Implementing Curve25519/X25519: A Tutorial on Elliptic Curve Cryptography

MARTIN KLEPPMANN, University of Cambridge, United Kingdom

Many textbooks cover the concepts behind Elliptic Curve Cryptography, but few explain how to go from the equations to a working, fast, and secure implementation. On the other hand, while the code of many cryptographic libraries is available as open source, it can be rather opaque to the untrained eye, and it is rarely accompanied by detailed documentation explaining how the code came about and why it is correct. This tutorial bridges the gap between the mathematics and implementation of elliptic curve cryptography. It is written for readers who are new to cryptography, and it assumes very little mathematical background. Starting from first principles, this paper shows how to derive every line of code in an implementation of the X25519 Diffie-Hellman key agreement scheme, based on the Curve25519 elliptic curve. The implementation is fast and secure; in particular, it is constant-time to prevent side-channel attacks.

CCS Concepts: • **Security and privacy → Public key encryption**; • **Theory of computation → Cryptographic primitives**.

Additional Key Words and Phrases: elliptic curve cryptography, Diffie-Hellman key agreement, implementation of cryptographic algorithms, constant-time algorithms, resistance to side-channel attacks

ACM Reference Format:
Martin Kleppmann. 2022. Implementing Curve25519/X25519: A Tutorial on Elliptic Curve Cryptography. 1, 1 (October 2022), 34 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Curve25519 [3] is a very widely deployed elliptic curve: it is used for Diffie-Hellman key agreement in the X25519 standard [24], which is a mandatory algorithm in TLS 1.3 [29], securing a huge number of HTTPS connections in web browsers worldwide. It is also used for encryption in WhatsApp [30], Signal [26], and many other systems and protocols [21]. However, the standard textbooks on elliptic curve cryptography [11, 13, 20, 23] predate Curve25519, and there are not many good resources that explain how practical implementations of this algorithm actually work. Glancing at an implementation of X25519 (see, for example, Listing 5 on page 28, or Section 5 of the X25519 standard document [24]) reveals mysterious sequences of arithmetic operations with few comments, explanation, or justification.

The goal of this tutorial is to provide an introduction to elliptic curve cryptography by means of carefully analysing every line of code of one particular implementation of X25519. We show how the algorithm is derived from basic principles, walking through the algebraic derivations step by step, and justifying their correctness. No advanced mathematics background is required: all that is needed is some basic modular arithmetic, which should be covered in most undergraduate computer science courses.

The implementation we analyse is based on TweetNaCl [9, 10], a small but practical cryptography library with the same API as NaCl [7, 8]. The name derives from the fact that the implementation fits in 100 tweets of up to 140 characters each. TweetNaCl is originally written in C, but its simplicity has made it popular for porting to various other languages, such as JavaScript [12]. Despite its simplicity, TweetNaCl has strong security properties that we expect of fully-fledged cryptography libraries: in particular, it uses constant-time algorithms to prevent side-channel attacks (that is, it performs no branches or array lookups based on secret values). Compared to the original TweetNaCl

Author's address: Martin Kleppmann, mk428@cst.cam.ac.uk, University of Cambridge, Department of Computer Science and Technology, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, United Kingdom.

2022. Manuscript submitted to ACM

implementation [10], the code in this paper has been slightly reformatted and simplified to improve readability, while leaving functionality, security, and performance unchanged (the changes are detailed in Appendix A).

TweetNaCl advertises itself as "auditable" [9] in the sense that its code is short and simple enough that its correctness can be established through code review. However, to my knowledge, no detail of any such audit has been published. The JavaScript port has indeed been professionally audited, but the report [15] does not go into any technical detail. Previous analyses of NaCl/TweetNaCl [4, 22] give justification for some of the algorithms, but also leave many details unexplained. This paper partially fills that gap: we discuss only X25519, but not the other algorithms that appear in TweetNaCl, such as the Salsa20 stream cipher, the Poly1305 authenticator, or the Ed25519 signature scheme.

Unlike HACL* [1], which contains a formally verified implementation of X25519 [31], the goal of this paper is not so much to verify that TweetNaCl is correct, but rather to teach how secure implementations of elliptic curve cryptography work by carefully studying one particular algorithm and its implementation.

2 BACKGROUND

This section briefly introduces the mathematical tools and concepts that are needed for later sections.

2.1 Modular arithmetic

The set of *integers modulo* p is $\mathbb{Z}_p = \{0, 1, \ldots, p-1\} = [0,\ p-1]$. In cryptography, computations are often performed modulo p. This means that if a calculation would return a result outside of the range of 0 to $p-1$, we let it "wrap around" by adding or subtracting p until we get a number between 0 and $p-1$, inclusive.

For example, if we are working modulo 7, then $1+1=2$, $2+1=3$, …, $5+1=6$, but $6+1$ wraps around to 0. We write this as $6+1 \equiv 0 \pmod 7$. This is similar to unsigned integer overflow in the C programming language, where e.g. operations on unsigned 32-bit integers are performed modulo 2^{32}.

In general, we say that two integers $a, b \in \mathbb{Z}$ are *congruent modulo* p, written $a \equiv b \pmod p$, if and only if there exists $k \in \mathbb{Z}$ such that $a - b = kp$. That is, we can convert between a and b by adding or subtracting p repeatedly. When we bring a number within the range of $[0,\ p-1]$ by adding or subtracting multiples of p, we call that process *reduction modulo* p.

The $\equiv$ operator for congruence modulo p behaves in many ways like an equals sign. For example, if $a \equiv b \pmod p$, then $a + c \equiv b + c \pmod p$, and $a \cdot c \equiv b \cdot c \pmod p$. That is, we can substitute one expression for another expression if those expressions are congruent modulo p.

This means that when a calculation is performed modulo p, we can reduce modulo p after each step of the calculation. For example, say we want to calculate $3 \cdot (5+6)$ modulo 7. Then we can first calculate $5+6$ and reduce it modulo 7, i.e. $5 + 6 = 11 \equiv 4 \pmod 7$, and then calculate $3 \cdot 4 = 12 \equiv 5 \pmod 7$. The result is the same as if we calculated $3 \cdot (5+6) = 3 \cdot 11 = 33 = 4 \cdot 7 + 5 \equiv 5 \pmod 7$.

Reduction modulo p is very useful when performing calculations on a computer, since it allows us to represent values in a fixed number of bits. For example, if $0 < p \leq 2^n$, any number in $\mathbb{Z}_p$ fits in n bits.

In cryptography, p is often a large prime number. For example, Curve25519 uses arithmetic modulo the prime number $p = 2^{255} - 19 =$ `0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffed` in hexadecimal (hence the name of the curve), so the numbers in the curve calculations fit in 255 bits (just under 32 bytes). We can use open source mathematics software SageMath to check that p is indeed a prime:

```
(2^255 - 19).is_prime() # returns True
```

2.2 Groups

An *abelian group* is a set E together with an operation $\bullet$. The operation combines two elements of the set, denoted $a \bullet b$ for $a, b \in E$. Moreover, the operation must satisfy the following requirements:

Closure: For all $a, b \in E$, the result of the operation $a \bullet b$ is also in E.

Commutativity: For all $a, b \in E$ we have $a \bullet b = b \bullet a$.

Associativity: For all $a, b, c \in E$ we have $(a \bullet b) \bullet c = a \bullet (b \bullet c)$.

Identity element: There exists an element $e \in E$, called the *identity element* or *neutral element*, such that for all $a \in E$ we have $e \bullet a = a \bullet e = a$.

Inverse element: For every $a \in E$ there exists an element $b \in E$ such that $a \bullet b = b \bullet a = e$, where e is the identity element. We then call b the *inverse* of a, written $b = a^{-1}$, and vice versa ($a = b^{-1}$).

A non-abelian group has all of the above properties apart from commutativity, but all the groups we deal with in this paper are abelian. The number of elements in E is called the *order* of the group. A group is a very useful abstraction since it has many nice mathematical properties, especially if the number of elements in E is a prime number (this is called a *prime order group*).

There are many possible ways of constructing a group. For example, the set of (positive and negative) integers $\mathbb{Z}$ together with the addition operator + forms a group with identity element 0, where the inverse of element $a \in \mathbb{Z}$ is $-a$. This group is infinite, since there are an infinite number of integers $\mathbb{Z}$.

To construct a finite group, we can use $\mathbb{Z}_p$, the set of integers modulo p, and the group operator is addition + followed by reduction modulo p. This group has identity element 0, and the inverse of $a \in \mathbb{Z}_p$ is $p - a$. This construction is called the *additive group of integers modulo p*.

Another finite group construction uses as elements the set of integers $\mathbb{Z}_p^* = \{1, \ldots, p-1\} = \mathbb{Z}_p \setminus \{0\}$, where p is a prime number, and the group operator is multiplication $\cdot$ followed by reduction modulo p. This group is known as the *multiplicative group of integers modulo p*, and it has identity element 1. It is not obvious that every element $a \in \{1, \ldots, p-1\}$ has an inverse (also called *multiplicative inverse*), but this fact can be shown using Bézout's identity when p is prime, and it is covered in many textbooks on number theory. In Section 2.6 we will see an algorithm for computing the multiplicative inverse.

Finally, in Section 4.2 we will use elliptic curves to construct another type of finite group. This group forms the foundation of most algorithms in elliptic curve cryptography.

Many cryptographic algorithms and protocols use a group without specifying how that group should be implemented. This works because any two groups with the same prime order are *isomorphic* to each other: that is, one group can be transformed into the other by renaming elements. Thus, informally speaking, two groups with the same prime order "behave the same", regardless of how they are implemented (although their security properties may differ). This makes the concept of a group one of the most common and useful abstractions in cryptography.

2.3 Diffie-Hellman key exchange

A common use of groups in cryptography is the *Diffie-Hellman key exchange*, which allows two parties (Alice and Bob) to establish a shared secret by communicating over an insecure channel, under the assumption that the adversary can only observe but not modify the communication. This shared secret can then be used as a key to encrypt messages between the two parties. If the adversary may actively interfere with the communication, additional authentication is required, which we do not discuss here.

For a group element g and a non-negative integer k we define the repeated application of the group operator to g as follows:

$$g^k = \underbrace{g \bullet g \bullet \cdots \bullet g}_{k \text{ times}} \quad \text{or, recursively:} \quad g^0 = e \quad \text{(the identity element), and} \quad g^k = g \bullet g^{k-1} \quad \text{for } k > 0. \tag{1}$$

We show in Section 4.4 how to compute g^k efficiently in the Curve25519 group, even for large k.

Let's say that Bob wants anybody to be able to send him encrypted messages, which we can do using Diffie-Hellman. Bob must first generate a keypair consisting of a private and public key, and make the public key available to anybody. To generate his keypair, Bob chooses a random integer j as his private key, and computes g^j for a well-known group element g. This group element g is called the *base point* or *generator*, and we will see later how it is chosen. The group element g^j is Bob's public key.

When Alice wants to send an encrypted message to Bob, she obtains Bob's public key g^j and chooses a random integer k. She computes $(g^j)^k$ and uses the result as a key for a symmetric cipher to encrypt her message to Bob. She also computes g^k and sends this group element to Bob along with her message, while k remains private. When Bob receives g^k from Alice, he uses his private key j to compute:

$$(g^k)^j = \underbrace{\underbrace{(g \bullet \cdots \bullet g)}_{k \text{ times}} \bullet \cdots \bullet \underbrace{(g \bullet \cdots \bullet g)}_{k \text{ times}}}_{j \text{ times}} = \underbrace{g \bullet \cdots \bullet g}_{j \cdot k \text{ times}} = \underbrace{\underbrace{(g \bullet \cdots \bullet g)}_{j \text{ times}} \bullet \cdots \bullet \underbrace{(g \bullet \cdots \bullet g)}_{j \text{ times}}}_{k \text{ times}} = (g^j)^k$$

Due to the associativity of $\bullet$, the group element $(g^k)^j$ computed by Bob equals the value $(g^j)^k$ that Alice used to encrypt her message. Thus, Alice and Bob obtain the same shared secret, and Bob can decrypt Alice's message. An adversary knows g and g^j (since they are public) and may learn g^k by eavesdropping as it is sent over the network. For the Diffie-Hellman key exchange to be secure, it must be computationally extremely difficult for the adversary to compute g^{jk} given g, g^j, and g^k.

2.4 Security properties of groups

The Diffie-Hellman protocol is secure if the following assumptions are true (where i, j and k are chosen uniformly at random from $\mathbb{Z}_p$ for sufficiently large p):

Hardness of discrete logarithms: Given g and g^k, it is not feasible for the adversary to compute k.

Computational Diffie-Hellman (CDH) assumption: Given g, g^j, and g^k, it is not feasible for the adversary to compute g^{jk}.

Decisional Diffie-Hellman (DDH) assumption: The adversary is given one of two tuples, either (g, g^j, g^k, g^{jk}) or (g, g^j, g^k, g^i), chosen at random with equal probability. Then the adversary must choose whether it was given the tuple containing g^{jk} or the tuple containing the random group element g^i. The DDH assumption then states that it is not feasible for the adversary to choose the correct answer with probability significantly greater than $\frac{1}{2}$ (a random guess).

If the adversary can compute discrete logarithms, they can compute j and/or k from g^j and g^k, and hence they can compute g^{jk}; therefore, assuming CDH implies assuming that discrete logarithms are hard. Moreover, if the adversary can compute g^{jk} then they can tell the difference between g^{jk} and g^i, and therefore assuming DDH implies assuming

CDH. However, the converse is not necessarily true: for example, it might be possible to compute g^{jk} directly from g^j and g^k, without first computing j or k, although no such algorithm is currently known.

How do the groups from Section 2.2 fare with respect to these security properties?

- In the additive group of integers modulo p, discrete logarithms are easy to compute, so this group is not secure. Computing g^k in this group means adding g to itself k times, which is the same as computing the product $g \cdot k$. We can compute the multiplicative inverse of g (Section 2.6) and then compute $g^{-1} \cdot g \cdot k = k$ in order to efficiently recover k.
- In $\mathbb{Z}_p^*$, the multiplicative group of integers modulo p, discrete logarithms are believed to be hard, but the DDH assumption does not hold. A detailed explanation goes beyond the scope of this paper, but it can be stated briefly: the adversary can use Euler's criterion to determine whether g^k is a quadratic residue modulo p, which leaks the least significant bit of k (i.e. whether k is odd or even). Moreover, jk is even if and only if j and/or k is even. Thus, if the adversary is given (g, g^j, g^k, g^i), and if i is odd while j or k is even (or if i is even while j and k are both odd), then the adversary knows that the fourth element of the tuple must be a random group element, not g^{jk}. This gives the adversary a significant advantage over random guessing, so the DDH assumption does not hold. It is possible to work around this problem by constructing a prime-order subgroup of $\mathbb{Z}_p^*$; for example, the Digital Signature Algorithm (DSA) does this. In this subgroup, the DDH assumption is believed to hold. However, a problem that remains is that the numbers have to be quite large in order to be secure: if we want the difficulty of computing the discrete logarithm to be similar to the difficulty of breaking a 128-bit symmetric cipher, then p needs to be over 3,000 bits long.
- In the elliptic curve (EC) group that we construct in Section 4.2, the DDH assumption is believed to be true. Moreover, computing discrete logarithms in EC groups is believed to be harder than in subgroups of $\mathbb{Z}_p^*$ for the same size of numbers, so EC groups can use 256-bit numbers to achieve the same 128-bit security level as 3,000-bit $\mathbb{Z}_p^*$ groups. This means EC group elements take less space in the network packets, and EC algorithms are faster for a given security level. This is the main benefit of using elliptic curves for cryptography instead of RSA or $\mathbb{Z}_p^*$ subgroups.

In all of these cases, it is a conjecture, not a proven fact, that the DDH assumption holds for a group. The fact that we have not yet found an efficient attack against these cryptosystems does not guarantee that such an attack does not exist. However, it's the best we have for the time being.

2.5 Finite fields

Most of the computations behind elliptic curve cryptography take place in a *finite field*. Building upon the definition of an abelian group in Section 2.2, a field is a set of elements F along with two operators (addition $a + b$ and multiplication $a \cdot b$), with the following properties:

- The set F and the addition operator $+$ form an abelian group with identity element 0. We denote the inverse of $a \in F$ in this group as $-a$. The subtraction operator $a - b$ is then shorthand for $a + (-b)$.
- The set $F \setminus \{0\}$ and the multiplication operator $\cdot$ form an abelian group with identity element 1. (0 is excluded because it has no multiplicative inverse, i.e. there is no $a \in F$ such that $0 \cdot a = 1$.) We denote the inverse of $a \in F$ in this group as a^{-1}. The division operator $\frac{a}{b}$ is then shorthand for $a \cdot (b^{-1})$.
- The addition and multiplication operators satisfy the distributive law: $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

The real numbers $\mathbb{R}$ along with the usual addition and multiplication operators form a field with an infinite set of elements, but we can also define a field where the set of elements F is finite. This is called a *finite field* (*Galois field*).

Curve25519 uses the finite field of integers modulo $p = 2^{255} - 19$. Like in Section 2.2, this field has the elements $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, with the usual addition and multiplication operators on integers, followed by reduction modulo p. The additive inverse of a modulo p is $-a = p - a$, which always exists. We discuss multiplicative inverses in Section 2.6.

In a field we can perform algebraic manipulation of expressions, such as solving equations, in much the same way as when working with the real numbers $\mathbb{R}$: addition, subtraction, multiplication and division all behave as expected. Exponentiation is defined as repeated multiplication, like in (1). Note that in a finite field of integers, division modulo p is shorthand for multiplying by the multiplicative inverse: $\frac{a}{b} = a \cdot (b^{-1})$. Thus, the result of division is still an integer modulo p (a field element), not a fraction.

2.6 Cyclic groups and multiplicative inverses

Let $a \in E$ be an element of a finite group E with operator $\bullet$ and identity element e. Consider the set of powers of a, that is, $\{a^0, a^1, \ldots\}$, where $a^0 = e$ and $a^k = a \bullet a \bullet \cdots \bullet a$ means applying a to itself k times. By the closure property, the result of $\bullet$ is always an element of the group, so the set of powers must be a subset of the group elements E. When the group is finite, the set of powers must also be finite.

The set of powers of $a \in E$ is called the *subgroup* of E that is *generated* by a. (A subgroup is a subset of group elements such that the operator still satisfies the five group properties listed in Section 2.2.) The *order* of group element a is defined to be the number of elements in the subgroup generated by a (similarly to the order of the group, which is the number of elements in the group).

In particular, one possibility is that $\{a^0, a^1, \ldots\} = E$: that is, a generates the whole group E, and so the order of a is the same as the order of the group. If such a group element a exists, the group is called *cyclic*, and a is called a *generator* of the group. It is called "cyclic" because if you examine the sequence of group elements a^0, a^1, a^2, etc. then eventually that sequence must repeat (since the group is finite); in a cyclic group, that repetition cycle contains all of the group elements, so $a^0 = a^{|E|}$, $a^1 = a^{|E|+1}$, $a^2 = a^{|E|+2}$ and so on, where $|E|$ is the number of elements in E.

It can be shown that if the order of a group $|E|$ is a prime number, then that group is always cyclic. A nice property of a cyclic group is that the generator gives us a one-to-one mapping between the integers modulo $|E|$ and the group elements E. Thus, if we choose an integer $k \in [0,\ |E| - 1]$ uniformly at random and compute a^k, the result is a group element chosen uniformly at random – i.e. every group element is equally likely to be picked with probability $\frac{1}{|E|}$. This fact is used in various cryptographic protocols.

In a finite cyclic group with generator a and identity element e we have $e = a^0 = a^{|E|} = a \bullet a^{|E|-1}$, so therefore $a^{|E|-1}$ must be the inverse element of a. In general, for any group element a with order k we have $a^k = e$.

A similar rule applies to $\mathbb{Z}_p^*$, the multiplicative group of integers modulo p. If p is a prime, this group has order $p - 1$ since 0 is not an element of the group. For any $a \in \{1, \ldots, p-1\}$, the multiplicative inverse of a modulo p is $a^{-1} = a^{p-2}$ (like before, the exponent is the group order minus 1). This follows from Fermat's little theorem,[1] which states that $a^{p-1} \equiv 1 \pmod{p}$ when p is prime and $a \not\equiv 0 \pmod{p}$. Since $a^{p-1} = a \cdot a^{p-2}$ we have that a^{p-2} is the multiplicative inverse of a modulo p. We will use this fact in Section 3.3 to compute inverses.

It is also possible to compute multiplicative inverses using the extended Euclid's algorithm. However, this approach is generally not constant-time, whereas the approach using Fermat's little theorem is easy to make constant-time.

[1] Not to be confused with Fermat's last theorem – same Fermat, different theorem.

2.7 The cofactor and small subgroups

In Section 4.2 we will define a group using the Curve25519 elliptic curve. This group has order hq, where $h = 8$ and $q = 2^{252} + 27742317777372353535851937790883648493$ is a prime number. q is only slightly larger than 2^{252}, as we can see when written in hexadecimal: $q =$ `0x1000000000000000000000000000000014def9dea2f79cd65812631a5cf5d3ed`. The parameters of Curve25519 are specifically chosen such that the order of the group is the product of a small integer h (which is called the *cofactor*, in this case 8), and a prime q that is slightly greater than a large power of two [3].

Lagrange's theorem states that if a group E has a subgroup E', then the order of E is divisible by the order of E'. Since the set of powers of a group element generates a subgroup, this means the order of any group element must be a factor of the order of the group. Therefore, each Curve25519 group element must have order either 1, 2, 4, 8, q, $2q$, $4q$, or $8q$: those are all of the factors of $8q$, and thus all of the possible orders of group elements.

The group would be simpler if it had a prime order (i.e. if the cofactor was $h = 1$), since then all group elements would have order either 1 or q. However, elliptic curve groups using Montgomery curves such as Curve25519 (see Section 4) always have a cofactor that is a multiple of 4. Since Montgomery curves have other advantages, a design decision of Curve25519 was to accept the cofactor of 8, even though it makes the group more complicated to use safely [19].

For purposes of Diffie-Hellman (Section 2.3), Curve25519 uses a group element with order q as generator g (base point); since the subgroup generated by this element has prime order, it is cyclic. However, if Alice is malicious, instead of sending group element g^k to Bob, she could send a group element s with small order, say order 8. When Bob then computes s^j, where j is Bob's private key, the result is one of only 8 possible group elements, since s generates a subgroup of order 8, and so $s^a = s^{a+8} = s^{a+16} = \ldots$. Based on how Bob decrypts the message, Alice can use brute force to detect which of these 8 possible group elements Bob obtained, and thus Alice can determine the three least-significant bits of Bob's private key. This is known as a *small subgroup confinement attack*.

One way of preventing this attack is to *validate public keys* [2]: on receiving a public key s, the recipient first checks that s is indeed an element of the group, and is not the identity element. Moreover, to verify that s has order q, the recipient checks whether $s^q = e$ where e is the identity element of the group. Alternatively, the recipient could check that $s^h \neq e$ where h is the cofactor, which works because $s^h = e$ whenever the order of s is less than q.

X25519 does not perform public key validation, and uses a different approach to prevent small subgroup confinement attacks: it ensures that the exponent is always a multiple of the cofactor $h = 8$. Thus, if an attacker sends a group element s with order less than q, the result of s^{hj} will always be the identity element, regardless of the private key j, and so the adversary learns nothing about the private key. When the correct generator g with order q is used, and $j \in [0,\ q-1]$ is uniformly distributed, the distribution of g^{hj} is the same as the uniform distribution of g^j, so the extra factor of h in the exponent is harmless.

Here is a proof sketch to show that the factor of h does not change the distribution. Let g be a generator of a subgroup of prime order q, and let g^a be any element of that subgroup, where $a \in [0,\ q-1]$ is uniformly distributed. Assume h is not a multiple of q, so $\gcd(h, q) = 1$. Then the congruence equation $iq \equiv -a \pmod{h}$ has exactly one solution $i \in [0,\ h-1]$. Let i be that solution, so $iq + a \equiv 0 \pmod{h}$, so $iq + a$ is a multiple of h. Moreover, since $0 \leq i \leq h-1$ and $0 \leq a \leq q-1$ we have $0 \leq iq + a \leq hq - 1$, so there exists exactly one $j \in [0,\ q-1]$ such that $hj = iq + a$. Since the subgroup generated by g is cyclic, we have $g^q = g^0$, and thus $g^{iq+a} = g^a$. Therefore, for every $a \in [0,\ q-1]$ there exists a unique $j \in [0,\ q-1]$ such that $g^a = g^{hj}$, and vice versa. Hence, the probability of generating g^{hj} is the same as the probability of generating g^a; since the probability distribution is uniform, that probability is $\frac{1}{q}$.

3 FINITE FIELD ARITHMETIC

Before we can implement elliptic curve operations, we first have to implement arithmetic operators (addition, subtraction, multiplication, and division) for the underlying finite field. In the case of Curve25519 this is the field of integers modulo the prime $p = 2^{255} - 19$, as defined in Section 2.5.

Most programming languages do not have built-in support for such large numbers; moreover, a general-purpose implementation of big numbers might not be suitable for cryptographic purposes, because the running time and memory access patterns of an operation may depend on the values of its inputs. Such variations in timing and memory access can be exploited by *side-channel attacks* to potentially leak secrets to an adversary.

The implementation analysed in this paper includes its own implementation of field arithmetic, which takes care to use *constant-time* algorithms whose running time and memory access patterns do not depend on the input values. The algorithms only rely on standard arithmetic and bit operators in C, which always take the same time to execute.

3.1 Addition and subtraction

We use two representations for integers modulo $p = 2^{255} - 19$: a 32-element array of 8-bit values (bytes), and a 16-element array of 16-bit values. The 32-byte representation is used for input and output, while the 16-element representation is used internally by the arithmetic operators. In both cases, a little-endian order is used, i.e. the first element of an array contains the least significant bits, and the last element contains the most significant bits.

Listing 1 shows the implementation of addition, subtraction, and multiplication modulo p. The `field_elem` datatype, defined on line 3, is used for the 16-element number representation. Even though each element initially holds just 16 bits, it is declared as a 16-element array of 64-bit signed integers in order to simplify the following computations.

We can think of a 16-element array $(a_0, a_1, \ldots, a_{15})$ as representing a number a by multiplying each element with the appropriate power of 2:

$$a = a_0 2^0 + a_1 2^{16} + a_2 2^{32} + \cdots + a_{15} 2^{240}$$

This expression is still well-defined even if individual elements a_i lie outside of the range $[0,\ 2^{16} - 1]$.

We start with the `unpack25519` function on lines 5–10 of Listing 1, which converts a number (a $\mathbb{Z}_p$ field element) from the byte array representation to the `field_elem` representation. The loop takes two adjacent bytes, `in[2*i]` and `in[2*i + 1]`, and combines them into a 16-bit value by shifting the second byte left by 8 bits and then adding them. On line 9, it forces the most significant bit (the 255th bit) to be zero, since our numbers are always less than 2^{255}. Strictly speaking, this function allows the value to be within the range $[0,\ 2^{255} - 1]$, including the values $\{2^{255} - 19,\ \ldots,\ 2^{255} - 1\}$ that are not reduced modulo p, but this does not do any harm, since the functions that take the `field_elem` type as input handle these numbers correctly.

The `fadd` function on lines 23–27 of Listing 1 adds two field elements in `field_elem` form, and similarly the `fsub` function on lines 29–33 subtracts two field elements. These functions are straightforward: they just add or subtract each of the 16 elements separately.

$$\begin{aligned} a + b &= (a_0 2^0 + a_1 2^{16} + a_2 2^{32} + \cdots + a_{15} 2^{240}) + (b_0 2^0 + b_1 2^{16} + b_2 2^{32} + \cdots + b_{15} 2^{240}) \\ &= (a_0 + b_0)\, 2^0 + (a_1 + b_1)\, 2^{16} + (a_2 + b_2)\, 2^{32} + \cdots + (a_{15} + b_{15})\, 2^{240} \end{aligned} \tag{2}$$

$$\begin{aligned} a - b &= (a_0 2^0 + a_1 2^{16} + a_2 2^{32} + \cdots + a_{15} 2^{240}) - (b_0 2^0 + b_1 2^{16} + b_2 2^{32} + \cdots + b_{15} 2^{240}) \\ &= (a_0 - b_0)\, 2^0 + (a_1 - b_1)\, 2^{16} + (a_2 - b_2)\, 2^{32} + \cdots + (a_{15} - b_{15})\, 2^{240} \end{aligned} \tag{3}$$

```c
typedef unsigned char u8;
typedef long long i64;
typedef i64 field_elem[16];

static void unpack25519(field_elem out, const u8 *in)
{
  int i;
  for (i = 0; i < 16; ++i) out[i] = in[2*i] + ((i64) in[2*i + 1] << 8);
  out[15] &= 0x7fff;
}

static void carry25519(field_elem elem)
{
  int i;
  i64 carry;
  for (i = 0; i < 16; ++i) {
    carry = elem[i] >> 16;
    elem[i] -= carry << 16;
    if (i < 15) elem[i + 1] += carry; else elem[0] += 38 * carry;
  }
}

static void fadd(field_elem out, const field_elem a, const field_elem b) /* out = a + b */
{
  int i;
  for (i = 0; i < 16; ++i) out[i] = a[i] + b[i];
}

static void fsub(field_elem out, const field_elem a, const field_elem b) /* out = a - b */
{
  int i;
  for (i = 0; i < 16; ++i) out[i] = a[i] - b[i];
}

static void fmul(field_elem out, const field_elem a, const field_elem b) /* out = a * b */
{
  i64 i, j, product[31];
  for (i = 0; i < 31; ++i) product[i] = 0;
  for (i = 0; i < 16; ++i) {
    for (j = 0; j < 16; ++j) product[i+j] += a[i] * b[j];
  }
  for (i = 0; i < 15; ++i) product[i] += 38 * product[i + 16];
  for (i = 0; i < 16; ++i) out[i] = product[i];
  carry25519(out);
  carry25519(out);
}
```

Listing 1. Field arithmetic modulo $p = 2^{255} - 19$: addition, subtraction, and multiplication.

Since the `field_elem` type uses 64-bit signed integers for each of the 16 elements, these addition or subtraction operations don't overflow or underflow, and we don't have to worry about propagating carry bits. However, we have to keep in mind that each of the elements may now be greater than 2^{16}, or negative.

3.2 Multiplication modulo p

The `fmul` function on lines 35–46 multiplies two numbers in `field_elem` form. `product` is a 31-element array of 64-bit integers, initialised to zero. On lines 39–41 we iterate in a nested loop over each of the elements of the input numbers `a` and `b`, and adding their product to the appropriate elements of `product`. This is equivalent to how we do long multiplication with pen and paper:

$$\begin{aligned} product = a \cdot b &= (a_0 2^0 + a_1 2^{16} + a_2 2^{32} + \cdots + a_{15} 2^{240}) \cdot (b_0 2^0 + b_1 2^{16} + b_2 2^{32} + \cdots + b_{15} 2^{240}) \\ &= a_0 b_0 2^{0+0} + a_1 b_0 2^{16+0} + \cdots + a_{15} b_0 2^{240+0} + a_0 b_1 2^{0+16} + a_1 b_1 2^{16+16} + \cdots + a_{15} b_{15} 2^{240+240} \\ &= a_0 b_0 2^0 + (a_1 b_0 + a_0 b_1)\, 2^{16} + (a_2 b_0 + a_1 b_1 + a_0 b_2)\, 2^{32} + \cdots + a_{15} b_{15} 2^{480} \end{aligned} \tag{4}$$

`product` now contains the product of `a` and `b` (the product of two 255-bit numbers is a 510-bit number).

In order to bring the 31-element array `product` into a 16-element `field_elem` form, we can reduce modulo p, as explained in Section 2.1. However, for performance reasons, we do not fully reduce modulo p in the multiplication function `fmul`. Instead, we reduce modulo $2p = 2\,(2^{255} - 19) = 2^{256} - 38$ on line 42 of Listing 1. This is valid because reducing modulo a multiple of p preserves all of the necessary information; we can later reduce modulo p and the end result will be the same as if we had not performed the intermediate reduction modulo $2p$.

`for (i = 0; i < 15; ++i) product[i] += 38 * product[i+16]` *almost* reduces `product` modulo $2p$ because $2^{256} = 2p + 38$:

$$\begin{aligned} product &= t_0 2^0 + t_1 2^{16} + t_2 2^{32} + \cdots + t_{15} 2^{240} + t_{16} 2^{256} + t_{17} 2^{272} \cdots + t_{30} 2^{480} \\ &= t_0 2^0 + t_1 2^{16} + t_2 2^{32} + \cdots + t_{15} 2^{240} + t_{16} 2^0\,(2p + 38) + t_{17} 2^{16}\,(2p + 38) + \cdots + t_{30} 2^{224}\,(2p + 38) \\ &\equiv t_0 2^0 + t_1 2^{16} + t_2 2^{32} + \cdots + t_{15} 2^{240} + 38\, t_{16} 2^0 + 38\, t_{17} 2^{16} \cdots + 38\, t_{30} 2^{224} \pmod{2p} \\ &= (t_0 + 38\, t_{16})\, 2^0 + (t_1 + 38\, t_{17})\, 2^{16} + \cdots + (t_{14} + 38\, t_{30})\, 2^{224} + t_{15} 2^{240} \end{aligned} \tag{5}$$

After this step the result is in elements `product[0]` to `product[15]`, and we ignore `product[16]` to `product[30]`. This computation is "almost" a reduction modulo $2p$ because, although the number now fits in the `field_elem` type with 16 elements, we have not yet done anything to bring each element within the $[0,\ 2^{16} - 1]$ range, so the number as a whole is not fully reduced modulo $2p$.

Before we continue, we should check that the calculation so far does not overflow the 64-bit variables we are using (signed integer overflow is undefined behaviour in C, so it is important to be sure that it cannot happen). When we come to use the addition, subtraction, and multiplication functions in Section 4.6, it will turn out that the result of a multiplication undergoes at most one addition or subtraction before becoming the input to another multiplication. Thus, if we assume that a multiplication returns a `field_elem` number in which each element is in the range $[0,\ 2^{16}]$, then after one addition or subtraction, we can assume that each of the elements a_i, b_i of the inputs to a multiplication is in the range $[-2^{16},\ 2^{17}]$. The greatest number of terms being added to one element in equation (4) is for t_{15}:

$$t_{15} = a_{15} b_0 + a_{14} b_1 + a_{13} b_2 + \cdots + a_1 b_{14} + a_0 b_{15}$$

Each of the products $a_i b_j$ is in the range $[-2^{33},\ 2^{34}]$, so therefore the sum of 16 of these terms must be in the range $[-2^{37},\ 2^{38}]$. Reduction modulo $2p$ in equation (5) may further multiply an element by a factor of 38; over-approximating 38 as 2^6 gives us a final range of $[-2^{43},\ 2^{44}]$. Thus, we can conclude that 64-bit arithmetic gives us plenty of headroom to complete the calculation without overflowing.[2]

Finally, in order to bring the elements back into the range $[0,\ 2^{16}-1]$, the `fmul` function first copies `product[0]` to `product[15]` into the output variable `out`, and then calls the `carry25519` function twice (lines 43–45).

The `carry25519` function (lines 12–21 of Listing 1) cleans up a value of type `field_elem` by *almost* bringing all of the elements within the $[0,\ 2^{16}-1]$ range. I say "almost" because there are edge cases in which some elements exceed that range after one or two calls to `carry25519`. In order to be certain that all of the elements are within $[0,\ 2^{16}-1]$, the function needs to be called *three* times. However, after two calls to `carry25519`, that range can only be exceeded by a small amount; for the purposes of the multiplication function, two calls are sufficient, since we just need to ensure the element values are small enough that they do not cause overflow when they become the input to subsequent multiplications.

`carry25519` iterates over the 16 elements of the `field_elem` number `elem`, performing the following for each:

(1) Line 17 computes `carry = elem[i] >> 16;` selecting all bits that are greater than the low-order 16 bits.
(2) Line 18 updates `elem[i] -= carry << 16;` which subtracts the carry bits from `elem[i]`, leaving its value within the range $[0,\ 2^{16}-1]$ (even if the case where `elem[i]` was previously negative).
(3) On line 19, the carry bits are added to the next element, except when `elem[i]` is already the last element. If we are at the last element (`i == 15`), the carry is multiplied by 38 and added to the first element, performing reduction modulo $2p$ as previously in equation (5). This operation is constant-time despite the presence of an `if` statement, since it only depends on the variable `i`, which is not secret.

If it was not for the last element's carry wrapping around to `elem[0]`, the `carry25519` function would leave all elements within $[0,\ 2^{16}-1]$. However, this carry can cause `elem[0]` to be greater than 2^{16}, or even negative. A second call to `carry25519` can fix this; however, if elements `elem[1]` to `elem[15]` are close to `0xffff`, the carry bits from `elem[0]` can cause a cascade of carries on the second call, resulting in a further carry from `elem[15]` to `elem[0]`, causing `elem[0]` to exceed `0xffff` *again*. On a third call to `carry25519`, such a carry cascade is no longer possible, since the values of the middle elements are now zero, and so the third call finally brings all elements within the range $[0,\ 2^{16}-1]$.

In any case, the number of times `carry25519` is called must be constant (not dependent on whether or not there is a carry), since otherwise the function would no longer be constant-time.

3.3 Computing the multiplicative inverse

Now that we have defined addition, subtraction, and multiplication, the missing arithmetic operation on field elements is division modulo $p = 2^{255} - 19$. As explained in Section 2.6, we perform division $\frac{b}{a}$ by computing the multiplicative inverse of the denominator, a^{-1}, and then multiplying that inverse with the numerator b.

The `finverse` function on lines 5–15 of Listing 2 computes the multiplicative inverse of its input `in`, writing the result to the output variable `out`. Both input and output are in `field_elem` form. The computation of the inverse uses Fermat's little theorem as per Section 2.6, by computing $a^{-1} \equiv a^{p-2} \pmod{p}$. The exponential a^{p-2} can be computed

[2]Incidentally, the JavaScript port of TweetNaCl uses double-precision floating-point arithmetic, since JavaScript does not support 64-bit integer arithmetic. IEEE 754 double-precision floating point numbers use an exact representation for integers in the range $[-2^{53}+1,\ 2^{53}-1]$; as we can see from this analysis, that precision is also sufficient to perform this multiplication algorithm correctly.

```
typedef unsigned char u8;
typedef long long i64;
typedef i64 field_elem[16];

static void finverse(field_elem out, const field_elem in)
{
  field_elem c;
  int i;
  for (i = 0; i < 16; ++i) c[i] = in[i];
  for (i = 253; i >= 0; i--) {
    fmul(c, c, c);
    if (i != 2 && i != 4) fmul(c, c, in);
  }
  for (i = 0; i < 16; ++i) out[i] = c[i];
}

static void swap25519(field_elem p, field_elem q, int bit)
{
  i64 t, i, c = ~(bit - 1);
  for (i = 0; i < 16; ++i) {
    t = c & (p[i] ^ q[i]);
    p[i] ^= t;
    q[i] ^= t;
  }
}

static void pack25519(u8 *out, const field_elem in)
{
  int i, j, carry;
  field_elem m, t;
  for (i = 0; i < 16; ++i) t[i] = in[i];
  carry25519(t); carry25519(t); carry25519(t);
  for (j = 0; j < 2; ++j) {
    m[0] = t[0] - 0xffed;
    for(i = 1; i < 15; i++) {
      m[i] = t[i] - 0xffff - ((m[i - 1] >> 16) & 1);
      m[i - 1] &= 0xffff;
    }
    m[15] = t[15] - 0x7fff - ((m[14] >> 16) & 1);
    carry = (m[15] >> 16) & 1;
    m[14] &= 0xffff;
    swap25519(t, m, 1 - carry);
  }
  for (i = 0; i < 16; ++i) {
    out[2*i] = t[i] & 0xff;
    out[2*i + 1] = t[i] >> 8;
  }
}
```

Listing 2. Multiplicative inverse, and converting numbers from internal representation to byte arrays.

using the *square-and-multiply* method, which is based on the following recursive relations:

$$a^{2i} = a^i \cdot a^i \qquad \text{and} \qquad a^{2i+1} = a \cdot a^i \cdot a^i.$$

That is, we first compute a^i recursively, and then square a^i to obtain a^{2i}. If the exponent is odd, we additionally multiply the result with another copy of a to obtain a^{2i+1}.

$p - 2 = 2^{255} - 21 =$ `0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffeb` is a constant, so `finverse` hard-codes the pattern of squarings and multiplications. All of the bits of $p - 2$ are 1, except for bits 2 and 4, which are 0 (where bit 0 is the least-significant bit). The loop in the `finverse` function counts down from the most-significant to the least-significant bit, squaring the current value `c` using the `fmul` function for each bit, and also multiplying `c` with the input value `in` for each bit that is 1. Even though $p - 2$ consists of 255 bits, the loop is able to start at bit 253 and save one iteration by initialising `c` to `in` instead of 1. At the end, `c` is copied to the output variable `out`.

3.4 Converting back to a byte array

The final aspect of finite field arithmetic that we need to complete is to convert from the `field_elem` representation of a number back to the byte array representation, which is done by the `pack25519` function on lines 27–48 of Listing 2. This function performs the inverse of the `unpack25519` function discussed in Section 3.1. In the process, we will also reduce the number modulo p, which ensures that every distinct element in the field of integers modulo p is represented by a unique byte string. This will be essential when we later want to use such a field element to derive an encryption key.

First, we explain the helper function `swap25519(p, q, bit)` on lines 17–25 of Listing 2. If `bit` is 1, this function swaps the content of parameters `p` and `q` (both in `field_elem` representation), and it does nothing if `bit` is 0. Since `bit` may be part of a secret value, this function cannot use a simple `if` statement, since that would not be constant-time. Instead, it must ensure that it always performs exactly the same operations, regardless of the value of `bit`.

`swap25519` first sets `c = ~(bit - 1)`, which equals 0 if `bit == 0`, and `0xffff...` if `bit == 1`. Next it iterates over each of the 16 elements of the number, and computes `t = c & (p[i] ^ q[i])` for index `i`, which equals 0 if `bit == 0`, and `p[i] ^ q[i]` if `bit == 1`. The operation `p[i] ^= t` thus has no effect if `bit == 0`, and it sets `p[i] = p[i] ^ (p[i] ^ q[i]) = q[i]` if `bit == 1`. Likewise, `q[i]` is set to `p[i]` if `bit == 1`.

Next, we turn to `pack25519`. This function first copies the input value `in` to `t`, and then calls `carry25519` on it three times. As discussed in Section 3.2, this ensures that all of the 16 elements of `t` fall within the range $[0,\ 2^{16} - 1]$. Thus, we have a 256-bit number that is *not quite* reduced modulo $2p$, since it may fall within the entire range $[0,\ 2^{256} - 1]$ (i.e. it may be slightly greater than $2p$).

To reduce that number t modulo p, notice that there are three possibilities: either $0 \leq t < p$ (i.e. t is already reduced modulo p), or $p \leq t < 2p$ (i.e. we need to subtract p from t to reduce it modulo p), or $2p \leq t < 2^{256}$ (i.e. we need to subtract $2p$ from t). However, for the function to be constant-time, we cannot simply detect which of these three is the case, and subtract the appropriate multiple of p. Instead, we have to calculate both $t - p$ and $t - 2p$, and then use a constant-time algorithm to choose which of the three values $\{t,\ t - p,\ t - 2p\}$ to return. This is done by the loop `for (j = 0; j < 2; ++j)` on lines 33–43.

Most of the loop body is taken up by the logic to subtract p from `t`, and write the result to `m`. Rather than reusing the earlier `fsub` function to subtract, followed by repeated `carry25519` calls to handle the carry, this function implements its own carry handling.

On line 34, `m[0] = t[0] - 0xffed;` subtracts from `t[0]` the least-significant 16 bits of p, which are `0xffed`. This may result in a negative number, but the subsequent `m[0] &= 0xffff;` on line 37 puts it back in the range $[0,\ 2^{16} - 1]$, after first taking the carry bit `(m[0] >> 16) & 1` and subtracting it from `m[1]` (line 36). We then repeat this for the next 14 elements of `t`, except that we subtract `0xffff` (the middle bits of p) instead of `0xffed`. For `t[15]` we subtract `0x7fff` (the most-significant 16 bits of p), place the carry bit in variable `carry`, and otherwise perform the same steps.

The carry bit of the result of subtracting $t - p$ is 1 if the result is negative, and 0 if it is zero or positive. If this bit is 1, `1 - carry` on line 42 equals 0, so `swap25519` does nothing, so the variable `t` is left unchanged and the negative result is discarded. If this bit is 0, `1 - carry` equals 1, so `swap25519` swaps the values of `t` and `m`, i.e. it updates `t` to be the non-negative value $t - p$. Thus, the subtraction of p takes effect only if the result is non-negative. After two iterations of this loop, t is guaranteed to be in the range $[0,\ p - 1]$.

Finally, on lines 44–47 the value of `t` is copied to the output byte array `out`. Each of the 16-bit elements is split into two bytes and assigned to two adjacent elements of the byte array. This completes the cycle, starting with input values that arrive as byte arrays, which may be manipulated in `field_elem` form through field arithmetic operations, with the result eventually converted back to a byte array.

4 ELLIPTIC CURVE ARITHMETIC

Curve25519 uses the curve

$$y^2 = x^3 + Ax^2 + x \tag{6}$$

which is known as a *Montgomery curve*, with parameter $A = 486662$. We will use equation (6) as our starting point; a justification for the use of this equation and the choice of A appear in the Curve25519 paper [3]. Our derivations work for any $A^2 \neq 4$; this restriction ensures the curve has the required shape. Some other curve equations are also used for elliptic curve cryptography, such as the *short Weierstrass* equation

$$y^2 = x^3 + Ax + B \tag{7}$$

but in this paper we focus on Montgomery curves.

4.1 Straight line intersecting the elliptic curve

We say that a point $P = (x, y)$ lies on the curve if (x, y) is a solution of equation (6). Figure 1 shows an example of such a curve. Notice that the curve has reflection symmetry around the x axis; more formally, if (x, y) is on the curve then $(x, -y)$ is also on the curve. This is the case because the variable y appears only in the y^2 term in (6).

For now we will treat the coordinates x and y as real numbers. In Curve25519 they are actually integers modulo $2^{255} - 19$, but we will do the following derivation using real numbers as it is easier to visualise, and allows us to use some calculus. It turns out that the end result works with any field.

If we have two points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ that both lie on the curve, we can draw a straight line through those points. If we assume that $x_P \neq x_Q$, then that straight line intersects the curve at some third point R, as shown in Figure 1(a). We will show shortly that this third point R always exists. This straight line is defined by the equation

$$y = \lambda x + c \quad \text{where the slope is} \quad \lambda = \frac{y_Q - y_P}{x_Q - x_P} \quad \text{and the } y\text{-intercept is} \quad c = y_P - \lambda\, x_P. \tag{8}$$

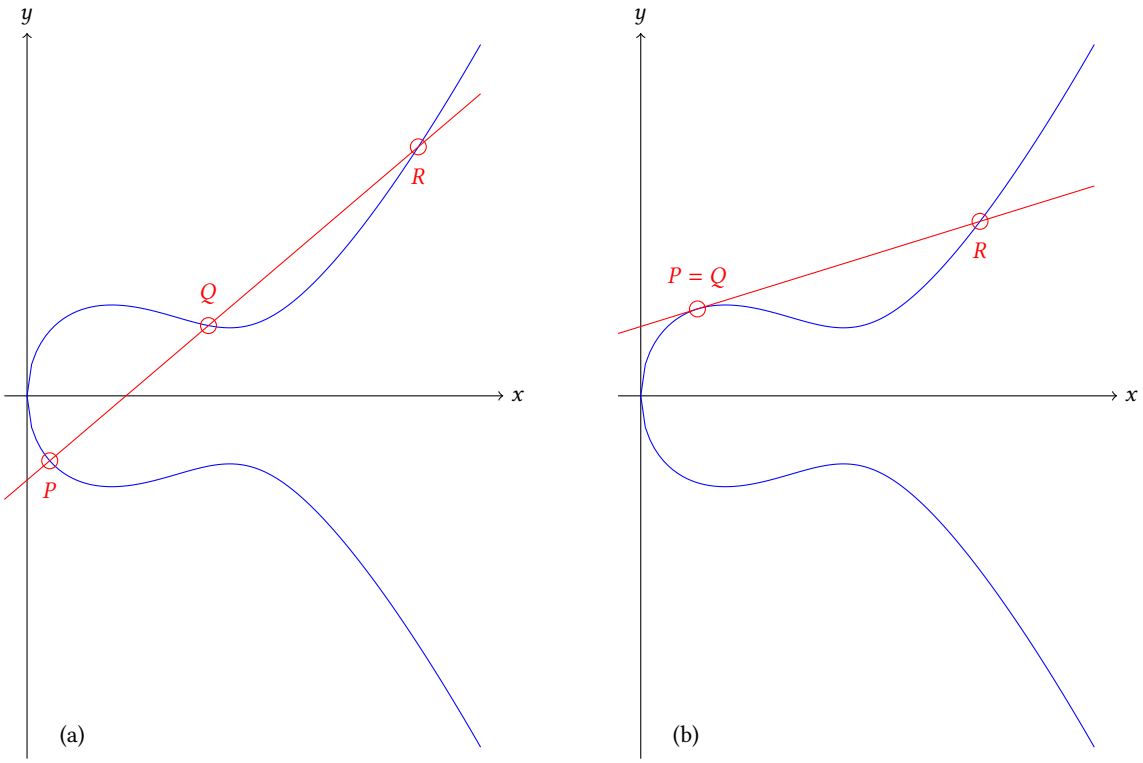

Fig. 1. (a) If we draw a line through two points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ on an elliptic curve, where $x_P \neq x_Q$, then that line intersects the curve again in a third point R. (b) Generalising to $P = Q$, we draw a tangent to the curve at P, which intersects the curve at R.

If $x_P = x_Q$ we distinguish three possibilities: either $y_P = y_Q = 0$, or $y_P = y_Q \neq 0$, or $y_P = -y_Q \neq 0$. Consider first the case where $y_P = -y_Q \neq 0$, shown in Figure 2: if we draw a straight line through the two points, that line is vertical, and there is no third intersection point. We will return to this case, and the $y_P = y_Q = 0$ case, in Section 4.2.

Next, consider the case where $y_P = y_Q \neq 0$, i.e. $P = Q$ and the points are not on the x axis. In this case we can still define a straight line through P and Q, and we choose the slope of the line such that it is a tangent to the curve (i.e. it touches the curve at P without crossing it). This is the natural generalisation of the slope $\lambda = (y_Q - y_P)/(x_Q - x_P)$ in the limit as the distance between P and Q tends to zero.

To compute the slope λ of this tangent, we can calculate the derivative of the curve equation (6) using the chain rule:

$$y^2 = x^3 + Ax^2 + x \quad \Longleftrightarrow \quad y = \pm\sqrt{x^3 + Ax^2 + x} \tag{9}$$

$$\lambda = \frac{\mathrm{d}y}{\mathrm{d}x} = \pm\frac{3x^2 + 2Ax + 1}{2\sqrt{x^3 + Ax^2 + x}} = \pm\frac{3x^2 + 2Ax + 1}{2|y|} = \frac{3x^2 + 2Ax + 1}{2y} \tag{10}$$

The sign of λ in equation (10) works out correctly for both positive and negative y. The tangent is then defined by $y = \lambda x + c$ as before, and it exists whenever $y \neq 0$. If $y = 0$, the tangent is vertical, and we handle this as part of the case $y_P = -y_Q$ in Section 4.2.

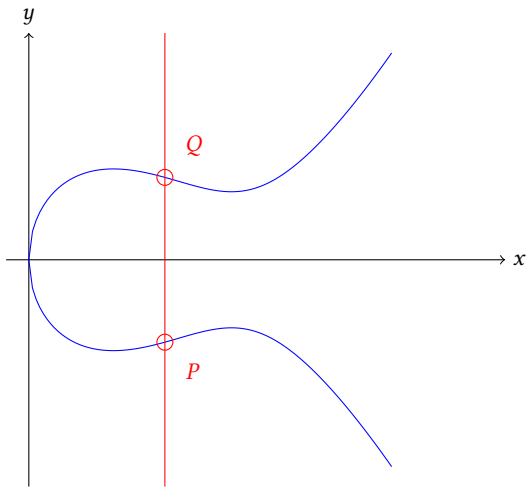

Fig. 2. If $x_P = x_Q$ and $y_P = -y_Q$, the line going through the two points is vertical.

In the cases where the straight line is not vertical, we work out the third point R at which the line intersects the elliptic curve. We do this by substituting the line equation (8) into the curve equation (6):

$$(\lambda x + c)^2 = x^3 + Ax^2 + x \quad \Longleftrightarrow \quad x^3 + (A - \lambda^2)\,x^2 + (1 - 2\lambda c)\,x - c^2 = 0 \tag{11}$$

The roots of the polynomial (11) are the x coordinates of the points at which the line intersects the curve. Since we know that P and Q lie on both the line and the curve, x_P and x_Q must be roots of (11), and so we can divide (11) by the polynomial $(x - x_P)(x - x_Q) = x^2 - (x_P + x_Q)\,x + x_P x_Q$. This works even if $x_P = x_Q$, for the following reason: in the case of $P = Q$ we chose the line to be a tangent to the curve; therefore, the derivative of (11) is zero at x_P; therefore, x_P is a double root of (11) and we can divide it by $x - x_P$ twice. Performing the polynomial division:

$$\begin{array}{rl}
 & x + A - \lambda^2 + x_P + x_Q \\
x^2 - (x_P + x_Q)\,x + x_P x_Q \;\Big)\; & x^3 + (A - \lambda^2)\,x^2 \qquad\qquad + (1 - 2\lambda c)\,x \qquad\qquad - c^2 \\
 & \underline{x^3 - (x_P + x_Q)\,x^2 \qquad\qquad + x_P x_Q x} \\
 & \qquad (A - \lambda^2 + x_P + x_Q)\,x^2 + (1 - 2\lambda c - x_P x_Q)\,x - c^2 \\
 & \qquad (A - \lambda^2 + x_P + x_Q)\,x^2 - (A - \lambda^2 + x_P + x_Q)\,(x_P + x_Q)\,x \\
 & \qquad\qquad\qquad\qquad + (A - \lambda^2 + x_P + x_Q)\,x_P x_Q \\
 & \qquad \underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad} \\
 & \qquad\qquad \ldots
\end{array}$$

The polynomial division produces an extremely ugly expression as remainder, but fortunately we do not need to compute it, since we know that it must be zero. From the quotient $x + A - \lambda^2 + x_P + x_Q$ we obtain the x coordinate of the third intersection point R:

$$x_R = \lambda^2 - A - x_P - x_Q \tag{12}$$

```
# The finite field of integers modulo p (GF is short for Galois Field)
field = GF(2^255 - 19)

# EllipticCurve(field, [a1, a2, a3, a4, a5]) constructs an elliptic curve over the
# given field, with curve equation y^2 + a1*x*y + a3*y = x^3 + a2*x^2 + a4*x + a5.
# We choose a1 = 0, a2 = 486662, a3 = 0, a4 = 1, a5 = 0 to give us the Montgomery
# curve equation y^2 = x^3 + 486662 * x^2 + x.
E = EllipticCurve(field, [0, 486662, 0, 1, 0])

# Check the order (cardinality) of the group defined by that curve
q = 2^252 + 27742317777372353535851937790883648493
q.is_prime() # returns True
E.cardinality() == 8 * q # returns True

# Define the base point (generator) g to be the point with x coordinate = 9,
# and check the order of that point
base = 9
g = [field(base), sqrt(field(base^3 + 486662 * base^2 + base))] # [x, y] coordinates
q * E(g) # returns (0 : 1 : 0), which is the point at infinity
# This indicates that point g has order q in the elliptic curve group E.
```

Listing 3. SageMath code to compute the order of the Curve25519 group and the base point.

and we obtain the y coordinate by substituting x_R into the line equation (8):

$$y_R = \lambda\, x_R + c = \lambda\, x_R + y_P - \lambda\, x_P = y_P + \lambda\,(x_R - x_P) \tag{13}$$

Since (x_R, y_R) is defined whenever λ exists, we know that the third intersection point R exists whenever the straight line is not vertical.

4.2 Constructing a group

We will now use the results from the last section to construct a group. The set of group elements is the set of points on the elliptic curve (6), plus one special element ∞ that we call the *point at infinity*:

$$E = \{(x, y) \mid y^2 = x^3 + Ax^2 + x\} \;\cup\; \{\infty\} \tag{14}$$

The group has an infinite number of elements when the x and y coordinates are real numbers, but when they are integers modulo p, the group order is finite. For $x, y \in \mathbb{Z}_p$ there are p^2 possible (x, y) pairs, and only some of them are solutions of the curve equation. The exact number of solutions depends on the curve equation, the parameter A, and the size of the underlying field. Algorithms for counting the number of curve points are presented in textbooks [11, 13]. In the case of Curve25519 with $p = 2^{255} - 19$ and $A = 486662$ we have $|E| = 8 \cdot (2^{252} + 27742317777372353535851937790883648493)$ as discussed in Section 2.7. We can use SageMath to check this is correct, as shown in Listing 3.

The point at infinity has no coordinates, and its purpose is to deal with vertical lines. When two different points have the same x coordinate and we draw a vertical line through them, like in Figure 2, we define the point at infinity to be the third point at which the line "intersects the curve". We can imagine this point as lying infinitely far up the y axis, and all vertical lines intersect that point.

The point at infinity will also serve as the identity element of our group. That is, we define the following to be true:

$$P \bullet \infty \;=\; \infty \bullet P \;=\; P \quad \text{for all } P \in E. \tag{15}$$

In particular, $\infty \bullet \infty = \infty$. Moreover, for any point $P = (x_P, y_P)$ on the curve, we define the *inverse* to be $P^{-1} = (x_P, -y_P)$, i.e. the point obtained by mirroring P with respect to the x axis. By definition, the inverse satisfies the following property:

$$P \bullet P^{-1} \;=\; P^{-1} \bullet P \;=\; \infty \quad \text{for all } P \in E. \tag{16}$$

We also define that $\infty^{-1} = \infty$.

Intuitively, we can think of the operator $P \bullet Q$ as combining two points P and Q by drawing a straight line through them and finding a third point on the curve. To fully define the operator $\bullet$, we start with the following idea: for any three points $P, Q, R \in E$, if those points lie on the same line, then we have

$$P \bullet Q \bullet R = \infty. \tag{17}$$

We can achieve this by defining $P \bullet Q = R^{-1}$, and then $(P \bullet Q) \bullet R = R^{-1} \bullet R = \infty$. That is, given two curve points P and Q, we can draw a straight line through those points, find the third point at which that line intersects the curve, and then invert that point by negating its y coordinate. The point obtained in this way is $P \bullet Q$.

Using our results (12) and (13) for the coordinates of the third intersection point, along with equations (8) and (10) for the slope λ, we can now define $P_1 \bullet P_2$ for any two points P_1 and P_2 on the curve with $P_1 \neq P_2^{-1}$:

$$P_1 \bullet P_2 = (x_1, y_1) \bullet (x_2, y_2) = (x_3, y_3) \quad \text{where}$$

$$x_3 = \lambda^2 - A - x_1 - x_2 = \begin{cases} \left(\dfrac{y_2 - y_1}{x_2 - x_1}\right)^2 - A - x_1 - x_2 & \text{if } x_1 \neq x_2 \\ \left(\dfrac{3x_1^2 + 2Ax_1 + 1}{2y_1}\right)^2 - A - 2x_1 & \text{if } x_1 = x_2 \end{cases} \tag{18}$$

$$y_3 = -(y_1 + \lambda\,(x_3 - x_1)) = \lambda\,(x_1 - \lambda^2 + A + x_1 + x_2) - y_1 = \lambda\,(2x_1 + x_2 + A) - \lambda^3 - y_1 = \tag{19}$$

$$= \begin{cases} \dfrac{(2x_1 + x_2 + A)(y_2 - y_1)}{x_2 - x_1} - \left(\dfrac{y_2 - y_1}{x_2 - x_1}\right)^3 - y_1 & \text{if } x_1 \neq x_2 \\ \dfrac{(2x_1 + x_2 + A)(3x_1^2 + 2Ax_1 + 1)}{2y_1} - \left(\dfrac{3x_1^2 + 2Ax_1 + 1}{2y_1}\right)^3 - y_1 & \text{if } x_1 = x_2 \end{cases}$$

The formulas (18) and (19), along with definitions (15) and (16), form the *group law* for Montgomery curves. These definitions may seem somewhat arbitrary, but $\bullet$ has to be defined this way in order to obtain a group. For example, if we did not invert the third intersection point of the line, the resulting operation would not form a group.

To prove that our definitions form an abelian group, we need to show that the five properties of Section 2.2 hold:

- The closure property holds by definition, since the point (x_3, y_3) defined by (18) and (19) lies on the curve, and the result of the group operation in (15) and (16) is also an element of E.
- The identity element ∞ exists and has the required behaviour according to definition (15).
- The inverse element P^{-1} exists for every $P \in E$ and has the required behaviour according to definition (16).
- To show that the commutativity property holds, consider several cases. If $P = \infty$ and/or $Q = \infty$, we have $P \bullet Q = Q \bullet P$ due to (15). If $P = Q^{-1}$, we have $P \bullet Q = Q \bullet P$ due to (16). Finally, if $P \neq \infty$, $Q \neq \infty$ and $P \neq Q^{-1}$,

consider the straight line through curve points P and Q. We show that the line through P and Q is the same as the line through Q and P, and thus the third intersection point of this line with the curve must be the same. If $P = Q$, the two lines are trivially the same according to (10). If $P \neq Q$, we examine the line equation (8):

$$y = \lambda x + c = \lambda (x - x_P) + y_P = \frac{y_Q - y_P}{x_Q - x_P} (x - x_P) + y_P \tag{20}$$

$$= \frac{-(y_P - y_Q)}{-(x_P - x_Q)} (x - x_P) + \frac{y_P (x_P - x_Q)}{x_P - x_Q}$$

$$= \frac{(y_P - y_Q) x - x_P y_P + x_P y_Q + x_P y_P - x_Q y_P}{x_P - x_Q}$$

$$= \frac{(y_P - y_Q) x + x_P y_Q - x_Q y_P + (x_Q y_Q - x_Q y_Q)}{x_P - x_Q}$$

$$= \frac{(y_P - y_Q) x - (y_P - y_Q) x_Q + (x_P - x_Q) y_Q}{x_P - x_Q}$$

$$= \frac{y_P - y_Q}{x_P - x_Q} (x - x_Q) + y_Q \tag{21}$$

The expressions (20) and (21) are equal except for swapping P and Q. Thus, we have $P \bullet Q = Q \bullet P$ for all $P, Q \in E$.

The final step is to show that $\bullet$ is associative: $(a \bullet b) \bullet c = a \bullet (b \bullet c)$. Unfortunately, proving associativity is rather more complex than the other properties: proofs of this property involve either some advanced mathematics, or the use of a computer algebra software package [17, 18]. We will therefore skip the proof of this property.

Instead, to check our result, we can look up Montgomery curves in the *Explicit Formulas Database* (EFD) [5], which lists the group law as follows:

```
name Montgomery curves
parameter a
parameter b
coordinate x
coordinate y
satisfying b y^2 = x^3 + a x^2 + x
addition x = b (y2-y1)^2/(x2-x1)^2-a-x1-x2
addition y = (2 x1+x2+a) (y2-y1)/(x2-x1)-b (y2-y1)^3/(x2-x1)^3-y1
doubling x = b (3 x1^2+2 a x1+1)^2/(2 b y1)^2-a-x1-x1
doubling y = (2 x1+x1+a) (3 x1^2+2 a x1+1)/(2 b y1)-b (3 x1^2+2 a x1+1)^3/(2 b y1)^3-y1
```

This database uses a slightly more general form $By^2 = x^3 + Ax^2 + x$ with an additional parameter B. We can see that with $B = 1$, the formulas in the database equal our formulas (18) and (19). The formulas in the EFD are checked using SageMath; we will not repeat those checks here.

A note on notation. In the literature on elliptic curves, the group operation $\bullet$ is traditionally written as $+$, the inverse of P is written as $-P$, combining two different points P and Q using the group operation $P + Q$ is called *point addition*, and combining a point P with itself $P + P = 2P$ is known as *point doubling*. On the other hand, in the literature on cryptographic protocols, the group operation is traditionally written as multiplication $\cdot$, the inverse of P is written as

P^{-1}, and combining a group element with itself is written as $P \cdot P = P^2$. In this paper we use $\bullet$ as the group operation on *elliptic curve group elements* E, in order to avoid confusion with the addition and multiplication of individual *field elements* (i.e. integers modulo p), which is used in the formulas above.

4.3 Elliptic-curve Diffie-Hellman

Now that we have constructed a group, we can use this group for cryptographic protocols. In particular, we can use the Curve25519 group to implement the X25519 Diffie-Hellman key exchange function.

For a group element $P \in E$ and a non-negative integer k we define the repeated application of the group operator to P as before:

$$P^k = \underbrace{P \bullet P \bullet \cdots \bullet P}_{k \text{ times}} \tag{22}$$

We show in Section 4.4 how to compute P^k efficiently, even for large k. This operation is known as *scalar multiplication* (where *scalar* refers to the fact that k is an integer, not a group element).

We can visualise the sequence P, P^2, P^3, … as repeatedly drawing a line through points P and P^i, as shown in Figure 1, finding a third intersection point of this line, and mirroring it with respect to the x axis to obtain point P^{i+1}. The effect, intuitively speaking, is a sequence of points that "jump around" the curve in a complicated pattern that is difficult to predict, as illustrated in Figure 3. This complicated pattern is what makes the group suitable for cryptography. In particular, it is believed that the Decisional Diffie-Hellman assumption (see Section 2.4) is true in this group.

The best known algorithms for discrete logarithms in this group, such as Pollard's rho algorithm [28], boil down to essentially trying lots of values of k until we find a result that matches the input P^k. This algorithm takes approximately $O(\sqrt{k})$ time to find k. Thus, if we choose k to be n bits long, the time taken is $O(2^{n/2})$. X25519 chooses $n = 251$ (less than 255 because four bits are set to constant values), making the difficulty of computing the discrete logarithm similar to the difficulty of breaking a 128-bit symmetric cipher.

Listing 4 shows how to implement Diffie-Hellman. The `scalarmult(out, scalar, point)` function takes three arguments: `point` is the input group element P, `scalar` is the scalar exponent, and `out` is a pointer to memory where the output P^{scalar} will be written. `point`, `scalar` and `out` are all 255-bit numbers, encoded as arrays of 32 bytes. In fact, `point` and `out` are not full group elements, but only the x coordinate of points on the curve; we will see in Section 4.4 why we can leave out the y coordinate. We will see the implementation of `scalarmult` in Listing 5.

`scalarmult_base(out, scalar)` performs scalar multiplication using a fixed group element `_9`, which is a curve point whose x coordinate equals 9. This point is chosen because its order is a large prime, as explained in Section 2.7. Listing 3 shows how we can check the order of this point using SageMath. The result is again returned in `out`.

`generate_keypair(pk, sk)` generates a new keypair, where the private key is written to `sk` and the public key is written to `pk`. The private key consists of 32 bytes (256 bits) drawn from a secure source of uniform random numbers. The public key is obtained from scalar multiplication of `sk` with the fixed base point in `scalarmult_base`.

`x25519(out, pk, sk)` is called by both the sender and the recipient of a message. If called by the sender, `pk` is the recipient's public key and `sk` is the random integer generated by the sender (Alice's k in the example in Section 2.3). If called by the recipient, `pk` is the group element sent along with the message (g^k in the example of Section 2.3) and `sk` is the recipient's private key. In either case, computing the scalar product of the group element and the secret integer produces the shared secret, which is written to `out`. The shared secret can then be used to initialise a symmetric cipher to encrypt the actual message, but this is beyond the scope of X25519, so we do not discuss it further in this paper.

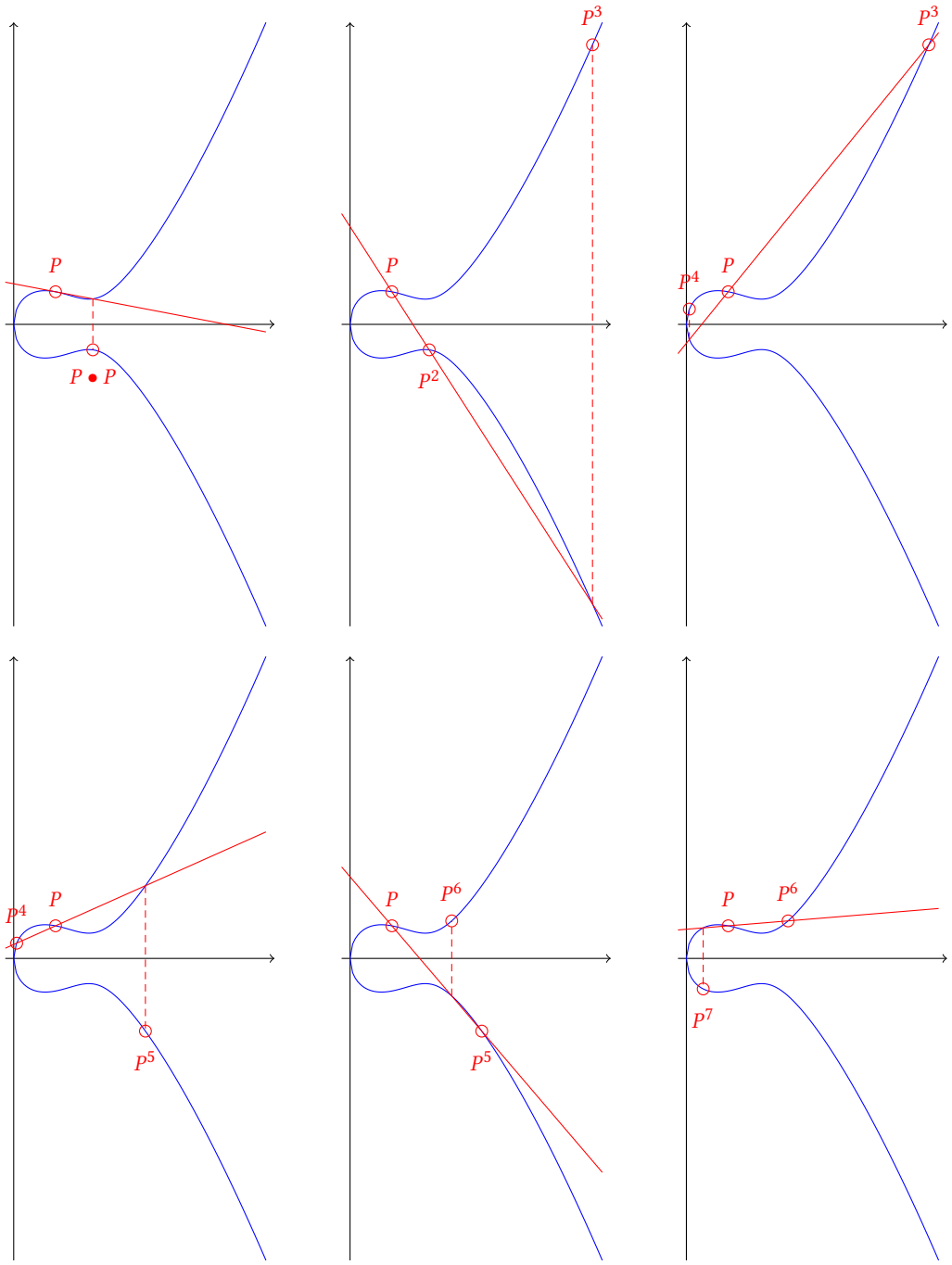

Fig. 3. Using the group operator to repeatedly combine a curve point P with itself. Each application of the operator can be visualised as drawing a straight line through P and P^i, finding the third point where this line intersects the curve, and mirroring that point with respect to the x axis to obtain P^{i+1}. The result is a sequence of points that "jump around" in a pseudorandom manner.

```c
typedef unsigned char u8;
typedef unsigned long long u64;
extern void randombytes(u8 *, u64);
static const u8 _9[32] = {9};

void scalarmult_base(u8 *out, const u8 *scalar)
{
  scalarmult(out, scalar, _9);
}

void generate_keypair(u8 *pk, u8 *sk)
{
  randombytes(sk, 32);
  scalarmult_base(pk, sk);
}

void x25519(u8 *out, const u8 *pk, const u8 *sk)
{
  scalarmult(out, sk, pk);
}
```

Listing 4. Using the `scalarmult` function to implement Diffie-Hellman.

4.4 The Montgomery ladder

The Montgomery ladder [6, 14, 27] is the algorithm used by X25519 to efficiently perform scalar multiplication. The function $L(P, i)$ takes a group element $P \neq \infty$ and a non-negative integer i, and it returns a pair of group elements (P^i, P^{i+1}) computed using the group operator $\bullet$:

$$\begin{aligned} L(P,\ 0) &= (\infty, P) && && (23)\\ L(P,\ 2i) &= (P_i \bullet P_i,\ P_i \bullet P_{i+1}) && \text{where } L(P,i) = (P_i, P_{i+1}) \text{ and } i > 0 \\ L(P,\ 2i+1) &= (P_i \bullet P_{i+1},\ P_{i+1} \bullet P_{i+1}) && \text{where } L(P,i) = (P_i, P_{i+1}) \text{ and } i \geq 0 \end{aligned}$$

Here $P^0 = \infty$ because ∞ is the identity element of the group. The second argument is halved (rounding down, i.e. shifting right by one bit) on each recursive call, resulting in $\lfloor \log_2 i \rfloor + 1$ recursive calls to compute $L(P, i)$ (i.e. one call per bit of i). In practice, we can use a loop instead of recursion, and in each iteration of the loop we examine one bit of i, starting with the most significant bit. (Listing 5 on page 28 shows our implementation of the Montgomery ladder; lines 19 to 43 contain the main loop that iterates over the bits of the integer.) At each loop iteration we perform two group operations:

- If the bit is zero (i.e. the $2i$ case), we combine P_i with itself to produce $P^i \bullet P^i = P^{2i}$, and we combine P_i with P_{i+1} to produce $P^i \bullet P^{i+1} = P^{2i+1}$.
- If the bit is one (i.e. the $2i + 1$ case), we combine P_i with P_{i+1} to produce $P^i \bullet P^{i+1} = P^{2i+1}$, and we combine P_{i+1} with itself to produce $P^{i+1} \bullet P^{i+1} = P^{2i+2}$.

The reason we return two group elements rather than just one is that, as we shall see shortly, there is a particularly efficient formula for computing $P^i \bullet P^{i+1} = P^{2i+1}$ given P, P^i and P^{i+1}, which is faster than directly using the group law in (18), (19).

In X25519 we need to execute this loop 255 times (the integer in the scalar multiplication function is 256 bits, but we set the topmost bit to always be zero, hence only 255 iterations are needed). As this is the most time-consuming part of the algorithm, we now examine how to perform these group operations as fast as possible.

Using projective coordinates. First of all, notice that equations (18) and (19) contain fractions. If we have to perform division on each iteration of the loop, the multiplicative inverse operation would become the slowest part of the algorithm. To save time, we can represent coordinates as fractions: $x = X/Z$ and $y = Y/Z$, where X, Y and Z are integers. (These are known as *projective coordinates*, whereas the (x, y) coordinates we have been using so far are called *affine coordinates*.) At each loop iteration we calculate the numerator and denominator for the coordinates separately, without dividing, and we perform the actual division only once, at the end after we have finished the loop.

Projective formulas for point doubling. Let's first derive optimised formulas for combining a group element with itself: $P_{2i} = (x_{2i}, y_{2i}) = P_i \bullet P_i = (x, y) \bullet (x, y)$. Using the expression in (18):

$$\begin{aligned}
x_{2i} &= \frac{(3x^2 + 2Ax + 1)^2}{4y^2} - A - 2x \\
&= \frac{(3x^2 + 2Ax + 1)^2}{4\,(x^3 + Ax^2 + x)} - A - 2x \qquad \text{since } (x, y) \text{ is on the curve } y^2 = x^3 + Ax^2 + x \\
&= \frac{(3x^2 + 2Ax + 1)^2 - 4\,(x^3 + Ax^2 + x)(2x + A)}{4\,(x^3 + Ax^2 + x)} \\
&= \frac{9x^4 + 12Ax^3 + (4A^2 + 6)\,x^2 + 4Ax + 1 - 4\,(2x^4 + 3Ax^3 + (A^2 + 2)\,x^2 + Ax)}{4\,(x^3 + Ax^2 + x)} \\
&= \frac{x^4 - 2x^2 + 1}{4\,(x^3 + Ax^2 + x)} = \frac{\frac{X^4}{Z^4} - \frac{2X^2}{Z^2} + 1}{4\left(\frac{X^3}{Z^3} + \frac{AX^2}{Z^2} + \frac{X}{Z}\right)} \qquad \text{using projective coordinates } x = \frac{X}{Z} \qquad (24) \\
&= \frac{\frac{1}{Z^4}\,(X^4 - 2X^2Z^2 + Z^4)}{\frac{4}{Z^3}\,(X^3 + AX^2Z + XZ^2)} = \frac{X^4 - 2X^2Z^2 + Z^4}{4XZ\,(X^2 + AXZ + Z^2)} = \frac{(X^2 - Z^2)^2}{4XZ\,(X^2 + AXZ + Z^2)}
\end{aligned}$$

Thus, we obtain the following projective formulas for point doubling:

$$\left(\frac{X_i}{Z_i}, \frac{Y_i}{Z_i}\right) \bullet \left(\frac{X_i}{Z_i}, \frac{Y_i}{Z_i}\right) = \left(\frac{X_{2i}}{Z_{2i}}, \frac{Y_{2i}}{Z_{2i}}\right) \quad \text{where} \quad \begin{aligned} X_{2i} &= (X_i^2 - Z_i^2)^2 \\ Z_{2i} &= 4X_iZ_i\,(X_i^2 + AX_iZ_i + Z_i^2) \end{aligned} \qquad (25)$$

Note that that the formulas for X_{2i} and Z_{2i} do not use the Y coordinate anywhere. This means that we can avoid computing any Y coordinates in the first place, and we don't even need to derive an expression for Y_{2i}. Next, we will derive formulas for point addition that also avoid using any Y coordinates.

Projective formulas for point addition. We now derive formulas for the group operation $P_{2i+1} = (x_{2i+1}, y_{2i+1}) = P_i \bullet P_{i+1} = (x_1, y_1) \bullet (x_2, y_2)$. We assume that $P_i \neq P_{i+1}$ (because we know that $P_{i+1} = P^{i+1} = P^i \bullet P = P_i \bullet P$ and we are assuming that $P \neq \infty$), and we also assume that $P_i^{-1} \neq P_{i+1}$ (in this case, $P_{2i+1} = \infty$). Therefore we can assume $x_1 \neq x_2$. Taking equation (18) as our starting point:

$$
\begin{aligned}
x_{2i+1} &= \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - A - x_1 - x_2 \\
&= \frac{y_2^2 - 2y_1y_2 + y_1^2 - (A + x_1 + x_2)(x_2 - x_1)^2}{(x_2 - x_1)^2} \\
&= \frac{x_2^3 + Ax_2^2 + x_2 + x_1^3 + Ax_1^2 + x_1 - 2y_1y_2}{(x_2 - x_1)^2} \\
&\quad - \frac{Ax_2^2 - 2Ax_1x_2 + Ax_1^2 + x_1x_2^2 - 2x_1^2x_2 + x_1^3 + x_2^3 - 2x_1x_2^2 + x_1^2x_2}{(x_2 - x_1)^2} \\
&= \frac{x_1 + x_2 + x_1^2x_2 + x_1x_2^2 + 2Ax_1x_2 - 2y_1y_2}{(x_2 - x_1)^2} \\
&= \frac{(x_1 + x_2)(1 + x_1x_2) + 2Ax_1x_2 - 2y_1y_2}{(x_2 - x_1)^2} \qquad (26)
\end{aligned}
$$

Next, we use the fact that $P_{i+1} = P_i \bullet P$. Because every element of the group has an inverse, we have $P_i^{-1} \bullet P_{i+1} = P_i^{-1} \bullet P_i \bullet P = P$. Let $P = (x_P, y_P)$, $P_i = (x_1, y_1)$, and $P_{i+1} = (x_2, y_2)$. Because $P_i^{-1} = (x_1, -y_1)$ we have

$$
P = (x_P, y_P) = P_i^{-1} \bullet P_{i+1} = (x_1, -y_1) \bullet (x_2, y_2) \quad \Longrightarrow \quad x_P = \frac{(x_1 + x_2)(1 + x_1x_2) + 2Ax_1x_2 + 2y_1y_2}{(x_2 - x_1)^2}
$$

That is, x_P equals the expression (26) with y_1 inverted. To clear the remaining occurrences of y_1 and y_2 from (26), without introducing square roots, we multiply x_{2i+1} and x_P (derivation from [6]):

$$
\begin{aligned}
x_P x_{2i+1} (x_2 - x_1)^4 &= ((x_1 + x_2)(1 + x_1x_2) + 2Ax_1x_2 + 2y_1y_2)((x_1 + x_2)(1 + x_1x_2) + 2Ax_1x_2 - 2y_1y_2) \\
&= ((x_1 + x_2)(1 + x_1x_2) + 2Ax_1x_2)^2 - (2y_1y_2)^2 \\
&= ((x_1 + x_2)(1 + x_1x_2) + 2Ax_1x_2)^2 - 4(x_1^3 + Ax_1^2 + x_1)(x_2^3 + Ax_2^2 + x_2) \\
&= (x_1 + x_2)^2(1 + x_1x_2)^2 + 4Ax_1x_2(x_1 + x_2)(1 + x_1x_2) + 4A^2x_1^2x_2^2 \\
&\quad - 4(x_1^3 + x_1)(x_2^3 + x_2) - 4Ax_1^2(x_2^3 + x_2) - 4Ax_2^2(x_1^3 + x_1) - 4A^2x_1^2x_2^2 \\
&= (x_1 + x_2)^2(1 + x_1x_2)^2 + 4Ax_1x_2(x_1 + x_2 + x_1^2x_2 + x_1x_2^2) \\
&\quad - 4(x_1^3 + x_1)(x_2^3 + x_2) - 4Ax_1x_2(x_1x_2^2 + x_1 + x_1^2x_2 + x_2) \\
&= (x_1 + x_2)^2(1 + x_1x_2)^2 - 4(x_1^3 + x_1)(x_2^3 + x_2) \\
&= (x_1^2 + 2x_1x_2 + x_2^2)(1 + 2x_1x_2 + x_1^2x_2^2) - 4(x_1^3x_2^3 + x_1x_2^3 + x_1^3x_2 + x_1x_2) \\
&= x_1^2 + 2x_1x_2 + x_2^2 + 2x_1^3x_2 + 4x_1^2x_2^2 + 2x_1x_2^3 + x_1^4x_2^2 + 2x_1^3x_2^3 + x_1^2x_2^4 \\
&\quad - 4x_1^3x_2^3 - 4x_1x_2^3 - 4x_1^3x_2 - 4x_1x_2 \\
&= x_1^2 - 2x_1x_2 + x_2^2 - 2x_1^3x_2 + 4x_1^2x_2^2 - 2x_1x_2^3 + x_1^4x_2^2 - 2x_1^3x_2^3 + x_1^2x_2^4 \\
&= (x_2 - x_1)^2 - 2x_1x_2(x_2 - x_1)^2 + x_1^2x_2^2(x_2 - x_1)^2 \\
&= (x_2 - x_1)^2(x_1x_2 - 1)^2
\end{aligned}
$$

Hence, under the assumption that $x_P \neq 0$, we obtain

$$x_{2i+1} = \frac{(x_1x_2 - 1)^2}{x_P\,(x_2 - x_1)^2} = \frac{\left(\frac{X_1X_2}{Z_1Z_2} - 1\right)^2}{\frac{X_P}{Z_P}\left(\frac{X_2}{Z_2} - \frac{X_1}{Z_1}\right)^2} \quad \text{where} \quad x_P = \frac{X_P}{Z_P},\ x_1 = \frac{X_1}{Z_1},\ x_2 = \frac{X_2}{Z_2}$$

$$= \frac{Z_P\left(\frac{X_1^2X_2^2}{Z_1^2Z_2^2} - \frac{2X_1X_2}{Z_1Z_2} + 1\right)}{X_P\left(\frac{X_2^2}{Z_2^2} - \frac{2X_1X_2}{Z_1Z_2} + \frac{X_1^2}{Z_1^2}\right)} = \frac{\frac{Z_P}{Z_1^2Z_2^2}\left(X_1^2X_2^2 - 2X_1X_2Z_1Z_2 + Z_1^2Z_2^2\right)}{\frac{X_P}{Z_1^2Z_2^2}\left(X_2^2Z_1^2 - 2X_1X_2Z_1Z_2 + X_1^2Z_2^2\right)}$$

$$= \frac{Z_P\,(X_1X_2 - Z_1Z_2)^2}{X_P\,(X_1Z_2 - X_2Z_1)^2} \tag{27}$$

In X25519, the base point P of the scalar multiplication $x_P = X_P/Z_P$ is given as an affine coordinate (i.e. not as a fraction), and $P \neq \infty$ so we can assume $Z_P = 1$ and $X_P = x_P$.

Thus, we can compute one step of the Montgomery ladder using the formulas from (25) and (27):

$$\begin{aligned} X_{2i} &= (X_i^2 - Z_i^2)^2 & X_{2i+1} &= (X_iX_{i+1} - Z_iZ_{i+1})^2 \\ Z_{2i} &= 4X_iZ_i\,(X_i^2 + AX_iZ_i + Z_i^2) & Z_{2i+1} &= x_P\,(X_iZ_{i+1} - X_{i+1}Z_i)^2 \end{aligned} \tag{28}$$

These formulas never use the y coordinate, allowing us to operate on the x coordinate alone. They take as input one bit of the scalar, the output from the previous step $(X_i, Z_i, X_{i+1}, Z_{i+1})$, as well as the base point x coordinate x_P. If the current bit is zero, they produce $(X_{2i}, Z_{2i}, X_{2i+1}, Z_{2i+1})$ as output, as shown in (23). If the current bit is one, they produce $(X_{2i+1}, Z_{2i+1}, X_{2i+2}, Z_{2i+2})$ as output, where X_{2i+2} and Z_{2i+2} are computed by applying the doubling formulas to (X_{i+1}, Z_{i+1}) instead of (X_i, Z_i).

One desirable property of these formulas is that each step of the ladder performs exactly the same arithmetic operations, regardless of the input coordinates and the bits of the scalar, making the algorithm constant-time.

4.5 Handling the point at infinity

In the derivation of the above formulas we have so far considered only points that are solutions to the curve equation, and ignored the point at infinity ∞. It is time that we now address this issue.

The point at infinity cannot be represented in affine coordinates (x, y) for any finite x, y. However, a convenient feature of using projective coordinates is that we can represent the point at infinity as a fraction with a denominator of zero: we define the x coordinate of ∞ to be $\frac{X}{0}$, i.e. $Z = 0$. We do not allow X and Z to both be zero. We can ignore the y coordinate since our formulas do not use it.

Fortunately, our formulas (28) already handle the point at infinity correctly. We demonstrate this by showing that they produce the required result if any of their inputs are ∞. Moreover, we show that provided each input is valid, $(X, Z) \neq (0, 0)$, then the outputs will also be different from $(0, 0)$. The following assumes that $x_P \neq 0$.

- Let $Z_i = 0$ and $X_i \neq 0$, so $P_i = \infty$. Then $Z_{2i} = 0$ and $X_{2i} = X_i^4 \neq 0$, so $P_{2i} = P_i \bullet P_i = \infty \bullet \infty = \infty$ as required by (15).
- Let $Z_i \neq 0$. We consider two cases depending on the value of $X_i^3 + AX_i^2Z_i + X_iZ_i^2$:

(1) Assume that $X_i^3 + AX_i^2Z_i + X_iZ_i^2 = 0$; hence $Z_{2i} = 0$. To prove that $X_{2i} \neq 0$, suppose to the contrary that $X_{2i} = 0$; then $X_i^2 = Z_i^2$ so $Z_i = \pm X_i$. Substituting into $X_i^3 + AX_i^2Z_i + X_iZ_i^2 = 0$ yields $X_i^3 \pm AX_i^3 + X_i^3 = 0$, so

$AX_i^3 = \pm 2X_i^3$. Since $Z_i \neq 0$ we have $X_i \neq 0$, and hence $A = \pm 2$. However, this contradicts the assumption that $A^2 \neq 4$, stated at the beginning of Section 4. Therefore we have $(X_{2i}, Z_{2i}) \neq (0, 0)$ as required.

(2) Assume that $X_i^3 + AX_i^2Z_i + X_iZ_i^2 \neq 0$, which implies $X_i \neq 0$. Since $Z_i \neq 0$ we have $Z_{2i} \neq 0$, so we have $(X_{2i}, Z_{2i}) \neq (0, 0)$ as required.

- Let $Z_i \neq 0$, $Z_{i+1} \neq 0$, and P_i has the same x coordinate as P_{i+1}, i.e. $X_i/Z_i = X_{i+1}/Z_{i+1}$. This implies one of two situations: either $P_i = P_{i+1}$ or $P_i^{-1} = P_{i+1}$. The former is ruled out by our assumption that $P \neq \infty$, so we have $P_i^{-1} = P_{i+1}$ and require that $P_i \bullet P_{i+1} = \infty$ as per (16). $X_i/Z_i = X_{i+1}/Z_{i+1}$ implies that $X_iZ_{i+1} = X_{i+1}Z_i$, so $Z_{2i+1} = 0$ as required.

  Next, we need to show that $X_{2i+1} \neq 0$. Suppose to the contrary that $X_{2i+1} = 0$, implying that $X_iX_{i+1} - Z_iZ_{i+1} = 0$. Taken together with the fact $X_iZ_{i+1} = X_{i+1}Z_i$ that we showed earlier, we have $X_iX_{i+1} - Z_iZ_{i+1} - X_iZ_{i+1} + X_{i+1}Z_i = (X_i + Z_i)(X_{i+1} - Z_{i+1}) = 0$. We now have two cases:

  (1) If $X_i + Z_i = 0$ then $X_i = -Z_i$ so $X_i/Z_i = -1$. Using assumption $X_i/Z_i = X_{i+1}/Z_{i+1}$ we have $X_{i+1}/Z_{i+1} = x_{i+1} = -1$, so the affine x coordinate of P_{i+1} equals -1.

  (2) If $X_i + Z_i \neq 0$ then $X_{i+1} = Z_{i+1}$ so $X_{i+1}/Z_{i+1} = x_{i+1} = 1$, so the affine x coordinate of P_{i+1} equals 1.

  In both cases, we use the point doubling expression (24) to calculate the x coordinate of $P_{i+1} \bullet P_{i+1}$, which is $(x_{i+1}^4 - 2x_{i+1}^2 + 1)/(4x_{i+1}^3 + 4Ax_{i+1}^2 + 4x_{i+1}) = 0$ for $x_{i+1} = \pm 1$. Note that $P_{i+1} = P_i \bullet P$, so $P = P_i^{-1} \bullet P_{i+1} = P_{i+1} \bullet P_{i+1}$ due to our earlier observation that $P_i^{-1} = P_{i+1}$. Hence, the x coordinate of P equals zero, which contradicts our earlier assumption that $x_P \neq 0$. Thus we have $X_{2i+1} \neq 0$ as required.
- Let $Z_i \neq 0$, $Z_{i+1} \neq 0$, and P_i has a different x coordinate from P_{i+1}, i.e. $X_i/Z_i \neq X_{i+1}/Z_{i+1}$. Then $X_iZ_{i+1} \neq X_{i+1}Z_i$ so $Z_{2i+1} \neq 0$, so $P_{2i+1} \neq \infty$ as required.
- Let $Z_i = 0$ and $X_i \neq 0$, so $P_i = \infty$. Since $P_{i+1} = P_i \bullet P = \infty \bullet P = P$ and $P \neq \infty$ we have $Z_{i+1} \neq 0$ and $X_{i+1}/Z_{i+1} = x_P$, so $X_{i+1} = x_PZ_{i+1}$. Hence, $X_{2i+1} = (X_iX_{i+1})^2 = x_P^2X_i^2Z_{i+1}^2$ and $Z_{2i+1} = x_P(X_iZ_{i+1})^2 \neq 0$. Thus, $X_{2i+1}/Z_{2i+1} = x_P^2X_i^2Z_{i+1}^2/x_PX_i^2Z_{i+1}^2 = x_P$ so $P_{2i+1} = P_i \bullet P_{i+1} = \infty \bullet P = P$ as required by (15).
- Let $Z_{i+1} = 0$ and $X_{i+1} \neq 0$, so $P_{i+1} = \infty$. Since $P_{i+1} = P_i \bullet P$ and $P \neq \infty$ we have $P_i = P^{-1}$ and $Z_i \neq 0$. The x coordinate of P^{-1} is the same as the x coordinate of P, namely x_P, so $X_i/Z_i = x_P$, so $X_i = x_PZ_i$. Hence, similarly to the last case, we have $X_{2i+1} = (X_iX_{i+1})^2 = x_P^2X_{i+1}^2Z_i^2$ and $Z_{2i+1} = x_P(-X_{i+1}Z_i)^2 \neq 0$. Thus, $X_{2i+1}/Z_{2i+1} = x_P^2X_{i+1}^2Z_i^2/x_PX_{i+1}^2Z_i^2 = x_P$, which is consistent with $P_{2i+1} = P_i \bullet P_{i+1} = P^{-1} \bullet \infty = P^{-1}$ as required by (15).

These bullet points cover all possible cases, demonstrating that the formulas (28) correctly handle all group elements, including the point at infinity, without need for any special handling of edge cases. That is good news because the lack of edge cases simplifies the implementation of the Montgomery ladder. Moreover, it is easier to make the algorithm constant-time if every step of the ladder performs exactly the same arithmetic operations, independently of the values of its inputs.

4.6 Optimising the Montgomery ladder step

The formulas (28) are nice and simple, but if we compare them to the implementation of the Montgomery ladder in Listing 5, the two look quite different. The reason is that the code is based on formulas that have been further optimised.

We have already avoided using division in the Montgomery ladder step by moving to projective coordinates. To further improve the performance, we will aim to reduce the number of finite field multiplications as far as possible,

since they are generally the most expensive operations after division. We ignore additions and subtractions since they are cheap by comparison.

If we break down the formulas (28) into one multiplication per equation, reusing common sub-expressions where possible, we see that each step of the ladder requires 14 multiplications:

$$
\begin{aligned}
v_1 &= X_i^2 & v_5 &= A v_3 & v_6 &= X_i X_{i+1} & v_{10} &= (v_8 - v_9)^2 \\
v_2 &= Z_i^2 & X_{2i} &= (v_1 - v_2)^2 & v_7 &= Z_i Z_{i+1} & X_{2i+1} &= (v_6 - v_7)^2 \\
v_3 &= X_i Z_i & Z_{2i} &= v_4 \, (v_1 + v_5 + v_2) & v_8 &= X_i Z_{i+1} & Z_{2i+1} &= x_P v_{10} \\
v_4 &= 4 v_3 & & & v_9 &= X_{i+1} Z_i & &
\end{aligned}
$$

Some authors count multiplication by a constant (4 or A, in the case of v_4 and v_5) and squaring separately from multiplication. However, our implementation uses the same multiplication function in all cases, so for simplicity we count all types of multiplication equally.

In contrast to the above 14 multiplications, the implementation in Listing 5 uses only 10 multiplications per step. In this section we focus on how to derive this algorithm from the formulas (28). The main ladder loop in lines 19 to 43 of Listing 5 performs the following operations:

- `bit = (clamped[i >> 3] >> (i & 7)) & 1` sets `bit` to be the `i`th bit from the little-endian byte array `clamped` (which was previously set to be a copy of the parameter `scalar`, with a few tweaks explained later).
- `swap25519(a, b, bit)` examines `bit`, which is either 0 or 1. If `bit == 0`, the function does nothing. If `bit == 1`, the function swaps the values in the two variables `a` and `b`. It does this in constant time, so the "do nothing" case takes the same execution time as the swapping case.
- `fadd`, `fsub`, and `fmul` perform field element addition, subtraction, and multiplication, as defined in Section 3.

Each iteration of the loop takes as input the values in variables `a`, `b`, `c`, `d`, `x`, and `clamped`, as well as the constant `_121665 = {0xDB41,1}`, which contains the number `0x1DB41` = 121665 (split into 16-bit chunks using the `field_elem` representation). As output it writes new values to the variables `a`, `b`, `c`, and `d`. Moreover, it uses `e` and `f` as temporary variables. In the code of Listing 5, variables are reused. For better readability, we give a new name to each variable assignment in the following breakdown of the operations. The 18 arithmetic operations in the Montgomery ladder loop (10 multiplications/squarings and 8 additions/subtractions) compute the following expressions:

```
typedef long long i64;
typedef i64 field_elem[16];
static const field_elem _121665 = {0xDB41, 1};

void scalarmult(u8 *out, const u8 *scalar, const u8 *point)
{
  u8 clamped[32];
  i64 bit, i;
  field_elem a, b, c, d, e, f, x;
  for (i = 0; i < 32; ++i) clamped[i] = scalar[i];
  clamped[0] &= 0xf8;
  clamped[31] = (clamped[31] & 0x7f) | 0x40;
  unpack25519(x, point);
  for (i = 0; i < 16; ++i) {
    b[i] = x[i];
    d[i] = a[i] = c[i] = 0;
  }
  a[0] = d[0] = 1;
  for (i = 254; i >= 0; --i) {
    bit = (clamped[i >> 3] >> (i & 7)) & 1;
    swap25519(a, b, bit);
    swap25519(c, d, bit);
    fadd(e, a, c);
    fsub(a, a, c);
    fadd(c, b, d);
    fsub(b, b, d);
    fmul(d, e, e);
    fmul(f, a, a);
    fmul(a, c, a);
    fmul(c, b, e);
    fadd(e, a, c);
    fsub(a, a, c);
    fmul(b, a, a);
    fsub(c, d, f);
    fmul(a, c, _121665);
    fadd(a, a, d);
    fmul(c, c, a);
    fmul(a, d, f);
    fmul(d, b, x);
    fmul(b, e, e);
    swap25519(a, b, bit);
    swap25519(c, d, bit);
  }
  finverse(c, c);
  fmul(a, a, c);
  pack25519(out, a);
}
```

Listing 5. The Montgomery ladder for scalar multiplication.

`fadd(e, a, c);`	$v_1 = a + c$	
`fsub(a, a, c);`	$v_2 = a - c$	
`fadd(c, b, d);`	$v_3 = b + d$	
`fsub(b, b, d);`	$v_4 = b - d$	
`fmul(d, e, e);`	$v_5 = v_1^2$	$= (a+c)^2$
`fmul(f, a, a);`	$v_6 = v_2^2$	$= (a-c)^2$
`fmul(a, c, a);`	$v_7 = v_3 \cdot v_2$	$= (b+d)(a-c) = ab - bc + ad - cd$
`fmul(c, b, e);`	$v_8 = v_4 \cdot v_1$	$= (b-d)(a+c) = ab + bc - ad - cd$
`fadd(e, a, c);`	$v_9 = v_7 + v_8$	$= 2(ab - cd)$
`fsub(a, a, c);`	$v_{10} = v_7 - v_8$	$= 2(ad - bc)$
`fmul(b, a, a);`	$v_{11} = v_{10}^2$	$= 4(ad - bc)^2$
`fsub(c, d, f);`	$v_{12} = v_5 - v_6$	$= (a+c)^2 - (a-c)^2 = a^2 + 2ac + c^2 - a^2 + 2ac - c^2 = 4ac$
`fmul(a, c, _121665);`	$v_{13} = 121665 \cdot v_{12}$	$= 486660\,ac = (A-2)\,ac$
`fadd(a, a, d);`	$v_{14} = v_{13} + v_5$	$= (A-2)\,ac + a^2 + 2ac + c^2 = a^2 + Aac + c^2$
`fmul(c, c, a);`	$v_{15} = v_{12} \cdot v_{14}$	$= 4ac\,(a^2 + Aac + c^2)$
`fmul(a, d, f);`	$v_{16} = v_5 \cdot v_6$	$= (a+c)^2 (a-c)^2 = (a^2 + 2ac + c^2)(a^2 - 2ac + c^2)$
		$= a^4 + 2a^3c + a^2c^2 - 2a^3c - 4a^2c^2 - 2ac^3 + a^2c^2 + 2ac^3 + c^4$
		$= a^4 - 2a^2c^2 + c^4 = (a^2 - c^2)^2$
`fmul(d, b, x);`	$v_{17} = v_{11} \cdot x$	$= 4x\,(ad - bc)^2$
`fmul(b, e, e);`	$v_{18} = v_9^2$	$= 4(ab - cd)^2$

Let $a = X_i$, $b = X_{i+1}$, $c = Z_i$, $d = Z_{i+1}$, and $x = x_P$ at the start of a loop iteration. Further let `bit` be 0, so the `swap25519` operations have no effect. Then the expressions above match the equations (28) with $A = 486662$, $v_{16} = X_{2i}$, $v_{18} = 4X_{2i+1}$, $v_{15} = Z_{2i}$, and $v_{17} = 4Z_{2i+1}$. The values $(v_{16}, v_{18}, v_{15}, v_{17})$ are written to variables `a`, `b`, `c`, and `d` respectively, forming the input to the next iteration.

If `bit` is 1, the values in `a` and `b` are swapped before and after the computation of these expressions, and likewise the values in `c` and `d` are swapped. Thus, the inputs to the computation are $a = X_{i+1}$, $b = X_i$, $c = Z_{i+1}$, and $d = Z_i$, and the outputs are $v_{16} = X_{2i+2}$, $v_{18} = 4X_{2i+1}$, $v_{15} = Z_{2i+2}$, and $v_{17} = 4Z_{2i+1}$. After the final swaps, the variables `a`, `b`, `c`, and `d` contain the values $(v_{18}, v_{16}, v_{17}, v_{15})$.

Thus, each loop iteration maps the input tuple $(X_i, Z_i, X_{i+1}, Z_{i+1})$ to either the output tuple $(X_{2i}, Z_{2i}, 4X_{2i+1}, 4Z_{2i+1})$ or the output tuple $(4X_{2i+1}, 4Z_{2i+1}, X_{2i+2}, Z_{2i+2})$ depending on the value of `bit`. This exactly matches the Montgomery ladder step (23), except for the additional factor of 4 in X_{2i+1} and Z_{2i+1}. However, since these two variables are just an expanded representation of the fraction $x_{2i+1} = X_{2i+1}/Z_{2i+1}$, these two factors of 4 cancel out and have no effect on the final result.

4.7 Clamping

We now turn to the first few lines of the `scalarmult` function.

The scalar parameter `scalar` is assumed to be a uniformly distributed random array of 32 bytes. First, `scalar` is copied to `clamped`, and then five bits of `clamped` are set to constant values. `clamped[0] &= 0xf8` sets the three least significant bits to 0. `clamped[31] = (clamped[31] & 0x7f) | 0x40` sets the most significant bit to 0, and the second-most-significant bit to 1. This process is known as *clamping* [25].

Setting the three least significant bits to 0 ensures that `clamped` is a multiple of 8. By also setting the most significant bit to 0, `clamped` becomes a number of the form hk, where $h = 8$ and k is a uniformly distributed random number from the range $[0,\ 2^{252} - 1]$. As explained in Section 2.7, making `clamped` a multiple of the cofactor $h = 8$ prevents small subgroup confinement attacks. The upper end of the range, $2^{252} - 1$, is slightly below $q = 2^{252} + 27742317777372353535851937790883648493$, the order of the base point/generator. Choosing k from the range $[0,\ 2^{252} - 1]$ is, in practice, equivalent to using the range $[0,\ q - 1]$: when using the latter range, the probability of picking a value in the range $[2^{252}, q - 1]$ is approximately 10^{-38}.

The reason for setting the second-most-significant bit to 1 is unrelated to the cofactor: it is instead a precaution to help ensure constant-time implementations. If this bit was zero, then an implementation of scalar multiplication could save the first iteration of the Montgomery ladder without affecting the result; an adversary could then use this timing variation to leak the most significant bit of the private key (and perhaps more). Setting this bit to one forces the Montgomery ladder to always use the full 255 iterations. Our implementation would be constant-time even without setting this bit to 1, but X25519 is defined to always have this bit set as a precaution to protect less careful implementations.

After clamping, `clamped` is a number of the form $8k$, where $k \in [2^{251},\ 2^{252} - 1]$. Thus, the distribution of group elements produced by X25519 is non-uniform: only about half of the group elements in the subgroup of order q will be generated. However, this non-uniformity does not significantly weaken the security of X25519 for Diffie-Hellman purposes (apart from the loss of 1 bit of entropy).

A final reason for clamping: if $k = 0$ or k is a multiple of q, then we would expect $P^k = \infty$ for a base point P of order q. Although the Montgomery ladder correctly handles the point at infinity as an intermediate value, as shown in Section 4.5, `scalarmult` does not have the ability to return the point at infinity, since the function only returns an x coordinate of a curve point. If there was no clamping and a scalar argument of 0 or q were passed in, `scalarmult` would return zero, which the caller cannot distinguish from the curve point whose x coordinate is zero. Returning an error in this case would make the function non-constant-time, if not done carefully.

By forcing k to be within $0 < k < q$, clamping avoids ever needing to return ∞ (except in the case where an adversary provides a group element with small order, as discussed in Section 2.7, in which case it's fine to return zero). In the course of Diffie-Hellman with a base point of prime order q, ∞ will never be generated, since $(P^j)^k = P^{jk} = \infty$ only if jk is a multiple of q, which is only possible if j or k is a multiple of q.

4.8 Finishing off scalar multiplication

At the start of `scalarmult`, the x coordinate of the base point is passed in as parameter `point`. On line 13 this byte array is translated into `field_elem` representation and copied to `x`. We then initialise `a` = 1, `b` = `x`, `c` = 0, and `d` = 1. This gives us the starting state of the Montgomery ladder as defined in equation (23): $L(P, 0) = (\frac{a}{c}, \frac{b}{d}) = (\infty, P)$.

After the Montgomery ladder is finished, `a` and `c` contain the numerator and denominator of the x coordinate of curve point $P^{clamped}$. In order to return this value, we must first convert projective coordinates back into affine coordinates by dividing the two, as discussed in Section 3.3: we use the `finverse` function to replace `c` with its multiplicative inverse, and then multiply `a` and `c` using `fmul`.

Finally, we use the `pack25519` function, described in Section 3.4, to convert the `field_elem` representation of the result back into an array of 32 bytes, and we write the result to the output variable `out`. This value can now be used as a public key (if `point` was the base point), or as shared secret to initialise a symmetric encryption scheme (if `point` was a public key or the group element received from the other user), as shown in Section 4.3.

5 CONCLUSIONS

This paper has shown how to derive the X25519 implementation in TweetNaCl, line by line, from first principles. Starting by assuming only minimal mathematical background knowledge we have explored how modern, constant-time cryptography is implemented, and justified the correctness of this implementation. Although we have only studied one particular implementation, other implementations such as libsodium share many principles with the code studied here; after reading this paper, you will find it much easier to figure out what other implementations are doing.

For future work it would be interesting to expand this discussion to other common algorithms in elliptic curve cryptography, such as the Ed25519 signature scheme, and other curves such as NIST curves and the secp256k1 curve used by Bitcoin. The Ristretto255 prime-order group [16], which eliminates the cofactor of Curve25519, would also be interesting to discuss. I believe that by carefully studying existing implementations of cryptographic algorithms, and by analysing in detail what makes them correct, we can gain a deeper understanding of cryptography in general and improve the quality of implementations of cryptographic protocols.

ACKNOWLEDGMENTS

Thank you to Alastair Beresford, Daniel Hugenroth, Markus Kuhn, and Alex Mason for feedback on a draft of this paper. I am grateful for financial support from a Leverhulme Trust Early Career Fellowship, the Isaac Newton Trust, Nokia Bell Labs, and crowdfunding supporters including Ably, Adrià Arcarons, Chet Corcos, Macrometa, Mintter, David Pollak, RelationalAI, SoftwareMill, Talent Formation Network, and Adam Wiggins.

REFERENCES

[1] [n.d.]. HACL*, a high-assurance cryptographic library. https://hacl-star.github.io/

[2] Adrian Antipa, Daniel Brown, Alfred Menezes, René Struik, and Scott Vanstone. 2003. Validation of Elliptic Curve Public Keys. In *6th International Workshop on Practice and Theory in Public Key Cryptography (PKC 2003, Vol. LNCS, volume 2567)*. Springer, 211–223. https://doi.org/10.1007/3-540-36288-6_16

[3] Daniel J Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *9th International Conference on Theory and Practice in Public-Key Cryptography (PKC 2006)*. Springer, 207–228. https://doi.org/10.1007/11745853_14

[4] Daniel J Bernstein. 2009. Cryptography in NaCl. https://cr.yp.to/highspeed/naclcrypto-20090310.pdf

[5] Daniel J. Bernstein and Tanja Lange. [n.d.]. Montgomery curves. In *Explicit Formulas Database* (EFD). https://hyperelliptic.org/EFD/g1p/auto-montgom.html

[6] Daniel J Bernstein and Tanja Lange. 2017. Montgomery Curves and the Montgomery Ladder. In *Topics in Computational Number Theory Inspired by Peter L. Montgomery*. Cambridge University Press, 82–115. https://doi.org/10.1017/9781316271575.005

[7] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. [n.d.]. NaCl: Networking and Cryptography library. https://nacl.cr.yp.to/

[8] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. 2012. The Security Impact of a New Cryptographic Library. In *2nd International Conference on Cryptology and Information Security in Latin America (LATINCRYPT 2012)*. Springer, 159–176. https://doi.org/10.1007/978-3-642-33481-8_9

[9] Daniel J Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. 2014. TweetNaCl: A Crypto Library in 100 Tweets. In *3rd International Conference on Cryptology and Information Security in Latin America (LATINCRYPT 2014)*. Springer, 64–83.

https://doi.org/10.1007/978-3-319-16295-9_4
[10] Daniel J Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, and Sjaak Smetsers. 2014. TweetNaCl: A Crypto Library in 100 Tweets. https://tweetnacl.cr.yp.to/
[11] Ian Blake, Gadiel Seroussi, and Nigel Smart. 1999. *Elliptic Curves in Cryptography*. Cambridge University Press.
[12] Dmitry Chestnykh, Devi Mandiri, and AndSDev. [n.d.]. TweetNaCl.js. https://github.com/dchest/tweetnacl-js
[13] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. 2006. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press.
[14] Craig Costello and Benjamin Smith. 2018. Montgomery curves and their arithmetic. *Journal of Cryptographic Engineering* 8 (2018), 227–240. Issue 3. https://doi.org/10.1007/s13389-017-0157-6
[15] Cure53, Mario Heiderich, Jonas Magazinius, and Joachim Strömbergson. 2017. Review-Report Crypto Library TweetNaCl-js. https://cure53.de/tweetnacl.pdf
[16] Henry de Valence, Jack Grigg, George Tankersley, Filippo Valsorda, and Isis Lovecruft. 2020. The ristretto255 Group. https://ietf.org/id/draft-irtf-cfrg-ristretto255-00.html IETF Internet-Draft.
[17] Stefan Friedl. 2017. An elementary proof of the group law for elliptic curves. *Groups Complexity Cryptology* 9, 2 (Oct. 2017), 117–123. https://doi.org/10.1515/gcc-2017-0010
[18] Kazuyuki Fujii and Hiroshi Oike. 2017. An Algebraic Proof of the Associative Law of Elliptic Curves. *Advances in Pure Mathematics* 7, 12 (Dec. 2017), 649–659. https://doi.org/10.4236/apm.2017.712040
[19] Mike Hamburg. 2015. Decaf: Eliminating Cofactors Through Point Compression. In *35th Annual Cryptology Conference (CRYPTO 2015)*. Springer, 705–723. https://doi.org/10.1007/978-3-662-47989-6_34
[20] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. 2004. *Guide to Elliptic Curve Cryptography*. Springer.
[21] Ianix.com. 2020. Things that use Curve25519. https://ianix.com/pub/curve25519-deployment.html
[22] Wesley Janssen. 2014. *Curve25519 in 18 tweets*. Bachelor's Thesis. Radboud University Nijmegen. http://www.cs.ru.nl/bachelors-theses/2014/Wesley_Janssen___4037332___Curve25519_in_18_tweets.pdf Archived at https://perma.cc/4HVB-D6BJ.
[23] Neal Koblitz. 1994. *A Course in Number Theory and Cryptography*. Springer.
[24] Adam Langley, Mike Hamburg, and Sean Turner. 2016. Elliptic Curves for Security. RFC7748. https://doi.org/10.17487/RFC7748
[25] Neil Madden. 2020. What's the Curve25519 clamping all about? https://neilmadden.blog/2020/05/28/whats-the-curve25519-clamping-all-about/ Archived at https://perma.cc/6QPY-GJLR.
[26] Moxie Marlinspike and Trevor Perrin. 2016. The X3DH Key Agreement Protocol. https://www.signal.org/docs/specifications/x3dh/ Archived at https://perma.cc/MSA4-DP4G.
[27] Peter L Montgomery. 1987. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Math. Comp.* 48, 177 (1987), 243–264. https://doi.org/10.1090/S0025-5718-1987-0866113-7
[28] J M Pollard. 1978. Monte Carlo Methods for Index Computation (mod p). *Math. Comp.* 32, 143 (July 1978), 918–924. https://doi.org/10.2307/2006496
[29] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 5869. https://doi.org/10.17487/RFC8446
[30] WhatsApp. 2017. WhatsApp Encryption Overview. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf Archived at https://perma.cc/QD7M-GPG5.
[31] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*. ACM, 1789–1806. https://doi.org/10.1145/3133956.3134043

A DIFFERENCES TO THE ORIGINAL TWEETNACL

The code listings in this paper are based on version 20140427 of Bernstein et al.'s TweetNaCl implementation [10]. I have made a number of changes in the interest of readability; these changes do not affect the functionality or the constant-time property of the code. This appendix details those changes.

- All functions that are not part of the X25519 implementation are omitted.
- All uses of the `FOR` and `sv` macros are replaced with their definitions.
- The `gf` type definition is renamed to `field_elem`.
- I added whitespace around operators to improve readability.
- In the `unpack25519` function the parameters `o` and `n` are renamed to `out` and `in`, respectively.
- The `car25519` function is renamed to `carry25519`, its parameter `o` is renamed to `elem`, and its local variable `c` is renamed to `carry`. Its loop body originally read:

```
o[i]+=(1LL<<16);
c=o[i]>>16;
o[(i+1)*(i<15)]+=c-1+37*(c-1)*(i==15);
o[i]-=c<<16;
```

The addition of 2^{16} to `o[i]` and the subsequent subtraction of 1 from `o[i+1]` cancel out and serve no apparent purpose, so I removed them. I hypothesise that perhaps the addition of 2^{16} was supposed to ensure that `o[i]` is non-negative, but this is not true because it is possible to have `o[i]` $< -2^{16}$, and it is not necessary because the code handles negative values correctly anyway. Dan Bernstein and Tanja Lange did not respond to my emails requesting a clarification of these lines of code. My simplified version of the loop body also replaces the hard-to-read "multiply-by-boolean" idiom with a simple if statement:

```
carry = elem[i] >> 16;
elem[i] -= carry << 16;
if (i < 15) elem[i + 1] += carry; else elem[0] += 38 * carry;
```

It is safe to perform a conditional branch on the value of `i` since it is not a secret (it always ranges from 0 to 15).

- The `A` function is renamed to `fadd`, and its parameter `o` is renamed to `out`.
- The `Z` function is renamed to `fsub`, and its parameter `o` is renamed to `out`.
- The `M` function is renamed to `fmul`, its parameter `o` is renamed to `out`, and its local variable `t` is renamed to `product`.
- I removed the `S` (square) function, which only called `M` anyway, and replaced it with calls to `fmul`.
- The `inv25519` function is renamed to `finverse`, its parameters `o` and `i` are renamed to `out` and `in` respectively, and its local variable `a` is renamed to `i` (since it is used as loop counter).
- The `sel25519` function is renamed to `swap25519`, and its parameter `b` is renamed to `bit`.
- In the `pack25519` function the parameters `o` and `n` are renamed to `out` and `in` respectively, and the local variable `b` is renamed to `carry`.
- The `crypto_scalarmult_base` function is renamed to `scalarmult_base`, and its parameters `q` and `n` are renamed to `out` and `scalar`, respectively.
- The `crypto_box_keypair` function is renamed to `generate_keypair`, and its parameters `y` and `x` are renamed to `pk` and `sk`, respectively.
- The `x25519` function is added as an alias of `scalarmult`.
- The `crypto_scalarmult` function is renamed to `scalarmult`, its parameters `q`, `n` and `p` are renamed to `out`, `scalar` and `point`, respectively, and its local variables `z` and `r` are renamed to `clamped` and `bit`, respectively. The first loop's upper bound is changed from 31 to 32 and the following two lines (which perform the clamping) are slightly refactored to make the code clearer without changing its behaviour. I have changed the variable `x`, which was originally an 80-element array of `i64`, to be of type `field_elem` (i.e. a 16-element array of `i64`) instead. The first 16 elements of `x` have the same function as they did originally (namely, the internal representation of the x coordinate of the input point). The original code, after completing the Montgomery ladder, copies the value of local variable `a` to indexes 16 . . . 31 of `x`, `c` to indexes 32 . . . 47, `b` to indexes 48 . . . 63, and `d` to indexes 64 . . . 79, and then performs the last three function calls (the inversion of `c`, the multiplication of `a` and `c`, and the `pack25519` of the result) on offsets of the variable `x`. The purpose of copying `a`, `b`, `c` and `d` to `x` is unclear, since nothing ever uses the fact that these values have been copied into one contiguous array. Perhaps it is a remnant of earlier debugging logic? Dan Bernstein and Tanja Lange did not respond to my emails requesting a

clarification of why this is happening. I therefore removed the copying into `x` and changed the last three function calls of `scalarmult` to operate directly on `a` and `c` instead.

I have run the modified code with the test vectors provided as part of the NaCl package [7], and checked that they compute the same result.