Curve25519 in 18 tweets

Radboud University Nijmegen

Bachelor's Thesis

Author: Wesley Janssen

Supervisor: Peter Schwabe

February 21, 2014

Contents

1 Introduction **2**

2 Background and the Problem **3**

2.1 Mathematical background 3

2.1.1 Finite Fields 3

2.1.2 Elliptic curves 4

2.1.3 Elliptic curves over $\mathbb{F}_p$ 6

2.1.4 Curve25519 7

2.2 Cryptographic background 8

2.2.1 Introduction 8

2.2.2 Diffie-Hellman 8

2.2.3 Elliptic curve Diffie-Hellman 9

2.2.4 Timing attacks and countermeasures 9

2.2.5 NaCl 10

2.2.6 Previous implementations 11

3 Implementation Details **12**

3.1 Representation of elements of $\mathbb{F}_{2^{255}-19}$ 12

3.2 Field arithmetic 12

3.3 Elliptic curve arithmetic 15

3.4 Packing and unpacking 17

4 Results **18**

A Program, seperated in tweets **21**

B Program, no layout short version **23**

C Program, shortest version **24**

1 Introduction

There is an average of 1-10 bugs per thousand lines of code in software of industrial standards [Cov][McC04]. That means that for a project with 20 million lines of code, there is an average of 20.000 to 200.000 bugs. If this project happens to be a security protocol, bugs can easily be (maliciously) exploited. An example is in [BBPV12] where a bug in a function of the OpenSSL implementation is found. Considering that money transfers, for example, rely on these protocols, where there are supposedly no bugs, we want to have our 'secure' protocols truly secure.

That is why this thesis's aims is to shrink the code size of an already existing cryptographic protocol. Specifically, there was a challenge proposed by cryptographer Matthew Green to fit an entire cryptographic library in 100 tweets, in C (1 tweet equals a maximum of 140 characters, so 14.000 characters in total). TweetNaCl is currently the only serious contender in this category. TweetNaCl implements the NaCl library (found at 2.2.5) in the smallest, still human-readable, form. Its paper can be found at `http://tweetnacl.cr.yp.to/index.html` and the entire 100 tweets here: `https://twitter.com/TweetNaCl`. This bachelor's thesis is a part of this paper, and implements one of the five 'core functions' of the NaCl library, namely the key-agreement protocol. NaCl uses curve25519 in its reference implementation, which is introduced here [Ber06]. This is used as a basis for my own adaptation of this protocol. The five core functions of NaCl were estimated to take up roughly the same amount of code, so the goal for this thesis is $100/5 = 20$ tweets! This is equal to $20 * 140 = 2800$ characters of code.

There have been other attempts to combine twitter's short-message constraints and cryptography, as seen for example in [Cyb] and [Twi]. These however, are no serious attempts at making a full cryptographic library and as such, do not provide the security TweetNaCl (and therefore this thesis' algorithm) give.

One of the main focusses is readability. Because there are only a handful of lines, one can easily check and verify every line in the entire protocol. To make it easier to read, some conciseness of code has to be sacrificed, while still being beneath the 20-tweet mark. Another goal is portability, meaning the implementation is processor-indepedent, and can be compiled and used on any computer.

In section 2 I explain some of the background information of key-agreement, as well as the mathematics behind it. In section 3 I talk about the actual implementation. I justify some of the choices I have made in order to get the code as small as possible, without sacrificing security or any of the aforementioned goals. In section 4 I briefly present and discuss the achieved results.

2 Background and the Problem

2.1 Mathematical background

2.1.1 Finite Fields

Elliptic curves in cryptography have first been proposed by [Mil86] and [Kob87]. To understand an elliptic curve defined over a finite field, we first need to get down to the basics of finite fields, fields in general, and groups.

A group is a set of elements, together with an operator, which combines any two elements of the group to form a third element. This operator has to satisfy four conditions, also known as the group axioms:

- Closure: all formed elements are also in the group.
- Associativity: $\forall a, b, c$: $(a \circ b) \circ c = a \circ (b \circ c)$, where $\circ$ is the operator used.
- Identity: there is an element Id, such that $\forall a$: $(a \circ \mathrm{Id} = a)$.
- Invertibility: every element a in the group has an inverse, such that $a \circ a^{-1} = \mathrm{Id}$.

If, in addition to these, it satisfies the commutativity equation $a \circ b = b \circ a$, it is called an abelian group. In the case of the set of integers combined with the $+$ operator, $\circ = +$, $\mathrm{Id} = 0$, and $a^{-1} = -a$, because $a + b = b + a$, this is also an abelian group.

A finite field $\mathbb{F}_p$, also called a Galois Field, used in elliptic curve arithmetic, is a field with a finite number of elements in it. The order p of the field is the number of elements in it. A field is an abelian group with addition and an extra operator, namely multiplication. Addition satisfies all conditions from abelian groups. Multiplication is closed, associative and commutative in the same way addition is. Additionally, it has a neutral element of its own, 1, where $1 \cdot a = a \cdot 1 = a$ for some element a. Every element, except for 0, has an inverse for multiplication as well, such that $a \cdot a^{-1} = a^{-1} \cdot a = 1$. Lastly, multiplication is distributive with respect to addition, i.e. $\forall a, b, c$: $(a \cdot (b + c) = a \cdot b + a \cdot c)$.

2.1.2 Elliptic curves

Let $\mathbb{F}$ be a field, and E a curve defined over this $\mathbb{F}$. $E(\mathbb{F})$ is the set of solutions to the following general equation:

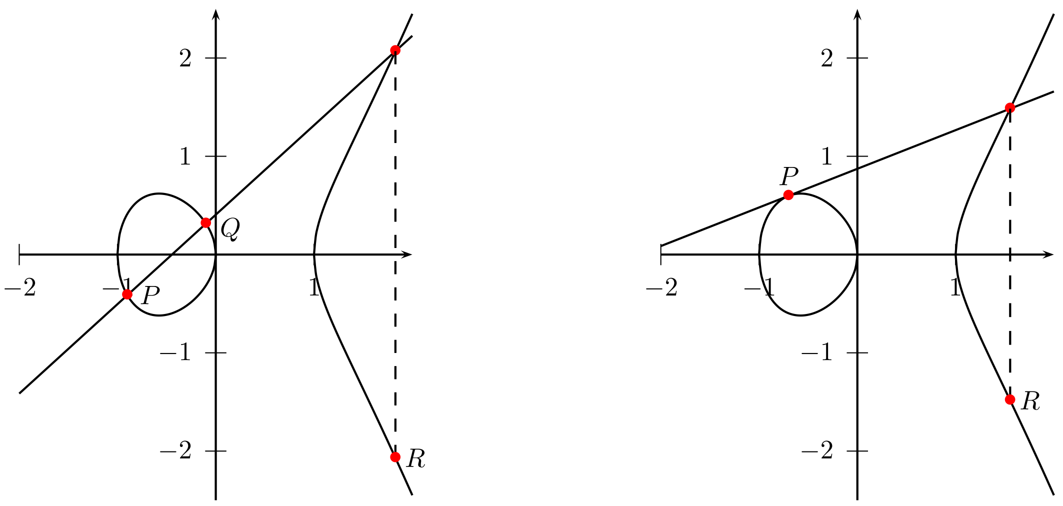
$$y^2 = x^3 + ax^2 + bx + c$$

Where $a, b, c \in \mathbb{F}$.

For us, Montgomery curves are more interesting because Montgomery curves are used in the NaCl-library. They generally use less code and are more resistant to timing attacks than normal elliptic curves [OKS00]. A Montgomery curve E is of the form:

$$E(\mathbb{F}_p) = \{\infty\} \cup \{(x, y) \in F_p : By^2 = x^3 + Ax^2 + x\}$$

where ∞ is the 'point at infinity' which also serves as the identity element, and $A =$ a (large) integer, with $A^2 - 4 \neq$ a square mod p. This function should have no cusps, 'sharp points', and no self-intersections. This is necessary because later on, we need to be able to take the derivative of the function, and a function with cusps has no unique derivative at such a cusp. We can now define addition on this curve, in order to do our encryption. Together with the point of infinity, the points now form an abelian group.



(a) Addition of points P and Q, resulting in R [1]

(b) Doubling of a point P, resulting in $R = 2P$[1]

Figure 1: $E : y^2 = x^3 - x$ over $\mathbb{R}$

[1]Taken from Peter Schwabe's `http://cryptojedi.org/misc/pstricks.shtml`.

We will start with the binary operator + which can most easily be described in words as follows: consider different points A and B we want to add and draw a line through these points. There now is exactly one more point of intersection on the curve. Calculate where this third point is, then reflect this point about the x-axis. An example can be seen in figure 1a. "Doubling" a point is done by taking the tangent line of a point A, then calculate the second point of intersection, then take the negation of the point. If the line is parallel to the y-axis, we take the identity-element ∞. Considering this, we can define + as:

1. $\infty + \infty = \infty$
2. $\infty + (x, y) = (x, y) + \infty = (x, y)$
3. $(x, y) + (x, -y) = \infty$
4. If $y \neq 0$, then $(x, y) + (x, y) = (x'', y'')$, where $x'' = \lambda - A - 2x = (x^2 - 1)^2/4y^2$, and $y'' = \lambda(x - x'') - y$. λ refers to the first derivate of E, being $\lambda = (3x^2 + 2Ax + 1)/2y$. $\triangleright$ (Doubling a point)
5. If $x \neq x'$, then $(x, y)+(x', y') = (x'', y'')$, where $x'' = \Delta^2 - A - x - x'$, and $y'' = \Delta(x - x'') - y$. Here, Δ is defined as $\Delta = (y' - y)/(x' - x)$, or in other words the slope of the function between points (x, y) and (x', y'). $\triangleright$ (Addition)

The addition of the identity element and some other element X will result in X, as seen in (1) and (2). Addition of a point and its negation will also result in the identity element (3). In (4), doubling a point happens by first computing the first derivative, then calculate the tangent line, and taking the negation of y. In the case that $y = 0$, $(x, 0) + (x, 0) = \infty$. (5) defines the 'normal' addition, we take the slope between two lines Δ, and compute the third point x''. If $x = x''$, then (4) will be in effect. In figure 1b we can see the doubling of a point, in figure 1a we can see addition.

2.1.3 Elliptic curves over $\mathbb{F}_p$

The above mentioned arithmetic are slow and not suitable for cryptographical purposes, because we need an infinite amount of points. We need something fast and precise, to obtain a finite group for the discrete logarithm problem. This is achieved by combining the elliptic curves and before-mentioned finite fields: elliptic curves over $\mathbb{F}_p$. This elliptic curve $E(\mathbb{F}_p)$ contains all points (x, y) which satisfy the elliptic curve equation, modulo p: $y^2 \bmod p = x^3 + Ax^2 + x \bmod p$.

As can be seen in figure 2, an elliptic curve defined over a finite field does not quite look like a normal elliptic curve anymore, though the points still form a group with the above-defined addition formulas. Because of this, a trivial geometric construction of addition and doubling as seen in figure 1 is not possible. However, the arithmetic is the same, but because we need to stay in $\mathbb{F}_p$, every operation will become modulo p.

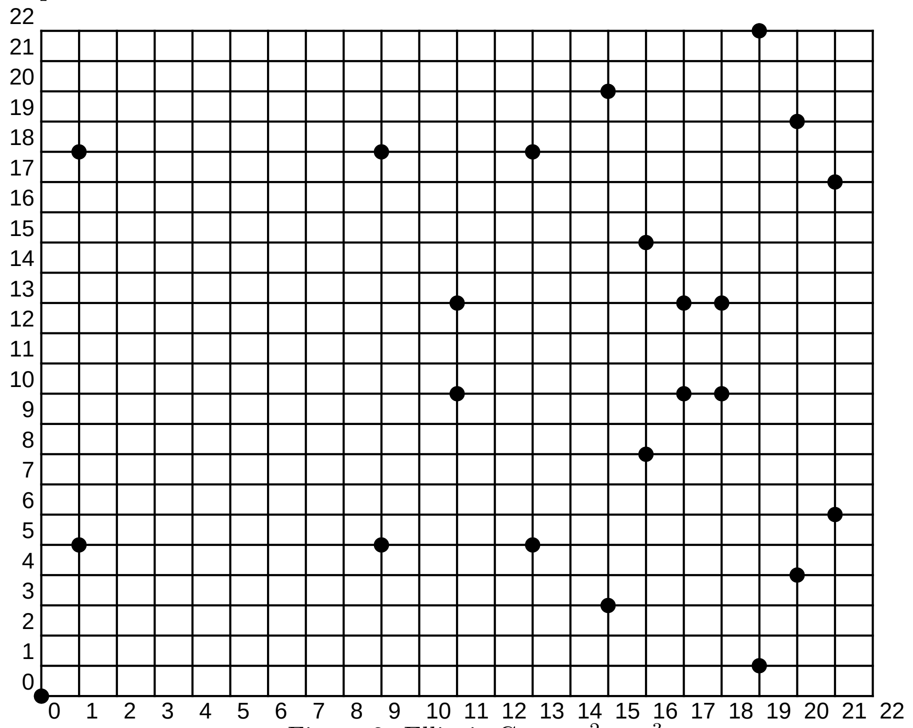


Figure 2: Elliptic Curve $y^2 = x^3 + x$

Because we have addition defined on an elliptic curve, we now want to be able to do point multiplication. This is done by repeatedly adding a point along the curve to itself. A neat property of this operation is that it is a one-way operation: it is considered easy to compute this multiplication, but considered hard to gain the initial point from the output. In order to compute $x_n = nP$, where x is the x-coordinate of P, and P a point on the curve $E(\mathbb{F}_p)$, we will use what is often called the Montgomery Ladder. In this algorithm each x-coordinate x_P of a point

P is represented as (X_P, Z_P), where $x_P = X_P/Z_P$.

Algorithm 2 shows the pseudo-code for one step in the Montgomery-function, the full algorithm is in algorithm 1.

Algorithm 1 Montgomery Ladder.

Input: a scalar $n = (n_0, \dots, n_t)$ and the x-coordinate x_P of point P on curve E

```
1: $X_1 = x_P, X_2 = 1, Z_2 = 0, X_3 = x_P, Z_3 = 1$
2: for $i \leftarrow t, 0$ do
3:     Swap if $n_i =$ even
4:     Perform one Montgomery Ladder Step
5:     Swap back if $n_i = 0$
6: end for
7: return $(X2, Z2)$
```

Algorithm 2 Montgomery Ladder Step.

$const_E = (A + 2)/4$, where A is the A from the curve equation

Input: X_1, X_P, Z_P, X_Q, Z_Q

```
1: $t_1 \leftarrow X_P + Z_P$
2: $t_2 \leftarrow X_P - Z_P$
3: $t_3 \leftarrow X_Q - Z_Q$
4: $t_4 \leftarrow X_Q + Z_Q$
5: $t_5 \leftarrow t_1^2 - t_2^2$
6: $t_6 \leftarrow t_1 \cdot t_4$
7: $t_7 \leftarrow t_2 \cdot t_3$
8: $X_{P+Q} \leftarrow (t_6 + t_7)^2$
9: $Z_{P+Q} \leftarrow X_1 \cdot (t_6 - t_7)^2$
10: $X_{2P} \leftarrow t_1^2 \cdot t_2^2$
11: $Z_{2P} \leftarrow t_5 \cdot (t_2^2 + const_E \cdot t_5)$
12: return $(X_{2P}, Z_{2P}, X_{P+Q}, Y_{P+Q})$
```

2.1.4 Curve25519

In [Ber06], Bernstein proposed the Curve25519 function for elliptic curve Diffie-Hellman key agreement. This function uses the previously mentioned arithmetic on a curve $E : y^2 = x^3 + Ax^2 + x$ over the field $\mathbb{F}_p$, where $A = 486662$ and $p = 2^{255} - 19$. These parameters are all carefully chosen for high-security high-speed reasons. See [Ber06] for a discussion of the security properties.

The function takes two 32-byte strings. One represents the x-coordinate of a point P and the other represents a 256-bit scalar k. As output it gives a 32-byte string output representing the x-coordinate of $Q = kP$. More details are given in section 3.

2.2 Cryptographic background

2.2.1 Introduction

Assume there are two parties that want to communicate securely with each other. These parties, commonly called Alice and Bob for convenience, need a way to send messages to one another, such that an evil adversary - commonly called Eve or Mallory - cannot 'listen' to the messages being sent. When this attacker can only read the messages, he is called a passive attacker. If he is also able to alter or even delete them, he is an active attacker. When talking about the Internet, one can not easily assume that the communication line is secure, so Alice and Bob need to 'encrypt' their messages. Encryption is the process of transforming messages in such a way that adversaries cannot read the messages on this insecure line, but Alice and Bob can. Previously this was always done by means of a shared secret: Alice and Bob both have a key with which they can encrypt and decrypt messages. This key is practically a long string of characters or numbers. Because both parties have the same key, it is called symmetric cryptography.

2.2.2 Diffie-Hellman

But what do we do if we do not have a shared secret? Now key-agreement protocols come into play, we are looking at Diffie-Hellman in particular, as proposed by Whitfield Diffie and Martin Hellman in [DH76]. Algorithm 3 shows how the protocol works. It assumes that there is some common knowledge between A and B. If there is no such knowledge, A can openly send it. The crucial part here is that Alice and Bob never share their secret knowledge; they only share the public knowledge. Standard Diffie-Hellman is based on discrete logarithms, i.e. $P_A = g^{S_A} \bmod p, P_B = g^{S_B} \bmod p$, where p is some prime number, known by both parties, and g is a generator of the multiplicative group of integers modulo p. A generator is an element of the group G, such that any element $a \in G$ can be written as $a = g^n$ for some integer n. From this follows that $(g^A)^B \equiv (g^B)^A \bmod p$.

Algorithm 3 Diffie-Hellman Key Agreement.

```
1: Commonly known are a group $G$ and a generator $g$ of $G$.
2: Alice has secret knowledge $X_A \in G$, Bob has $X_B \in G$.
3: A and B compute their 'public' knowledge, respectively $P_A = g^{S_A} \bmod p$
   and $P_B = g^{S_B} \bmod p$.
4: Alice sends $P_A$ to Bob, Bob sends $P_B$ back.
5: A computes $S_{AB} = X_B^{P_A} \bmod p$, B computes $S_{AB} = X_A^{P_B} \bmod p$
6: Both parties have a shared secret $S_{AB}$ now.
```

The only thing a passive attacker sees on the insecure line are G, g, and Alice and Bob's public keys, P_A and P_B. It is assumed that it is unfeasible for this attacker to break Diffie-Hellman, based on the assumption of the discrete logarithm problem: Given g and g^x, compute x. Infeasible means that it is impossible for the adversary to compute it in reasonable time, using modern supercomputers. An active attacker, i.e. an attacker that can send, delete and alter messages is able to impersonate Bob, because Alice is unable to check who the person sending actually is. There are ways to deal with this particular problem, but this is out of the scope of this thesis.

2.2.3 Elliptic curve Diffie-Hellman

The elliptic-curve key agreement protocol is based on the basic Diffie-Hellman protocol, but uses, as its name implies, elliptic curves. Algorithm 4 shows how the flow of the protocol goes. In the standard case, first Alice and Bob need to agree upon the domain parameters. In Curve25519 however, the curve E and prime p are fixed. A passive attacker is assumed to be unable to break this protocol in reasonable time, because all they can see is Q_A and Q_B, and by the elliptic curve discrete logarithm assumption it is hard to compute d_A such that $d_A G = Q_A$.

Algorithm 4 Elliptic curve Diffie-Hellman Key Agreement.

```
1: Curve $E$, its group order $n$ and a point $P$ on this curve are agreed upon
   between Alice and Bob.
2: Alice and Bob have as private key $d_A$ or $d_B$, an integer between 0 and $n$.
3: A and B compute their public key, respectively $Q_A$ and $Q_B$, by computing
   $d_A G$ and $d_B G$.
4: Alice sends $Q_A$ to Bob, Bob sends $Q_B$ back.
5: A computes $d_A Q_B$, B computes $d_B Q_A$.
6: The shared secret is now $x_k$, where $(x_k, y_k) = d_A Q_B = d_B Q_A = d_A d_B P =$
   $d_B d_A P$.
```

2.2.4 Timing attacks and countermeasures

A side-channel attack is an attack that is based on the actual implementation of a cryptographic protocol. Because this implementation is very concise and manually verifiable, it should be easier to prevent any implementation-specific attacks. As has been said before, the more lines of code a program has, the bigger the chance that a bug will sneak in. AES, for example, which is used in OpenSSL, has a couple of different side-channel attacks. [Ber05] [Per05] and [OST06] [Koc96]

In the original paper [Ber06] Bernstein tries to prevent known side-channel attacks, that are in some implementations of cryptosystems. One of these is the timing attack. In general, timing attack is any attack that analyses the time taken to execute the algorithm. More specific, there are two particular timing attacks that Berstein tries to prevent.

First is the attack on lookup tables. If we have a secret key $K = (K_0, K_1, ..., K_n)$, and we want an element of K, equal to m, a naive implementation would be:

```
for(i=0; i<n; i++)
{
  if(K[i] == m)
    return i;
}
```

However, this is not very secure, as this lookup does not execute in constant time. An adversary with a precise measuring device can measure the time taken. This can result in a leak of information, an adversary now knows at what spot in K m is located. There is no such lookup in both the original implementation as my short one.

The other timing attack is a little more subtle, and harder to prevent. It is based on modern processors' ability to do branch prediction: if a branch (e.g. an if-then-else structure) gets fetched in the microprocessor, before the statement is actually executed, the processor will try to guess which way this branch will go. If it fails to predict correctly, all fetched statements in the pipeline will be discarded, leading to a measurable loss in time. This can leak information if the branch is based on the secret key, and is especially notable if we have something like:

```
for(i=0; i<n; i++)
{
  if(K[i]%0)
    swap;
}
```

This can leak information because if the ith bit of K is even, an adversary will be able to know this because of the branch prediction. We will need to counter this by swapping in constant time, this is explained in more detail in section 3.3.

2.2.5 NaCl

NaCl (short for Networking and Cryptography library, pronounced "salt") is a software library for secure Internet communication, much like OpenSSL. It was developed mostly by Daniel Bernstein, Tanja Lange and Peter Schwabe, and offers high security, high speeds and already low code size for its implementations. But best of all, anyone can contribute an implementation of an already existing fuction. This is why this library was chosen for this implemenation of the protocol. NaCl can be accessed at http://nacl.cr.yp.to.

2.2.6 Previous implementations

In most implementations [Ber06][CS09][vD][Lan12], code size is not what most cryptographers aim for. The primary target here is almost always speed, in clock cycles or actual milliseconds. My implementation is significantly shorter than any of the above mentioned. With that comes the highest manual verifiability of all of them. The obvious trade-off here is speed. But because it is based on the fast NaCl-implementation, it is sufficiently fast enough for typical cryptographical applications. Second, my implementation is made in C for the NaCl-library, meaning that it has higher portability than most implementations, meaning that it can be compiled on most CPUs without any additional trouble. Because it is based upon Bernstein's original curve25519, there is no time variability, it is immune to hyperthreading and cache timing attacks as seen in 2.2.4. Check [Ber06] for further Security analysis.

3 Implementation Details

The full code is given in Appendix A. The lines indicate where the possible tweet-seperations could be. It has a total of 19 tweets, not 18, because of aesthetic and functional reasons, mostly to try to maintain readability of the code, unlike in Appendix B, where all layout characters have been removed.

3.1 Representation of elements of $\mathbb{F}_{2^{255}-19}$

We want to be able to perform arithmetic on integers modulo $2^{255} - 19$. Unfortunately, C does not support integers of this size, so we will have to think of something else. A 32-bit integer in C holds values from $-(2^{31})$ to $(2^{31} - 1)$. A long long integer (shortened as lli) has - independent of the architecture - 64 bits, from $-(2^{63} - 1)$ to (2^{63}). It is still not quite enough, so a long long integer array of length 16 is used. $16 \cdot 64 = 1024$, which gives a lot of overhead. This overhead is used because we need to be able to buffer the big outputs of the multiplication step. The signed long long int representation was chosen here to minimise code size. An element $A \in \mathbb{F}_{2^{255}-19}$ is represented as 16 64-bit integers $(a_0, \ldots, a_{15})$ with $A = \sum_{i=0}^{15} a_i 2^{16 \cdot i}$.

The code can be devided into two sections: the field arithmetic, and the actual elliptic curve arithmetic.

3.2 Field arithmetic

The elliptic curve arithmetic needs to be able to do simple field arithmetics such as addition, subtraction, multiplication and computing multiplicative inverses.

Algorithm 5 Addition.

```
Input: input arrays $A$ and $B$
1: for $i \leftarrow 0, 15$ do
2:     $out_i \leftarrow b_i + b_i$                ▷ (component-wise addition)
3: end for
4: return $out$
```

The pseudocode for addition and subtraction (here omitted) should be trivial. They both get two input arrays, A and B, and produce an output-array *out* by doing a component-wise addition or subtraction respectively. Because we have so much overhead in this representation, and assuming that we will not encounter many additions or subtractions in a row, we will not have to worry about overflowing and carrying. Because carrying requires a significant amount of extra coding (as can be seen for instance in algorithm 7), having this representation helps to keep code size down.

Of course, there are a lot of different ways to compute multiplications. This specific one was chosen because it is the algorithm with most concise code. In this algorithm, multiplication exists of three steps: first computing $(out_0, out_1, ..., out_{31})$, then reducing it modulo p, which is done in two steps. The first step, calculating the actual multiplication, is seen in algorithm 6.

Algorithm 6 Multiplication, step 1.

```
Input: input arrays $A$ and $B$
1: $out \leftarrow 0$
2: for $i \leftarrow 0, 15$ do
3:     for $j \leftarrow 0, 15$ do
4:         $temp \leftarrow a_i \cdot b_i$
5:         $out_{i+j} \leftarrow out_{i+j} + temp$
6:     end for
7: end for
8: return $out$ mod $p$
```

The next step is reducing it modulo p, which, in turn, is done in two steps. First we bring it down to the regular size of 16 elements, which is seen in algorithm 7.

Algorithm 7 Reducing.

```
Input: an input array $out = (out_0, out_1, ..., out_{31})$
1: for $i \leftarrow 0, 15$ do
2:     $out_i \leftarrow out_i + 38 \cdot out_{i+16}$
3: end for
4: return $out$, $(out_0, out_1, ..., out_{15})$
```

This is correct because in our representation $out_{16} = out \cdot 2^{16 \cdot 16} = out \cdot 2^{256}$ and $2^{256} = 38 \bmod p$, because:

$$\begin{aligned} 2^{255} - 19 &\equiv 0 \bmod p \Leftrightarrow \\ 2 \cdot (2^{255} - 19) &\equiv 0 \bmod p \Leftrightarrow \\ 2^{256} - 38 &\equiv 0 \bmod p \Leftrightarrow \\ 2^{256} &\equiv 38 \bmod p \qquad \text{if } p = 2^{255} - 19 \end{aligned}$$

After we have fitted the original array into 16 elements, these elements will have a size of more than 32 bits, and can therefore not be used for another operation. We thus have to carry from each limb to the next higher in what is called here the "carrying algorithm". Its pseudocode is shown in algorithm 8:

Algorithm 8 Carrying.

```
Input: an input array $out = (out_0, out_1, ..., out_{15})$
 1: for $i \leftarrow 0, 15$ do
 2:     $out_i \leftarrow out_i + 2^{16}$                         ▷ (needed for 'packing')
 3:     $temp \leftarrow out_i$
 4:     $temp \leftarrow temp/2^{16}$
 5:     if $(i \neq 15)$ then
 6:         $out_{i+i} \leftarrow out_{i+i} + temp$
 7:     else
 8:         $out_0 \leftarrow out_0 + (temp - 1) \cdot 38$        ▷ (if $i = 15$, carry to first bit)
 9:     end if
10:     $out_i \leftarrow out_i - (temp \cdot 2^{16})$
11: end for
12: return $out \bmod p$
```

It is implemented as follows:

```
for(i=0; i<16; i++)
{
  o[i]+=(1<<16);
  c = o[i]>>16;
  o[(i+1)*(i<15)]+=c-1+37*(c-1)*(i==15);
  o[i] -= c<<16;
}
```

Here it can be seen that the last step is actually integrated inside the for-loop, to decrease code size.

Inversion is done with Fermat's little theorom, which states that if p is prime, for any integer a it holds that $a^p - a$ is an integer multiple of p, or:

$$\rightarrow a^p - a \equiv 0 \bmod p$$
$$\rightarrow a^p \equiv a \bmod p$$
$$\rightarrow a^{p-1} \equiv 1 \bmod p$$
$$\rightarrow a^{p-1} \equiv a^{-1} \cdot a \bmod p$$
$$\rightarrow a^{p-2} \equiv a^{-1} \bmod p$$

where p is prime and a^{-1} is the multiplicative inverse of $a \bmod p$. So what we want to compute in order to get the inverse is $a^{2^{255}-21}$. However, in C exponentiations are not natively supported, so we need something else to compute exponentiations. Other implementations often use an algorithm consisting of a set amount of squarings and multiplications for maximisation in speed. Instead, the square-and-multiply algorithm was chosen here because it needs much less code and speed is not the main focus of this thesis. The pseudocode can be seen in algorithm 9.

Algorithm 9 Square-and-multiply algorithm.

```
Input: an input array $X$, exponent $n = (n_0, n_1, ..., n_\ell)$
    $out \leftarrow X$
    for $i \leftarrow \ell, 0$ do
        $out \leftarrow out^2$
        if($n_i == 1$) $out \leftarrow out \cdot X$
    end for
    return $out$
```

Because we only need to do exponentiations with a fixed exponent, namely $p-2$, we do not need to have the exponent as a (dynamic) input, and we can try to shorten the algorithm. Luckily for us, $2^{255} - 21$ has a very structured representation in binary: $\underbrace{111...111}_{250}01011$. This makes it possible to write very short code as follows:

```
for(a=253;a>=0;a--)
{
  sq(c,c);
  if(a!=2&&a!=4)
    mul(c,c,i);
}
```

This works because it always needs to multiply when the exponent's ith bit is odd, and p_i is only even at p_2 and p_4, as seen in the binary representation.

3.3 Elliptic curve arithmetic

The main loop is an implementation of the Montgomery Ladder as seen in Algorithm 1 in section 2.1.3. It iterates over the bits of the scalar n, swapping bits if $n_i == 0$, and then performing a Ladder Step. The swapping is a very important part to make the implementation secure. A naive implementation might be:

```
if(!n_i)
  swap(...)_i;
```

In section 2.2.4 the notion of timing attacks was introduced. The naive implementation is not protected against these kind of attacks, so we need something capable of withstanding them: a constant-time conditional swap:

Algorithm 10 Conditional Swap for one bit.

```
Input: input bits $P$ and $Q$, and $Bool \in 0, 1$
1: $X \leftarrow ((P \text{ XOR } Q) \text{ AND } Bool)$
2: $P \leftarrow (P \text{ XOR } X)$
3: $Q \leftarrow (Q \text{ XOR } X)$
```

If $Bool = 0$, then

$$X \leftarrow 0 \quad \text{because for any } A,\ A \text{ AND } 0 = 0. \tag{1}$$

$$P \text{ XOR } 0 = P \tag{2}$$

$$Q \text{ XOR } 0 = Q \quad \text{because for any } A,\ A \text{ XOR } 0 = A. \tag{3}$$

If $Bool = 1$, then

$$X \leftarrow P \text{ XOR } Q \quad \text{because for any } A,\ A \text{ AND } 1 = A. \tag{1}$$

$$P \text{ XOR } Q = Q \tag{2}$$

$$Q \text{ XOR } Q \text{ XOR } P = P \tag{3}$$

(3) works because for any A, B it holds that

$$\begin{aligned} A \text{ XOR } (A \text{ XOR } B) &= \\ (A \text{ XOR } A) \text{ XOR } B &= \\ 0 \text{ XOR } B &= B \end{aligned}$$

Of course, this is only for a single bit, but it can be extended to handle larger numbers. In the code, it is implemented as follows:

```
long long int b1=~(b-1);
for(i=0;i<16;i++)
{
  t=b1&(p[i]^q[i]);
  p[i]^=t;
  q[i]^=t;
}
```

The $b - 1$ operation ensures that we get a long string of all zeros or all ones, depending on b, the $\sim$ negates this bit-wise, so we get a long string of b's back. Now it is possible to XOR this $b1$ bit-wise with an element of the input array. Lastly, a for-loop is needed to go through all the elements in the array.

3.4 Packing and unpacking

To be able to communicate with the NaCl-library, we need translation functions that convert from and to NaCl's representation of large numbers, unpack and pack respectively. The large numbers NaCl uses are 32-byte hexadecimal strings, which means we need 256 bits, or 8 bytes, to encode it.

Unpacking is fairly straightforward:

Algorithm 11 Unpacking.

```
Input: input bytestring $n$, outputarray $out$
1: for $i \leftarrow 0, 15$ do
2:     $out_i \leftarrow n_{2i} + n_{2i+1} * 2^8$
3: end for
```

Packing is a lot more complicated, and reuses the carrycode, which is why we needed to add 2^{16} in that algorithm, but in general it comes down to:

Algorithm 12 Packing.

```
1: Carry enough times to get small enough limbs.
2: With these small limbs go from signed to unsigned representation.
3: Fit this unsigned array into the 8 bytes.
```

4 Results

As can be seen in Appendix A, the quotum of 20 tweets is reached. Specifically, my implementation of Curve25519 has 2442 characters. $2442/140 \approx 17.4$, so the code is even smaller than 18 tweets! But most importantly, no compromises were made with respect to the goals previously mentioned, namely security, readability and portability.

As we have seen in section 2.2.6, we still have the same amount of security as Bernstein's original implementation.

In Appendix B is a version where all aesthetic newline characters, tabs and spaces have been removed. This version is terribly unreadable, but will still compile and has exactly the same behaviour as the 'normal' version. It is used to show the minimum amount of characters necessary, namely 1964! That is only $1964/140 \approx 14$ tweets! It is so short, when printed on a single A4-sheet in an 11-point font, it fits on a $5\frac{1}{4}$-inch floppy disk! This means that $2442 - 1964 = 478$ characters used in the original code are layout characters: newlines, tabs and spaces.

There is also a third program, it is the result of maximising size reduction. Methods used include: removing all whitespaces and unnecessary newlines (as in the other short program, seen in Appendix B), shortening variable and function names to 1 character and substituting the frequent for-loops with short function names. With these techniques, the code can be shortened to just 1611 characters! That is $1611/140 \approx 11.5$ tweets. So we started at 2442 characters. Without the newlines, it was 1964, and with all these substitutions and short variable names, we got 1611 characters!

Although Appendix B and C have fewer characters, readability has a higher priority. In any case, the code of the original version is still short enough to be printed on 2 sheets of A4-paper (as seen in Appendix A). Compare this to other libraries, e.g. OpenSSL, with 21 million lines of code! Every independent function can easily be manually verified to check for any side-channel attacks, for maximum protection.

This is only the key-agreement of the library. The original challenge was to create a full cryptographical library in 100 tweets, which has been met by TweetNaCl. Its paper can be found at `http://tweetnacl.cr.yp.to/index.html` and the entire 100 tweets here: `https://twitter.com/TweetNaCl`.

References

[BBPV12] Billy B. Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ECC-related software bug attack. In *Topics in Cryptology – CT-RSA 2012*, pages 171–186. Springer, 2012. `http://link.springer.com/content/pdf/10.1007f`. 2

[Ber05] Daniel J Bernstein. Cache-timing attacks on AES, 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 9

[Ber06] Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer-Verlag Berlin Heidelberg, 2006. http://cr.yp.to/ecdh/curve25519-20060209.pdf. 2, 7, 9, 11

[Cov] Coverity. Coverty scan 2011 open source integrity report. http://www.coverity.com/library/pdf/coverity-scan-2011-open-source-integrity-report.pdf. 2

[CS09] Neil Costigan and Peter Schwabe. Fast elliptic-curve cryptography on the Cell Broadband Engine. In *Progress in Cryptology–AFRICACRYPT 2009*, pages 368–385. Springer, 2009. http://eprint.iacr.org/2009/016.pdf/. 11

[Cyb] Cybermashup.com. Tweetcipher crypto challenge, @veorq. http://cybermashup.com/2013/06/12/tweetcipher-crypto-challenge/. 2

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976. http://www.cs.jhu.edu/~rubin/courses/sp03/papers/diffie.hellman.pdf. 8

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987. http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf. 3

[Koc96] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology – CRYPTO96*, pages 104–113. Springer, 1996. http://link.springer.com/content/pdf/10.1007%2F3-540-68697-5_9.pdf. 9

[Lan12] Adam Langley. Fast Python implementation of Curve25519, 2012. https://code.google.com/p/curve25519-donna/. 11

[McC04] Steve McConnell. *Code Complete*. Microsoft press, 2004. http://xa.yimg.com/kq/groups/32249099/966630514/name/Microsoft.Press.Code.Complete.Second.Edition.eBook.pdf. 2

[Mil86] V. Miller. Uses of elliptic curves in cryptography. In *Advances in Cryptology, proceedings of Crypto '85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer-Verlag, 1986. 3

[OKS00] Katsuyuki Okeya, Hiroyuki Kurumatani, and Kouichi Sakurai. Elliptic curves with the Montgomery-form and their cryptographic applications. In *Public Key Cryptography*, pages 238–257. Springer, 2000. `http://saluc.engr.uconn.edu/refs/sidechannel/okeya00elliptic.pdf`. 4

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA 2006*, pages 1–20. Springer, 2006. `http://tau.ac.il/~tromer/papers/cache.pdf`. 9

[Per05] Colin Percival. Cache missing for fun and profit, 2005. `http://www.daemonology.net/papers/htt.pdf`. 9

[Twi] Twitter. Tweets indexed with C1T. `https://twitter.com/search?q=%23C1T&src=hash`. 2

[vD] Matthijs van Duin. Java port of Matthijs van Duin's implementation of Daniel J Bernstein's Curve25519. `https://code.google.com/p/curve25519-java/`. 11

A Program, seperated in tweets

```
typedef long long int lli;
typedef lli gf[16];
typedef unsigned char uch;
#define sv static void
#define sq(o,i) mul(o,i,i)
```

```
static gf _121665 = {0xDB41,1};
static const uch base[32] = {9};

sv car(gf o)
{
  int i;
  lli c;
  for(i=0;i<16;i++)
  {
    o[i]+=(1<<16);
```

```
    c = o[i]>>16;
    o[(i+1)*(i<15)] +=
      c-1 + 37*(c-1)*(i==15);
    o[i] -= c<<16;
  }
}
sv add(gf o,gf a,gf b)
{
  int i;
```

```
  for(i=0;i<16;i++)
    o[i]=a[i]+b[i];
}
sv sub(gf o,gf a,gf b)
{
  int i;
  for(i=0;i<16;i++)
    o[i]=a[i]-b[i];
}
sv mul(gf o,gf a,gf b)
{
```

```
  lli i,j,c[31];
  for(i=0;i<31;i++)
    c[i] = 0;
  for(i=0;i<16;i++)
    for(j=0;j<16;j++)
      c[i+j] += a[i] * b[j];
  for(i=16;i<31;i++)
```

```
    c[i-16] += 38*c[i];
  for(i=0;i<16;i++)
    o[i] = c[i];
  car(o);
  car(o);
}
sv inv(gf o,gf i)
{
  gf c;
  int a;
  for(a=0;a<16;a++)
```

```
    c[a]=i[a];
  for(a=253;a>=0;a--)
  {
    sq(c,c);
    if(a!=2&&a!=4)
      mul(c,c,i);
  }
  for(a=0;a<16;a++)
    o[a]=c[a];
}
```

```
sv sel(gf p,gf q,int b)
{
  lli t,u,i,b1=~(b-1);
  for (i=0;i<16;i++)
  {
    t = b1 & (p[i]^q[i]);
    p[i] ^= t;
    q[i] ^= t;
  }
}
```

```
sv mainloop(lli x[32],uch *z)
{
  gf a,b,c,d,e,f;
  lli p,i;
  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)
  {
    p = (z[i>>3] >> (i&7))&1;
    sel(a,b,p);
    sel(c,d,p);
    add(e,a,c);
    sub(a,a,c);
    add(c,b,d);
    sub(b,b,d);
    sq(d,e);
    sq(f,a);
    mul(a,c,a);
    mul(c,b,e);
    add(e,a,c);
    sub(a,a,c);
    sq(b,a);
    sub(c,d,f);
    mul(a,c,_121665);
    add(a,a,d);
    mul(c,c,a);
    mul(a,d,f);
    mul(d,b,x);
    sq(b,e);
    sel(a,b,p);
    sel(c,d,p);
  }
  for(i=0;i<16;i++)
  {
    x[i] = a[i];
    x[i+16] = c[i];
  }
}
sv unpack(gf o,const uch *n)
{
  int i;
  for(i=0;i<16;i++)
    o[i] = n[2*i] +
      ((lli)n[2*i+1]<<8);
}
sv pack(uch *o,gf n)
{
  int i,j,b;
  gf m;
  car(n);
  car(n);
  car(n);
  for(j=0;j<2;j++)
  {
    m[0] = n[0] - 0xffed;
    for(i=1;i<15;i++)
    {
      m[i] = n[i] - 0xffff -
        ((m[i-1] >> 16)&1);
      m[i-1] &= 0xffff;
    }
    m[15] = n[15] - 0x7fff -
      ((m[14] >> 16)&1);
    b = (m[15] >> 16)&1;
    m[14] &= 0xffff;
    sel(n,m,1-b);
  }
  for(i=0; i<16; i++)
  {
    o[2*i] = n[i]&0xff;
    o[2*i+1] = n[i]>>8;
  }
}
int crypto_scalarmult(uch *q,
  const uch *n,const uch *p)
{
  uch z[32];
  lli x[32];
  int i;
  for(i = 0;i < 31;++i)
    z[i] = n[i];
  z[31] = (n[31] & 127) | 64;
  z[0] &= 248;
  unpack(x,p);
  mainloop(x,z);
  inv(x+16,x+16);
  mul(x,x,x+16);
  pack(q,x);
  return 0;
}
int crypto_scalarmult_base(uch *q,
  const uch *n)
{
  return crypto_scalarmult(q,n,base);
}
```

B Program, no layout short version

```
#define sv static void
#define sq(o,i) mul(o,i,i)
typedef long long int lli;typedef lli gf[16];typedef unsigned char uch;
static gf _121665 = {0xDB41,1};static const uch base[32] = {9};sv car(gf
o){int i;lli c;for(i=0;i<16;i++){o[i]+=(1<<16);c=o[i]>>16;o[(i+1)*(i<15
)]+=c-1+37*(c-1)*(i==15);o[i]-=c<<16;}}sv add(gf o,gf a,gf b){int i;for
(i=0;i<16;i++)o[i]=a[i]+b[i];}sv sub(gf o,gf a,gf b){int i;for(i=0;i<16
;i++)o[i]=a[i]-b[i];}sv mul(gf o,gf a,gf b){lli i,j,c[31];for(i=0;i<31;
i++)c[i]=0;for(i=0;i<16;i++)for(j=0;j<16;j++)c[i+j]+=a[i]*b[j];for(i=16
;i<31;i++)c[i-16]+=38*c[i];for(i=0;i<16;i++)o[i]=c[i];car(o);car(o);}sv
inv(gf o,gf i){gf c;int a;for(a=0;a<16;a++)c[a]=i[a];for(a=253;a>=0;a--
){sq(c,c);if(a!=2&&a!=4)mul(c,c,i);}for(a=0;a<16;a++)o[a]=c[a];}sv sel(
gf p,gf q,int b){lli t,u,i,b1=~(b-1);for(i=0;i<16;i++){t=b1&(p[i]^q[i])
;p[i]^=t;q[i]^=t;}}sv mainloop(lli x[32],uch *z){gf a,b,c,d,e,f;lli p,i
;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){p=(z[i>>3]>>(i&7))&1;sel(a,b,p);sel(c,d,p);add(e,a,c);sub(a,a,
c);add(c,b,d);sub(b,b,d);sq(d,e);sq(f,a);mul(a,c,a);mul(c,b,e);add(e,a,
c);sub(a,a,c);sq(b,a);sub(c,d,f);mul(a,c,_121665);add(a,a,d);mul(c,c,a)
;mul(a,d,f);mul(d,b,x);sq(b,e);sel(a,b,p);sel(c,d,p);}for(i=0;i<16;i++)
{x[i]=a[i];x[i+16]=c[i];}}sv unpack(gf o,const uch *n){int i;for(i=0;i<
16;i++)o[i]=n[2*i]+((lli)n[2*i+1]<<8);}sv pack(uch *o,gf n){int i,j,b;
gf m;car(n);car(n);car(n);for(j=0;j<2;j++){m[0]=n[0]-0xffed;for(i=1;i<
15;i++){m[i]=n[i]-0xffff-((m[i-1]>>16)&1);m[i-1]&=0xffff;}m[15]=n[15]-
0x7fff-((m[14]>>16)&1);b=(m[15]>>16)&1;m[14]&=0xffff;sel(n,m,1-b);}for
(i=0;i<16;i++){o[2*i]=n[i]&0xff;o[2*i+1]=n[i]>>8;}}int crypto_scalarmult
(uch *q,const uch *n,const uch *p){uch z[32];lli x[32];int i;for(i=0;i<
31;++i)z[i]=n[i];z[31]=(n[31]&127)|64;z[0]&=248;unpack(x,p);mainloop(x,
z);inv(x+16,x+16);mul(x,x,x+16);pack(q,x);return 0;{int
crypto_scalarmult_base(uch *q,const uch *n){return crypto_scalarmult
(q,n,base);}
```

C Program, shortest version

```
#define sv static void
#define sq(o,i) m(o,i,i)
#define F for(i=0;i<16;i++)
#define FS(a,b) for(i=a;i<b;i++)
typedef long long int l;typedef l g[16];typedef unsigned char c;static g cc
={0xDB41,1};static const c bs[32]={9};sv r(g o){l i;l c;F{o[i]+=(1<<16);c=o
[i]>>16;o[(i+1)*(i<15)]+=c-1+37*(c-1)*(i==15);o[i]-=c<<16;}}sv a(g o,g a,g b
){l i;F o[i]=a[i]+b[i];}sv b(g o,g a,g b){l i;F o[i]=a[i]-b[i];}sv m(g o,g a
,g b){l i,j,c[31];FS(0,31)c[i]=0;F for(j=0;j<16;j++)c[i+j]+=a[i]*b[j];FS(16,
31)c[i-16]+=38*c[i];F o[i]=c[i];r(o);r(o);}sv iv(g o,g x){g c;l i;F c[i]=x[i]
;for(i=253;i>=0;i--){sq(c,c);if(i!=2&&i!=4)m(c,c,x);}F o[i]=c[i];}sv e(g p,
g q,l b){l t,u,i,b1=~(b-1);F{t=b1&(p[i]^q[i]);p[i]^=t;q[i]^=t;}}sv ml(l y[32]
,c *z){g x,w,v,u,t,s;l p,i;F{w[i]=y[i];u[i]=x[i]=v[i]=0;}x[0]=u[0]=1;for(i=254
;i>=0;i--){p=(z[i>>3]>>(i&7))&1;e(x,w,p);e(v,u,p);a(t,x,v);b(x,x,v);a(v,w,u);
b(w,w,u);sq(u,t);sq(s,x);m(x,v,x);m(v,w,t);a(t,x,v);b(x,x,v);sq(w,x);b(v,u,s)
;m(x,v,cc);a(x,x,u);m(v,v,x);m(x,u,s);m(u,w,y);sq(w,t);e(x,w,p);e(v,u,p);}F{y
[i]=x[i];y[i+16]=v[i];}}sv up(g o,const c *n){l i;F o[i]=n[2*i]+((l)n[2*i+1]<<
8);}sv pk(c *o,g n){l i,j,b;g m;r(n);r(n);r(n);for(j=0;j<2;j++){m[0]=n[0]-
0xffed;FS(1,15){m[i]=n[i]-0xffff-((m[i-1]>>16)&1);m[i-1]&=0xffff;}m[15]=n[15]
-0x7fff-((m[14]>>16)&1);b=(m[15]>>16)&1;m[14]&=0xffff;e(n,m,1-b);}F{o[2*i]=n
[i]&0xff;o[2*i+1]=n[i]>>8;}}int crypto_scalarmult(c *q,const c *n,const c *p)
{c z[32];l x[32],i;FS(0,31)z[i]=n[i];z[31]=(n[31]&127)|64;z[0]&=248;up(x,p);
ml(x,z);iv(x+16,x+16);m(x,x,x+16);pk(q,x);return 0;}int crypto_scalarmult_bs
(c *q,const c *n){return crypto_scalarmult(q,n,bs);}
```