflect manual [9], `move` performs just the standard `hnf` Coq tactic, which brings a term into head normal form. We can combine it with the `=>` tactical to pop hypotheses from the stack into the proof context. For example, the tactic `move=> H1 H2`, applied to the goal `T1 -> T2 -> G`, transforms the goal into `G` and enriches the context with `H1 : T1` and `H2 : T2`. The `=>` tactical is always applied after the tactic. On the other hand, we can also push hypotheses back on the stack with `move: H1 H2`. These are applied in reverse order, hence, we end with the goal `T1 -> T2 -> G` again. In contrast with `=>`, the `:` tactical is also applied before the tactic itself.

It suffices to let the `case` tactic just perform case analysis on the top of the stack. For example, apply `case` on the goal `T1 /\ T2 -> G` results in `T1 -> T2 -> G`. We can combine `case` with the `:` tactical to perform case analysis on a hypothesis in the context. If we have `H : T1 /\ T2` in the context and `G` as goal, `case: H => H1 H2` results in `H` first being pushed to the proof stack, the conjunction being destructed by `case`, and `H1 : T1` and `H2 : T2` being added to the context.

Another key feature of Ssreflect is its strong `rewrite` tactic. It lets the user combine multiple rewrites, gives more control on what occurrence is rewritten, allows trivial goals to be dismissed, can perform simplification, and can fold and unfold. In standard Coq, the user can only control the rewriting by explicitly stating the number of the occurrence, for example, `rewrite {2}addcomm` rewrites the second occurrence of the left-hand side of `addcomm`. In Ssreflect, we could also state this with patterns, for example, `rewrite [x + _]addcomm` rewrites all occurrences of `x + _` (where '`_`' is a wild card).

Finally, small-scale reflection is well-supported in Ssreflect. In fact, this is where the name is derived from. Coq already has some tactics which utilize reflection to solve goals. For example, the standard library of Coq contains the `ring` tactic, which can decide equality between two calculations in a ring by reflecting the actual computational functions to syntactic objects describing the computation, and bringing both sides to a normal form.

However, the user never sees nor can manipulate the syntactic objects created by `ring`. The Ssreflect tactics instead support a fine-grained control over which subterms to reflect. For example, most predicates in Ssreflect are defined as functions to `bool` (the boolean type with constructors `true` and `false`). Suppose the user has the hypothesis `(a && b) = true`, where `a` and `b` are booleans and `&&` is the boolean conjunction. Ssreflect conveniently enables the reflection to `(a = true) /\ (b = true)`, which can be broken up in two hypotheses easily. Note that Ssreflect contains the coercion `is_true b := (b = true)`, which coerces a value of `bool` to a value of `Prop`. The user thus sees only `a && b` and `a /\ b`.

Along with the extension itself, the team behind Ssreflect developed an extensive library of mathematics, the Mathematical Components library. They have found an innovative way to utilize Coq's canonical structures and coercions mechanisms to imitate typeclasses [11]. This way, they managed to encode a large algebraic hierarchy which is easy in use. For our own work we used the Ssreflect extension and a substantial part of the Mathematical Components library.

We use a library for elliptic curves in Ssreflect developed by Bartzia and Strub [2]. Their main feature is an isomorphism between an elliptic curve an its Picard group of divisors. They use this to prove the points on an elliptic curve form a group.

Unfortunately, their library is still young and only supports elliptic curves in Weierstraß form. We have adapted their definitions to the form of elliptic curves we need (the Montgomery form) and proved the correspondence between those two forms. This does give us a high certainty our adaptions are correct, and we were able to transport associativity[1] from Weierstraß curve groups to our Montgomery curve groups.

Many of the statements we have proved about Montgomery form elliptic curves are quite computational. For this, we heavily relied on the `ssring` and `ssrfield` tactics, adaptions of the `ring` and `field` tactics to Ssreflect. We are convinced that nicer arguments exist in some cases, but those could not be found in a quick literature scan. We did not pursue this.

Finally, Laurent Théry provided a proof of the primality of $2^{255} - 19$. This number is too large for Coq to prove its primality directly from the definition of primality. An external program was used to generate a primality certificate for this number. This certificate comprises several subproofs of progressively larger primes using primality theorems like Pocklington's. The certificate is significantly less expensive to check than proving primality directly, although generating a certificate can be more expensive. The CoqPrime library [15] contains the necessary theory for verifying these certificates, and hence, for proving primality of large numbers.

3.2 Montgomery curves

We use the library from [2] as a basis for our development of the theory of Montgomery curves. Unfortunately, that library only supports elliptic curves in Weierstraß form. It is not our goal to generalize the library to support all forms of elliptic curves. Hence, we adapt the definitions in the original library to Montgomery curves. We show a correspondence between the two types of curves in the spirit of Lemma 2.10.

The core of our definitions are `mcuType` and `oncurve`. The first, `mcuType`, describes the parameters a and b of a Montgomery curve $M_{a,b}$. It contains the two parameters and proofs of $b \neq 0$ and $a^2 \neq 4$. The second, `oncurve`, is a computable predicate which tells if the given point lies on the given curve.

We also copy and adapt the definitions of addition from the original library, both the normal algebraic definition (as in Lemma 2.8) and the geometric definition (like Definition 2.3). Along with a proof that the two definitions are equivalent, this gives us a high level of confidence that the definition is correct.

[1]The associativity property of elliptic curve groups is remarkably hard to prove, much hard than the other group properties.

However, at this point we don't know yet that the points on a Montgomery curve form a group in our formalization. The group laws are easy to prove, except for associativity. Instead of adapting the whole argument in the original library, we take an easier approach.

We define a function `ecuType_of_mcuType` which maps the parameters of a Montgomery curve to parameters of a Weierstraß curve. Then we define functions `ec_of_mc_point` and `mc_of_ec_point` that map the points between the two types of curves. These functions coincide with Lemma 2.10.

We prove that `ec_of_mc_point` and `mc_of_ec_point` cancel each other, and prove that `ec_of_mc_point` is additive[2]. With this, and the fact that addition on Weierstraß curves is associative, we can prove that addition on our Montgomery curves is associative. Hence, we prove that the points on a Montgomery curve form a group.

Proving the additiveness of `ec_of_mc_point` can be done without deep mathematics, but it requires a lot of algebraic rewriting then. For this purpose, we heavily rely on the `ssring` and `ssfield` tactics. These tactics are adaptions to Ssreflect of the standard Coq `ring` and `field` tactics. They all use reflection to decide (and construct a proof) whether computations in a ring or a field are equal.

Neither do we use deep mathematics to prove the formulas of Lemma 2.12, Lemma 2.14 and Lemma 2.16 correct, but rely on the `ssring` and `ssfield` to do the bulk of rewriting.

3.3 Scalar multiplication algorithms

The main result here is the correctness of Algorithm 2.21. To this end, we also formalize Algorithm 2.17 and Algorithm 2.19. We give correctness proofs for all three algorithms, which closely follow the correctness proofs given here.

For each algorithm, we define a `Fixpoint` which comprises the loop body. Since in all three cases the loop is a simple "for" statement, termination is not a problem. We use `let .. := .. in` to emulate variable assignment.

All in all, the definitions in Coq (a functional programming language) closely resembles the procedural algorithms in this document. This resemblance is important to create trust in the formalization. For example, a worried user can easily verify that our algorithm matches the algorithm of TweetNaCl. See Figure 3.1.

[2]A function $f : A \to B$ is additive when $f(x +_A y) = f(x) +_B f(y)$.

Figure 3.1: Comparison between the loop body of `crypto_scalar_mult` in TweetNaCl and the definition of `opt_montgomery_rec`.

```c
...
for(i=254;i>=0;--i) {
  r=(z[i>>3]>>(i&7))&1;
  sel25519(a,b,r);
  sel25519(c,d,r);
  A(e,a,c);
  Z(a,a,c);
  A(c,b,d);
  Z(b,b,d);
  S(d,e);
  S(f,a);
  M(a,c,a);
  M(c,b,e);
  A(e,a,c);
  Z(a,a,c);
  S(b,a);
  Z(c,d,f);
  M(a,c,_121665);
  A(a,a,d);
  M(c,c,a);
  M(a,d,f);
  M(d,b,x);
  S(b,e);
  sel25519(a,b,r);
  sel25519(c,d,r);
}
...
```

```coq
Fixpoint opt_montgomery_rec (n m : nat)
    (x a b c d : K) : K :=
  if m is m.+1 then
    let (a, b) := cswap (bitn n m) a b in
    let (c, d) := cswap (bitn n m) c d in
    let e := a + c in
    let a := a - c in
    let c := b + d in
    let b := b - d in
    let d := e^+2 in
    let f := a^+2 in
    let a := c * a in
    let c := b * e in
    let e := a + c in
    let a := a - c in
    let b := a^+2 in
    let c := d - f in
    let a := c * ((M#a - 2%:R) / 4%:R) in
    let a := a + d in
    let c := c * a in
    let a := d * f in
    let d := b * x in
    let b := e^+2 in
    let (a, b) := cswap (bitn n m) a b in
    let (c, d) := cswap (bitn n m) c d in
    opt_montgomery_rec n m x a b c d
  else
    a / c.
```

Chapter 4

Conclusion

The elliptic curve library by Bartzia and Strub [2] provides a good framework to base formalizations using elliptic curve theory on. Together with Ssreflect and the Mathematical Components library, this creates a pleasant formalization environment. Note that we didn't even use some of the deeper theory in Bartzia's elliptic curve library. A couple of our statements can probably proved more elegantly with this deeper theory.

Although we didn't prove the correctness of TweetNaCl's data representations, the high similarity between TweetNaCl's `crypto_scalarmult` function and `opt_montgomery` in our formalization does provide much confidence that the high-level algorithm implemented by TweetNaCl is correct. The correctness statement of our `opt_montgomery` is also easy to read, which is an important factor for creating trust in the formalization.

4.1 Future work

We didn't provide a concrete instance of our algorithm for the Curve25519 curve. This would require the construction of the finite field $\mathbf{F}_{(2^{255}-19)^2}$, and proving that $486662^2 - 4$ is not a square. Our formalization does already contain a proof of the primality of $2^{255} - 19$. We don't expect this would be very hard.

An interesting other next step would be proving TweetNaCl's representation of elements of $\mathbf{F}_{2^{255}-19}$ correct. TweetNaCl uses a radix 2^{16} representation, where it stores the 16 limbs in signed 64-bit integers. There are finicky details about operations on these representation which can lead to subtle bugs [16].

The logical next step would be to use a formal semantics of C, to prove the functional correctness of the actual code in TweetNaCl. The (unpublished) work by Lennart Beringer on the Salsa20 implementation of TweetNaCl, which uses the Verifiable C semantics and the Verified Software Toolchain, could be useful for this purpose.

Appendix A

Formalization overview

We give an aggregation of the most important Coq definitions and statements in our formalization. Note that in the actual formalization we try to be a bit more general. For example, here, we let `K` be a finite field, though most of the time we can get away with letting `K` be a ring or a general field. We also copied the definition of `point` from Bartzia's library [2] for clarity.

```
Inductive point (K : Type) : Type :=
| EC_Inf : point K
| EC_In  : K -> K -> point K.
Notation "∞"              := (@EC_Inf _)     (at level 8).
Notation "(| x , y |)" := (@EC_In _ x y) (at level 8, x, y at level
    15).

Record mcuType (K : ringType) := {
  cA : K;
  cB : K;
  _   : cB != 0;
  _   : cA^+2 != 4%:R;
}.

Notation "M '#a'" := (M.(cA)) (at level 30).
Notation "M '#b'" := (M.(cB)) (at level 30).

Variable K : finECUFieldType.
(* K : finite field with characteristic not 2 or 3 *)
Variable M : mcuType K.

Definition oncurve (p : point K) := (
  match p with
  | EC_Inf     => true
  | EC_In x y => M#b * y^+2 == x^+3 + M#a * x^+2 + x
  end).

Inductive mc : Type := MC p of oncurve p.

Definition neg (p : point K) :=
  if p is (|x, y|) then (|x, -y|) else ∞.
```

```

Definition add (p1 p2 : point K) :=
  let p1 := if oncurve M p1 then p1 else ∞ in
  let p2 := if oncurve M p2 then p2 else ∞ in

    match p1, p2 with
    | ∞, _ => p2 | _, ∞ => p1

    | (|x1, y1|), (|x2, y2|) =>
      if   x1 == x2
      then if   (y1 == y2) && (y1 != 0)
           then
             let s  := (3%:R * x1^+2 + 2%:R * M#a * x1 + 1) / (2%:R * M#b * y1) in
             let xs := s^+2 * M#b - M#a - 2%:R * x1 in
               (| xs, - s * (xs - x1) - y1 |)
           else
             ∞
      else
        let s  := (y2 - y1) / (x2 - x1) in
        let xs := s^+2 * M#b - M#a - x1 - x2 in
          (| xs, - s * (xs - x1) - y1 |)
    end.

Notation "\- x"   := (neg x).
Notation "x \+ y" := (add x y).
Notation "x \- y" := (x \+ (\- y)).

Lemma ecuType_of_mcuType_discriminant_ok :
  let: a' := (3%:R - (M#a)^+2) / (3%:R * (M#b)^+2) in
  let: b' := (2%:R * (M#a)^+3 - 9%:R * (M#a)) / (27%:R * (M#b)^+3) in
  4%:R * a' ^+ 3 + 27%:R * b' ^+ 2 != 0.
Definition ecuType_of_mcuType :=
  let: a' := (3%:R - M#a^+2) / (3%:R * M#b^+2) in
  let: b' := (2%:R * M#a^+3 - 9%:R * M#a) / (27%:R * M#b^+3) in
  (@Build_ecuType K a' b' ecuType_of_mcuType_discriminant_ok).
Notation E := (ecuType_of_mcuType).

Definition ec_of_mc_point p :=
  match p with
  | ∞ => ∞
  | (|x, y|) => (|x / (M#b) + (M#a) / (3%:R * (M#b)), y / (M#b)|)
  end.
Lemma ec_of_mc_point_ok p : oncurve M p -> ec.oncurve E (ec_of_mc_point p).
Definition ec_of_mc p := EC (ec_of_mc_point_ok [oc of p]).

Definition mc_of_ec_point p :=
  match p with
  | ∞ => ∞
  | (|x, y|) => (|M#b * (x - M#a / (3%:R * M#b)), M#b * y|)
  end.
```

```
Lemma mc_of_ec_point_ok p : ec.oncurve E p -> oncurve M (mc_of_ec_point p).
Definition mc_of_ec p := MC (mc_of_ec_point_ok (oncurve_ec p)).

Lemma ec_of_mc_bij : bijective ec_of_mc.
Lemma ec_of_mc_additive : additive ec_of_mc.

Definition point_x (p : point K) :=
  if p is (|x, _|) then K_Fin x else K_Inf.
Definition point_x0 (p : point K) :=
  if p is (|x, _|) then x else 0.
Local Notation "p '#x'" := (point_x p) (at level 30).
Local Notation "p '#x0'" := (point_x0 p) (at level 30).

Definition point_is_fin (p : point K) :=
  if p is ∞ then false else true.

Lemma montgomery_neq (p1 p2 : point K) :
  let p3 := p1 \+ p2 in
  oncurve M p1 -> oncurve M p2 -> p1 != p2 ->
  all point_is_fin [:: p1; p2; p3] ->
  p3#x0 * p4#x0 * (p1#x0 - p2#x0)^+2 = (p1#x0 * p2#x0 - 1)^+2.

Lemma montgomery_eq (p1 : point K) :
  let p3 := p1 \+ p1 in
  oncurve M p1 -> all point_is_fin [:: p1; p3] ->
  4%:R * p1#x0 * p3#x0 * (p1#x0^+2 + M#a * p1#x0 + 1) = (p1#x0^+2 - 1)^+2.

Inductive K_infty :=
| K_Inf : K_infty
| K_Fin : K -> K_infty.
Definition inf_div (x z : K) :=
  if z == 0 then K_Inf else K_Fin (x / z).

Definition hom_ok (x z : K) := (x != 0) || (z != 0).

Hypothesis mcu_no_square : forall x : K, x^+2 != (M#a)^+2 - 4%:R.
Lemma aux_no_square x : x != 0 -> x^+2 + M#a * x + 1 != 0.

Lemma montgomery_hom_neq (x1 z1 x2 z2 x4 z4 : K) :
  hom_ok x1 z1 -> hom_ok x2 z2 -> (x4 != 0) && (z4 != 0) ->
  let x3 := z4 * ((x1 - z1)*(x2 + z2) + (x1 + z1)*(x2 - z2))^+2 in
  let z3 := x4 * ((x1 - z1)*(x2 + z2) - (x1 + z1)*(x2 - z2))^+2 in
  forall p1 p2 : point K,
  oncurve M p1 -> oncurve M p2 ->
  p1#x = inf_div x1 z1 ->
  p2#x = inf_div x2 z2 ->
  (p1 \- p2)#x = inf_div x4 z4 ->
  hom_ok x3 z3 && ((p1 \+ p2)#x == inf_div x3 z3).

Lemma montgomery_hom_eq_ok (x1 z1 : K) :
  hom_ok x1 z1 ->
  let x1z1_4 := (x1 + z1)^+2 - (x1 - z1)^+2 in
  let x3 := (x1 + z1)^+2 * (x1 - z1)^+2 in
  let z3 := x1z1_4 * ((x1 + z1)^+2 + ((M#a - 2%:R)/4%:R) * x1z1_4)
    in
  hom_ok x3 z3.

Lemma montgomery_hom_eq (x1 z1 : K) :
  hom_ok x1 z1 ->
  let x1z1_4 := (x1 + z1)^+2 - (x1 - z1)^+2 in
  let x3 := (x1 + z1)^+2 * (x1 - z1)^+2 in
  let z3 := x1z1_4 * ((x1 + z1)^+2 + ((M#a - 2%:R)/4%:R) * x1z1_4)
    in
  forall p : point K, oncurve M p ->
  p#x = inf_div x1 z1 ->
  (p \+ p)#x = inf_div x3 z3.

Definition bitn (n k : nat) :=
  (n %/ (2^k)) %% 2.

Fixpoint doubleadd_rec (x : mc M) (n m : nat) (y : mc M) : mc M :=
  if m is m'.+1 then
    let y' := (y *+ 2)%R in
    let y'' := if bitn n m' == 1 then (y' + x)%R else y' in
    doubleadd_rec x n m' y''
  else y.

Definition doubleadd (x : mc M) (n m : nat) : mc M :=
  doubleadd_rec x n m 0.

Lemma doubleadd_rec_ok (x y : mc M) (n m : nat) :
  n < 2^m -> doubleadd_rec x n m y = (x *+ n + y *+ 2^m)%R.

Lemma doubleadd_ok (x : mc M) (n m : nat) :
  n < 2^m -> doubleadd x n m = (x *+ n)%R.

Fixpoint montgomery_rec (n m : nat) (y z : mc M) : mc M :=
  if m is m'.+1 then
    if bitn n m' == 0 then
      let z' := (y + z)%R in
      let y' := (y*+2)%R in
      montgomery_rec n m' y' z'
    else
      let y' := (y + z)%R in
      let z' := (z*+2)%R in
      montgomery_rec n m' y' z'
  else y.

Definition montgomery (x : mc M) (n m : nat) : mc M :=
  montgomery_rec n m 0 x.

Lemma montgomery_rec_ok (x y : mc M) (n m : nat) :
  n < 2^m -> montgomery_rec n m y (x + y) = (x *+ n + y *+ 2^m)%R.

```

```
Lemma montgomery_ok (x : mc M) (n m : nat) :
  n < 2^m -> montgomery x n m = (x *+ n)%R.

Definition cswap {A : Type} (c : nat) (a b : A) :=
  if c == 1%N then (b, a) else (a, b).

Fixpoint opt_montgomery_rec (n m : nat) (x a b c d : K) : K :=
  if m is m.+1 then
    let (a, b) := cswap (bitn n m) a b in
    let (c, d) := cswap (bitn n m) c d in
    let e := a + c in
    let a := a - c in
    let c := b + d in
    let b := b - d in
    let d := e^+2 in
    let f := a^+2 in
    let a := c * a in
    let c := b * e in
    let e := a + c in
    let a := a - c in
    let b := a^+2 in
    let c := d - f in
    let a := c * ((M#a - 2%:R) / 4%:R) in
    let a := a + d in
    let c := c * a in
    let a := d * f in
    let d := b * x in
    let b := e^+2 in
    let (a, b) := cswap (bitn n m) a b in
    let (c, d) := cswap (bitn n m) c d in
    opt_montgomery_rec n m x a b c d
  else
    a / c.

Definition opt_montgomery (n m : nat) (x : K) : K :=
  opt_montgomery_rec n m x 1 x 0 1.

Lemma opt_montgomery_rec_ok (n m : nat) (p q : mc M) (a b c d : K) :
  n < 2^m -> p#x0 != 0 ->
  hom_ok a c -> hom_ok b d ->
  q#x = inf_div a c -> (p + q)#x = inf_div b d ->
  opt_montgomery_rec n m (p#x0) a b c d = (p *+ n + q *+ 2^m)#x0.

Lemma opt_montgomery_ok (n m : nat) (x : K) :
  n < 2^m -> x != 0 ->
  forall (p : mc M), p#x0 = x -> opt_montgomery n m x = (p *+ n)#x0.

Lemma curve25519_prime : prime (2^255 - 19).
```

List of Figures

List of Tables

Bibliography

[1] Andrew W. Appel. 'Verification of a Cryptographic Primitive: SHA-256'. In: *ACM Trans. Program. Lang. Syst.* 37.2 (Apr. 2015), 7:1–7:31. ISSN: 0164-0925. DOI: 10.1145/2701415.

[2] Evmorfia-Iro Bartzia and Pierre-Yves Strub. 'A Formal Library for Elliptic Curves in the Coq Proof Assistant'. In: *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings.* Ed. by Gerwin Klein and Ruben Gamboa. Cham: Springer International Publishing, 2014, pp. 77–92. ISBN: 978-3-319-08970-6. DOI: 10.1007/978-3-319-08970-6_6. URL: https://hal.inria.fr/hal-01102288.

[3] Daniel J. Bernstein. *Cryptography in NaCl.* Tech. rep. Department of Computer Science, University of Illinois at Chicago, Mar. 2009. URL: http://cr.yp.to/papers.html#naclcrypto.

[4] Daniel J. Bernstein. 'Curve25519: New Diffie-Hellman Speed Records'. English. In: *Public Key Cryptography – PKC 2006.* Ed. by Moti Yung et al. Vol. 3958. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 207–228. ISBN: 978-3-540-33851-2. DOI: 10.1007/11745853_14. URL: http://cr.yp.to/papers.html#curve25519.

[5] Daniel J. Bernstein et al. 'TweetNaCl: A Crypto Library in 100 Tweets'. In: *Progress in Cryptology - LATINCRYPT 2014: Third International Conference on Cryptology and Information Security in Latin America Florianópolis, Brazil, September 17–19, 2014 Revised Selected Papers.* Ed. by F. Diego Aranha and Alfred Menezes. Cham: Springer International Publishing, 2015, pp. 64–83. ISBN: 978-3-319-16295-9. DOI: 10.1007/978-3-319-16295-9_4. URL: http://cr.yp.to/papers.html#tweetnacl.

[6] M. Carvalho et al. 'Heartbleed 101'. In: *IEEE Security Privacy* 12.4 (July 2014), pp. 63–67. ISSN: 1540-7993. DOI: 10.1109/MSP.2014.66.

[7] Leonard S. Charlap and David P. Robbins. *CRD Expository Report 31.* Expository report. An Elementary Introduction to Elliptic Curves. Center for Communications Research, Princeton, Dec. 1988.

[8] Yu-Fang Chen et al. 'Verifying Curve25519 Software'. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 299–309. ISBN: 978-1-4503-2957-6. DOI: `10.1145/2660267.2660370`.

[9] Georges Gonthier, Assia Mahboubi and Enrico Tassi. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. Inria Saclay Ile de France, 2015. URL: `https://hal.inria.fr/inria-00258384`.

[10] Paul C. Kocher. 'Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems'. In: *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*. Ed. by Neal Koblitz. CRYPTO '96. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113. ISBN: 3-540-61512-1. DOI: `10.1007/3-540-68697-5_9`. URL: `http://dx.doi.org/10.1007/3-540-68697-5_9`.

[11] Assia Mahboubi and Enrico Tassi. 'Canonical Structures for the Working Coq User'. In: *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring and David Pichardie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 19–34. ISBN: 978-3-642-39634-2. DOI: `10.1007/978-3-642-39634-2_5`. URL: `https://hal.inria.fr/hal-00816703`.

[12] Peter L. Montgomery. 'Speeding the Pollard and Elliptic Curve Methods of Factorization'. In: *Mathematics of Computation* 48 (Jan. 1987), pp. 243–264.

[13] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. second. Vol. 106. Graduate Texts in Mathematics. Springer-Verlag New York, 2009. DOI: `10.1007/978-0-387-09494-6`.

[14] Marc Stevens et al. 'Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate'. In: *Advances in Cryptology - CRYPTO 2009: 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*. Ed. by Shai Halevi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 55–69. ISBN: 978-3-642-03356-8. DOI: `10.1007/978-3-642-03356-8_4`.

[15] Laurent Théry and Guillaume Hanrot. 'Primality Proving with Elliptic Curves'. In: *TPHOL 2007*. Ed. by Klaus Schneider and Jens Brandt. Vol. 4732. Kaiserslautern, Germany: Springer-Verlag, Sept. 2007, pp. 319–333. URL: `https://hal.inria.fr/inria-00138382`.

[16] Edwin Török. *TweetNaCl: How cr.yp.to's developers got carried away by the carry bit ;-)*. Blog. May 2014. URL: `http://blog.skylable.com/2014/05/tweetnacl-carrybit-bug/`.

[17] Yuval Yarom and Naomi Benger. 'Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack'. In: *IACR Cryptology ePrint Archive* 2014 (2014), p. 140. URL: `https://eprint.iacr.org/2014/140.pdf`.