An Introduction to Elliptic Curve Cryptography

Tanner Prynn

Elliptic Curves

An **elliptic curve** is a set of points satisfying an equation of the form

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

for coefficients a, b and variables x, y in some field F (of characteristic not 2 or 3). This type of equation is a Weierstrass equation, which is a condensed form of a general cubic equation. One additional restriction is placed on an elliptic curve, which is that $4a^3 + 27b^2 \neq 0$. This condition ensures that the curve is *non-singular*, which allows us to find a tangent line to any point on the curve.
Figure 1 shows two simple elliptic curves.

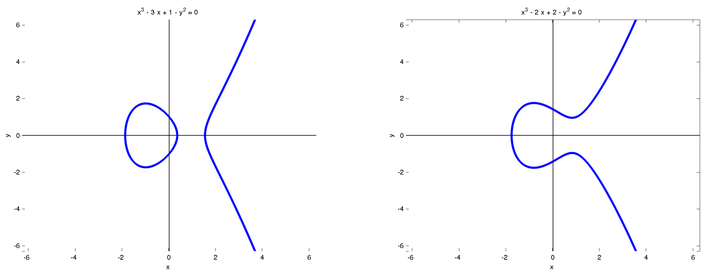

Figure 1: The elliptic curves $y^2 = x^3 - 3x + 1$ and $y^2 = x^3 - 2x + 2$

The Group Law

Define the set of points on the curve as

$$C = \{(x, y) \mid x, y \in F \text{ and } y^2 = x^3 + ax + b\} \cup \infty$$

where ∞ is the point at infinity in the projective plane.
There is an operation $+$ which allows the composition of any two points on the curve. Figure 2 shows the geometric application of this operation.

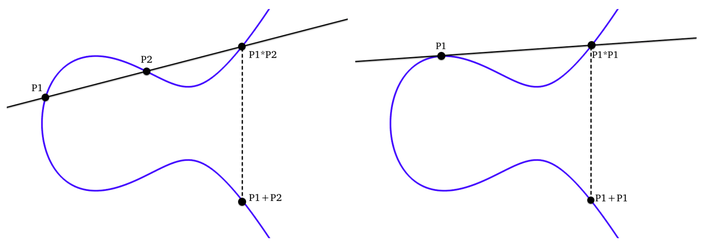

Figure 2: The $+$ operation for two points on an elliptic curve

To add two points together, take the line between them and find its third point of intersection with the curve, then reflect across the x-axis. To add a point to itself, use the tangent line. The identity of $+$ is ∞. The inverse of a point is its reflection across the x-axis.
From the geometric definition, it is possible to derive an algebraic formula for the $+$ operation by finding the equation of the line, and solving the resulting system of equations.

The Discrete Logarithm Problem

Given a group G with operation $*$ we can define a map $\cdot : \mathbb{Z} \times G \to G$ by

$$n \cdot g \mapsto \underbrace{g * g * \cdots * g}_{n \text{ times}}$$

On an elliptic curve, the map $\cdot$ is equal to the repeated addition of a point to itself. We call this map *point multiplication*. For a point P on C, $nP = P + P + \cdots + P$. If we have two points P, Q on C where $nP = Q$, then n is the *elliptic-curve discrete logarithm* (ECDL) of Q with respect to P.
Point multiplication is an example of a *trap-door function*: a function which is simple to compute in one direction but difficult to compute in the other direction. This is the exact operation we will use to construct a cryptosystem.

Naive Multiplication

If an attacker knows P and nP, where N is the order of C, then they can solve the discrete logarithm problem by testing if $mP = nP$ for each m, $1 \leq m < N$. This is the simplest solution to the ECDL, and will complete in $O(NM)$ time, where M is the cost of an elliptic curve multiplication. For small curves, a computer performing naive multiplication will quickly solve this problem. However, there is no known algorithm which feasibly solves the ECDL on a generic cryptographic curve. This problem, known as the Elliptic Curve Discrete Logarithm Problem (ECDLP), is what elliptic curve cryptography is based on. Generic attacks against the ECDLP all have complexity which is a function of the order of the curve, so cryptographic curves are picked such that their order is very large: more than 2^{200}.

Baby-Step Giant-Step

The Baby-Step Giant-Step algorithm rewrites the point $Q = nP$ as $(im + j)P$, with $m = \lceil \sqrt{N} \rceil$. Then jP is computed for $0 \leq j < m$ and stored. Finally, naive multiplication is used to find imP for $0 \leq i < m$, and subtracted from nP to solve for jP.

Baby-Step Giant-Step for ECDLP

$m \leftarrow \sqrt{N}$
for $0 \leq j < m$ **do**
  Compute and store (j, jP)
end for
for $0 \leq i < m$ **do**
  Compute $Q - imP$
  if $Q - imP = jP$ for some j **then**
    return $n \equiv j + im$
  end if
end for

Baby-Step Giant-Step completes in $O(\sqrt{N}M)$ time and $O(\sqrt{N})$ space complexity. This is a large improvement over naive multiplication, but still very slow on cryptographic curves.

Elliptic Curve Diffie-Hellman Exchange

One of the major problems of pre-modern cryptography was that every method of communication required a shared key. This problem is called the key distribution problem. Essentially, if two parties wanted to communicate they needed to meet and exchange a key - there was no way to communicate over an insecure channel without first communicating over a secure channel. The Elliptic Curve Diffie-Hellman Key Exchange (ECDHE) is a method of key derivation which allows two users to derive a shared key over an insecure channel.

Elliptic Curve Diffie-Hellman Exchange

1. Alice and Bob agree on a curve C and a generating point P. (In practice, these parameters are standardized and known to everyone.)
2. Alice generates a random, secret integer a and sends Bob aP.
3. Bob generates a random, secret integer b and sends Alica bP.
4. Alice computes $a(bP)$.
5. Bob computes $b(aP)$.
6. Alice and Bob now share the key abP.

Elliptic Curve Point Multiplication

Point multiplications is one component of Elliptic Curve Cryptography which is very important in practice. Consider the computation of nP for a point P and integer n. Naive multiplication by repeated addition is very slow (exponential). The *Double-and-Add* algorithm computes nP by looping through the bits of n, adding on a 1 bit and doubling on a 0 or 1 bit.
Double-and-Add is much faster than repeated addition, but it leaks information about n by performing a different number of operations based on the binary representation. This information leakage makes an implementation which uses Double-and-Add vulnerable to timing or power analysis. A real-world implementation of ECDHE will use the Montgomery Ladder, which operates in constant time.

Implementation of ECDHE on Curve25519

The result of my research is a pedagogical implementation of the Elliptic-Curve Diffie-Hellman Exchange, using Bernstein's Curve25519. Curve25519 is an elliptic curve in Montgomery form, over the prime field $\mathbb{Z}_p$ where $p = 2^{255} - 19$.

```
# Create ECDH users alice and bob
a = ECDH::User.new('alice')
b = ECDH::User.new('bob')
# alice initiates DH exchange with bob
a.negotiate(b)
# alice and bob now have a shared secret key
a.keys['bob'] == b.keys['alice'] # true
```