ImperialViolet

Implementing Elligator for Curve25519 (25 Dec 2013)

There are some situations where you would like to encode a point on an elliptic curve in such a way that it appears to be a uniform, random string. This comes up in some password authenticated key exchange systems, where one doesn't want to leak any bits of the password. It also comes up in protocols that are trying to be completely uniform so as to make life hard for network censors.

For the purposes of this post, I'm only going to be considering Curve25519. Recall that Curve25519 is the set of points (u,v) where $v^2 = u^3 + 486662x^2 + u$ and *u* and *v* are taken from GF($q=2^{255}-19$). Curve25519 works with a Montgomery ladder construction and thus only exchanges *v* coordinates. So, when sending a *v* coordinate there are two ways that an attacker can distinguish it from a random string:

Firstly, since the field is only 255 bits, the 256^{th} bit is always zero. Thus if an attacker sees a series of 32-byte strings where the top bit of the last byte is always zero, then they can be confident that they are not random strings. This is easy to fix however, just XOR in a random bit and mask it out before processing.

Secondly, the attacker can assume that a 32-byte string is a *v* coordinate and check whether $v^3 + 486662x^2 + v$ is a square. This will always be true if the strings are *v* coordinates, by the curve equation, but will only be true 50% of the time otherwise. This problem is a lot harder to fix.

Thankfully; djb, Tanja Lange, Mike Hamburg and Anna Krasnova

have published Elligator [paper]. In that paper they present two maps from elliptic curve points to uniform, random strings and you should, at least, read the introduction to the paper, which contains much more detail on the use cases and previous work.

Elligator 2 is suitable for Curve25519 and there are some hints about an implementation in section 5.5. However, this blog post contains my own notes about implementing each direction of the map with only a couple of exponentiations. You may need to read the paper first before the following makes much sense.

I'll start with the inverse map, because that's what's used during key-generation. Throughout I'll write (x, y) for affine coordinates on the twisted, Edwards isomorphism of Curve25519, as defined for Ed25519; (u,v) for affine coordinates on Curve25519 itself; and X, Y, Z for extended coordinates on the twisted, Edwards curve.

The inverse map, ψ^{-1}, takes a point on the curve with the following limitations and produces a uniform, representative value:

1. $u \neq -A$. (The value *A* is a parameter of Curve25519 and has the value 486662.)
2. $-2u(u + A)$ is a square

Since we're generating the points randomly, I'm going to ignore the first condition because it happens far less frequently than malfunctions in the CPU instructions that I might use to detect it.

The second condition excludes about half the points on the curve: these points don't have any representation as a uniform string. When generating a key we need to detect and restart if we happen upon one of those points. (When working in the other direction, $\psi(r) = \psi(-r)$, which explains why half the image is missing.)

If the point is in the image of the map then the uniform string representation, *r*, is defined by:

$$r = \begin{cases} \sqrt{(-1/2)(u/(u+A))} & \text{if } v \leq (q-1)/2 \\ \sqrt{(-1/2)((u+A)/u)} & \text{otherwise} \end{cases}$$

(If you can't see the equation, you need a browser that can handle inline SVG.)

So our key generation function is going to take a private key and either produce nothing (if the resulting point isn't in the image of the map), or produce a public key and the uniform representation of it. We're working with Curve25519, so the public key, if any, will exactly match the result of the `scalar_base_mult` operation on Curve25519.

Since I happened to have the `ref10` code from Ed25519 close to hand, we'll be calculating the scalar multiplication on the twisted, Edwards isomorphism of Curve25519 that Ed25519 uses. I've discussed that before but, in short, it allows for the use of precomputed tables and so is faster than `scalar_base_mult` with the Curve25519 code.

The result of the scalar multiplication of the base point with the private key is a point on the twisted, Edwards curve in extended coordinates: (X : Y : Z), where *x* = X/Z and *y* = Y/Z and where (x, y) are the affine coordinates on the Edwards curve. In order to convert a point on the Edwards curve to a point on Curve25519, we evaluate:

$$(u, v) = ((1+y)/(y-1), (\sqrt{-1}\sqrt{A+2}u)/x)$$

This is looking quite expensive: first we have to convert from extended coordinates on the Edwards curve to affine, then to affine on Curve25519. In finite fields, inversions (i.e. division) are very expensive because they are done by raising the divisor to the power of q-2 (where $q=2^{255}-19$). Compared to such large exponentiations, multiplications and additions are almost free.

But, we can merge all those inversions and do the whole conversion

with just one. First, substitute the extended coordinate conversion: x = X/Z, y = Y/Z.

$$u = \frac{1 + \frac{Y}{Z}}{\frac{Y}{Z} - 1} = \frac{Y + Z}{Y - Z}$$

$$\begin{aligned} v &= (\sqrt{-1}\sqrt{A+2}\frac{Y+Z}{Y-Z})/\frac{X}{Z} \\ &= (\sqrt{-1}\sqrt{A+2}(Y+Z))/\frac{X(Y-Z)}{Z} \\ &= (\sqrt{-1}\sqrt{A+2}Z(Y+Z))/(X(Y-Z)) \end{aligned}$$

By calculating just 1/(X(Y-Z)), we can get (u,v). The reciprocal can be used directly to calculate *v* and then multiplied by *X* to calculate *u*.

What remains is to check whether -2u(u + A) is a square and then to calculate the correct square root. Happily, we can calculate all that (including both square roots because we want to be constant-time), with a single exponentiation.

Square roots are defined in the standard way for finite fields where q ≅ 5 mod 8:

$$\sqrt{s} = \begin{cases} s^{\frac{q+3}{8}} & \text{if } s^{\frac{q+3}{4}} = s \\ \sqrt{-1}s^{\frac{q+3}{8}} & \text{otherwise} \end{cases}$$

Let's consider the square roots first. Both have a common, constant term of (-1/2) which we can ignore for now. With that gone, both are fractions that we need to raise to (q+3)/8. Section 5 of the Ed25519 paper contains a trick for merging the inversion with the exponentiation:

$$\begin{aligned} (u/(u+A))^{\frac{q+3}{8}} &= u^{\frac{q+3}{8}}(u+A)^{\frac{-(q+3)}{8}+(q-1)} \quad (\text{because } x^{q-1}{=}1 \text{ for all x}) \\ &= u^{\frac{q+3}{8}}(u+A)^{\frac{7q-11}{8}} \\ &= u(u+A)^3(u(u+A)^7)^{\frac{q-5}{8}} \\ &= u(u+A)^3 \cdot c \end{aligned}$$

But let's see what else we can do with the *c* value.

$$
\begin{aligned}
c &= ((u(u+A)^7)^{\frac{q-5}{8}} \\
&= u^{\frac{q-5}{8}}(u+A)^{\frac{7q-35}{8}} \\
c^7 &= u^{\frac{7q-35}{8}}(u+A)^{\frac{49q-245}{8}-6(q-1)} \\
&= u^{\frac{7q-35}{8}}(u+A)^{\frac{q-197}{8}} \\
u^3(u+A)^{25}\cdot c^7 &= u^{\frac{7q-11}{8}}(u+A)^{\frac{q+3}{8}} \\
&= ((u+A)/u)^{\frac{q+3}{8}}
\end{aligned}
$$

So c^7 gives us the variable part of the other square root without another large exponentiation! It only remains to multiply in the constant term of $(-1/2)^{(q+3)/8}$ to each and then, for each, test whether the square root of -1 needs to be multiplied. For the first square root, for example, we should have that $r^2 = -u/(2(u+A))$ thus $2r^2(u+A)+u$ should be zero. If not, then we need to multiply *r* by sqrt(-1).

Lastly, we need to check that the point is in the image of the map at all! (An implementation may want to do this sooner so that failures can be rejected faster.) For this we need to check that -2u(u + A) is a square. We use the fact that an element of the field is a non-zero square iff raising it to (q-1)/2 results in one. Thus we need to calculate $(-2)^{(q-1)/2}(u(u+A))^{(q-1)/2}$. The first part is just a constant, but the latter would be an expensive, large exponentiation, if not for *c* again:

$$
\begin{aligned}
c^4 &= u^{\frac{4q-20}{8}}(u+A)^{\frac{28q-140}{8}-3(q-1)} \\
&= u^{\frac{4q-20}{8}}(u+A)^{\frac{4q-116}{8}} \\
u^2(u+A)^{14}\cdot c^4 &= u^{\frac{4q-4}{8}}(u+A)^{\frac{4q-4}{8}} \\
&= (u(u+A))^{\frac{q-1}{2}}
\end{aligned}
$$

With that result we can pick the correct square root (and in constant time, of course!)

The map in the forward direction

In the forward direction, the map takes a uniform, representative value and produces a public key, which is the *x* coordinate of a Curve25519 point in this context: ψ(r) = u. (Remember that the affine coordinates on Curve25519 are written (u,v) here.) This is much simpler than the reverse direction, although, as best I can tell, equally expensive. Given a value, *r*, one can calculate the value *u* with:

$$
\begin{aligned}
d &= -A/(1+2r^2) \\
\epsilon &= (d^3 + Ad^2 + d)^{\frac{q-1}{2}} \\
u &= \epsilon d - (1-\epsilon)A/2
\end{aligned}
$$

This requires one inversion to calculate *d* and another, large exponentiation to calculate ε. The resulting *u* value will match the output of the key generation stage which, in turn, matches the value that running `scalar_base_mult` from the Curve25519 code would have produced for the same private key.

Note that the Elligator software page says that GMP can compute the Legendre symbol (the ε calculation) in only 9000 cycles, so I'm clearly missing some implementation trick somewhere. Additionally, the Elligator paper raises the possibility of implementing the inversions with a blinded, Euclidean algorithm. That would certainly be faster but I'm not comfortable that I know how to do that safely, so I'm sticking with exponentiation here.

Implementation

I've a constant-time, pure Go implementation, based on the field operations from the pure-C, SUPERCOP, Ed25519 code. The field operations have been ported pretty mechanically from the C code, so it's hardly pretty Go code, but it seems to work.

On my, 3.1GHz, Ivy Bridge system (*with* TurboBoost enabled) the key generation takes 155µs and the forward map operation takes 28µs. (But remember that half of the key generations will fail, so ~double that time.)

I have not carefully reviewed the code, so beware. There are also some non-obvious concerns when using Elligator: for example, if you are hiding a uniform representative in an nonce field then the attacker might assume that it's a field element, negate it and let the connection continue. In a normal protocol, altering the nonce usually leads to a handshake failure but negating an Elligator representative is a no-op. If the connection continues then the attacker can be pretty sure that the value wasn't really random. So care is required.

So if you think that Elligator might be right for you, consult your cryptographer.