Elliptic curve point multiplication

Elliptic curve scalar multiplication is the operation of successively adding a point along an elliptic curve to itself repeatedly. It is used in elliptic curve cryptography (ECC) as a means of producing a one-way function. The literature presents this operation as scalar multiplication, as written in Hessian form of an elliptic curve. A widespread name for this operation is also **elliptic curve point multiplication**, but this can convey the wrong impression of being a multiplication between two points.

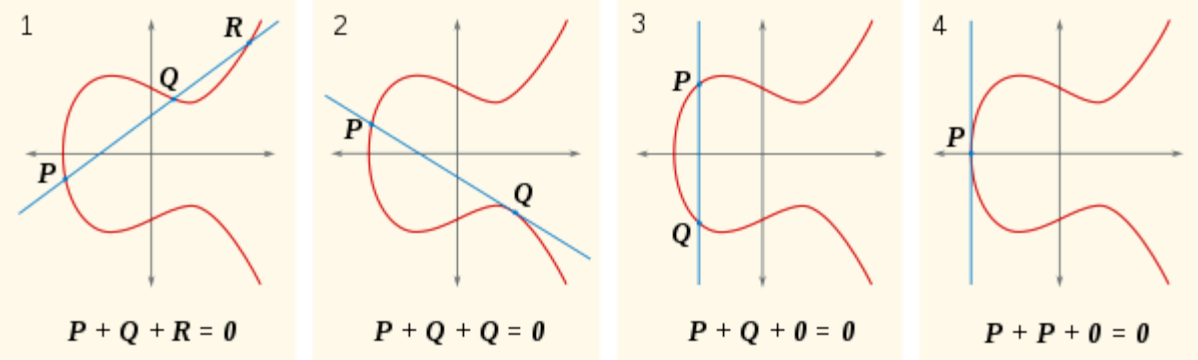
Contents

Basics

Given a curve, E, defined along some equation in a finite field (such as E: $y^2 = x^3 + ax + b$), point multiplication is defined as the repeated addition of a point along that curve. Denote as $nP = P + P + P + \ldots + P$ for some scalar (integer) n and a point $P = (x, y)$ that lies on the curve, E. This type of curve is known as a Weierstrass curve.

The security of modern ECC depends on the intractability of determining n from $Q = nP$ given known values of Q and P if n is large (known as the elliptic curve discrete logarithm problem by analogy to other cryptographic systems). This is because the addition of two points on an elliptic curve (or the addition of one point to itself) yields a third point on the elliptic curve whose location has no immediately obvious relationship to the locations of the first two, and repeating this many times over yields a point nP that may be essentially anywhere. Intuitively, this is not dissimilar to the fact that if you had a point P on a circle, adding 42.57 degrees to its angle may still be a point "not too far" from P, but adding 1000 or 1001 times 42.57 degrees will yield a point that may be anywhere on the circle. Reverting this process, i.e., given $Q=nP$ and P and determining n can therefore only be done by trying out all possible n—an effort that is computationally intractable if n is large.

Point operations



Elliptic curve point operations: Addition (shown in facet 1), doubling (facets 2 and 4) and negation (facet 3).

There are three commonly defined operations for elliptic curve points, addition, doubling and negation.

Point at infinity

Point at infinity is the identity element of elliptic curve arithmetic. Adding it to any point results in that other point, including adding point at infinity to itself. That is:

$$\mathcal{O} + \mathcal{O} = \mathcal{O}$$
$$\mathcal{O} + P = P$$

Point at infinity is also written as 0.

Point negation

Point negation is finding such a point, that adding it to itself will result in point at infinity ($\mathcal{O}$).

$$P + (-P) = \mathcal{O}$$

For elliptic curves that is a point with the same x coordinate but negated y coordinate:

$$(x, y) + (-(x, y)) = \mathcal{O}$$
$$(x, y) + (x, -y) = \mathcal{O}$$
$$(x, -y) = -(x, y)$$

Point addition

With 2 distinct points, P and Q, addition is defined as the negation of the point resulting from the intersection of the curve, E, and the straight line defined by the points P and Q, giving the point, R.

$$P + Q = R$$
$$(x_p, y_p) + (x_q, y_q) = (x_r, y_r)$$

Assuming the elliptic curve, E, is given by $y^2 = x^3 + ax + b$, this can be calculated as:

$$\lambda = \frac{y_q - y_p}{x_q - x_p}$$
$$x_r = \lambda^2 - x_p - x_q$$
$$y_r = \lambda(x_p - x_r) - y_p$$

These equations are correct when neither point is the point at infinity, $\mathcal{O}$. This is important for the ECDSA verification algorithm where the hash value could be zero.

Point doubling

Where the points *P* and *Q* are coincident (at the same coordinates), addition is similar, except that there is no well-defined straight line through *P*, so the operation is closed using limiting case, the tangent to the curve, *E*, at *P*.

This is calculated as above, except with:

$$\lambda = \frac{3x_p^2 + a}{2y_p}$$

where *a* is from the defining equation of the curve, *E*, above.

Point multiplication

The straightforward way of computing a point multiplication is through repeated addition. However, this is a fully exponential approach to computing the multiplication.

Double-and-add

The simplest method is the double-and-add method,[1] similar to multiply-and-square in modular exponentiation. The algorithm works as follows:

To compute *dP*, start with the binary representation for *d*: $d = d_0 + 2d_1 + 2^2 d_2 + \cdots + 2^m d_m$, where $d_0 \,..\, d_m \in \{0, 1\}, m = \lfloor \log_2 d \rfloor$. There are two possible iterative algorithms.

Iterative algorithm, index increasing:

```
N ← P
Q ← 0
for i from 0 to m do
   if d_i = 1 then
       Q ← point_add(Q, N)
   N ← point_double(N)
return Q
```

Iterative algorithm, index decreasing:

```
Q ← 0
for i from m down to 0 do
   Q ← point_double(Q)
   if d_i = 1 then
       Q ← point_add(Q, P)
return Q
```

An alternative way of writing the above as a recursive function is

```
f(P, d) is
   if d = 0 then
       return 0                          # computation complete
   else if d = 1 then
       return P
   else if d mod 2 = 1 then
      return point_add(P, f(P, d - 1)) # addition when d is odd
   else
      return f(point_double(P), d/2)   # doubling when d is even
```

where *f* is the function for multiplying, *P* is the coordinate to multiply, *d* is the number of times to add the coordinate to itself. Example: *100P* can be written as *2(2[P + 2(2[2(P + 2P)])])* and thus requires six point double operations and two point addition operations. *100P* would be equal to *f(P, 100)*.

This algorithm requires $\log_2(d)$ iterations of point doubling and addition to compute the full point multiplication. There are many variations of this algorithm such as using a window, sliding window, NAF, NAF-w, vector chains, and Montgomery ladder.

Windowed method

In the windowed version of this algorithm,[1] one selects a window size *w* and computes all 2^w values of dP for $d = 0, 1, 2, \ldots, 2^w - 1$. The algorithm now uses the representation $d = d_0 + 2^w d_1 + 2^{2w} d_2 + \cdots + 2^{mw} d_m$ and becomes

```
Q ← 0
for i from m to 0 do
    Q ← point_double_repeat(Q, w)
    if d_i > 0 then
        Q ← point_add(Q, d_iP) # using pre-computed value of d_iP
return Q
```

This algorithm has the same complexity as the double-and-add approach with the benefit of using fewer point additions (which in practice are slower than doubling). Typically, the value of *w* is chosen to be fairly small making the pre-computation stage a trivial component of the algorithm. For the NIST recommended curves, $w = 4$ is usually the best selection. The entire complexity for a *n*-bit number is measured as $n + 1$ point doubles and $2^w - 2 + \frac{n}{w}$ point additions.

Sliding-window method

In the sliding-window version, we look to trade off point additions for point doubles. We compute a similar table as in the windowed version except we only compute the points dP for $d = 2^{w-1}, 2^{w-1} + 1, \ldots, 2^w - 1$. Effectively, we are only computing the values for which the most significant bit of the window is set. The algorithm then uses the original double-and-add representation of $d = d_0 + 2d_1 + 2^2 d_2 + \cdots + 2^m d_m$.

```
Q ← 0
for i from m downto 0 do
   if d_i = 0 then
      Q ← point_double(Q)
   else
      t ← extract j (up to w − 1) additional bits from d (including d_i)
      i ← i − j
      if j < w then
         Perform double-and-add using t
         return Q
      else
         Q ← point_double_repeat(Q, w)
```

```
            Q ← point_add(Q, tP)
    return Q
```

This algorithm has the benefit that the pre-computation stage is roughly half as complex as the normal windowed method while also trading slower point additions for point doublings. In effect, there is little reason to use the windowed method over this approach, except that the former can be implemented in constant time. The algorithm requires $w - 1 + n$ point doubles and at most $2^{w-1} - 1 + \frac{n}{w}$ point additions.

w-ary non-adjacent form (*w*NAF) method

In the non-adjacent form we aim to make use of the fact that point subtraction is just as easy as point addition to perform fewer (of either) as compared to a sliding-window method. The NAF of the multiplicand d must be computed first with the following algorithm

```
    i ← 0
    while (d > 0) do
        if (d mod 2) = 1 then
            d_i ← d mods 2^w
            d ← d − d_i
        else
            d_i = 0
        d ← d/2
        i ← i + 1
    return (d_{i-1}, d_{i-2}, …, d_0)
```

Where the signed modulo function *mods* is defined as

```
    if (d mod 2^w) >= 2^{w-1}
        return (d mod 2^w) − 2^w
    else
        return d mod 2^w
```

This produces the NAF needed to now perform the multiplication. This algorithm requires the pre-computation of the points $\{1, 3, 5, \ldots, 2^{w-1} - 1\}P$ and their negatives, where P is the point to be multiplied. On typical Weierstrass curves, if $P = \{x, y\}$ then $-P = \{x, -y\}$. So in essence the negatives are cheap to compute. Next, the following algorithm computes the multiplication dP:

```
    Q ← 0
    for j ← i − 1 downto 0 do
        Q ← point_double(Q)
        if (d_j != 0)
            Q ← point_add(Q, d_jG)
    return Q
```

The wNAF guarantees that on average there will be a density of $\frac{1}{w+1}$ point additions (slightly better than the unsigned window). It requires 1 point doubling and $2^{w-2} - 1$ point additions for precomputation. The algorithm then requires n point doublings and $\frac{n}{w+1}$ point additions for the rest of the multiplication.

One property of the NAF is that we are guaranteed that every non-zero element d_i is followed by at least $w - 1$ additional zeroes. This is because the algorithm clears out the lower w bits of d with every subtraction of the output of the *mods* function. This observation can be used for several purposes. After every non-zero element the additional zeroes can be implied and do not need to be stored. Secondly, the multiple serial divisions by 2 can be replaced by a division by 2^w after every non-zero d_i element and divide by 2 after every zero.

It has been shown that through application of a FLUSH+RELOAD side-channel attack on OpenSSL, the full private key can be revealed after performing cache-timing against as few as 200 signatures performed.[2]

Montgomery ladder

The Montgomery ladder[3] approach computes the point multiplication in a *fixed* amount of time. This can be beneficial when timing or power consumption measurements are exposed to an attacker performing a side-channel attack. The algorithm uses the same representation as from double-and-add.

```
R0 ← 0
R1 ← P
for i from m downto 0 do
   if di = 0 then
      R1 ← point_add(R0, R1)
      R0 ← point_double(R0)
   else
      R0 ← point_add(R0, R1)
      R1 ← point_double(R1)
return R0
```

This algorithm has in effect the same speed as the double-and-add approach except that it computes the same number of point additions and doubles regardless of the value of the multiplicand *d*. This means that at this level the algorithm does not leak any information through timing or power. However, it has been shown that through application of a FLUSH+RELOAD side-channel attack on OpenSSL, the full private key can be revealed after performing cache-timing against only one signature at a very low cost.[4]

References

1. Hankerson, Darrel; Vanstone, Scott; Menezes, Alfred (2004). *Guide to Elliptic Curve Cryptography*. Springer Professional Computing. New York: Springer-Verlag. doi:10.1007/b9764891 (https://doi.org/10.1007%2Fb9764891) (inactive 2020-11-18). ISBN 0-387-95273-X.
2. Benger, Naomi; van de Pol, Joop; Smart, Nigel P.; Yarom, Yuval (2014). Batina, Lejla; Robshaw, Matthew (eds.). *"Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way* (https://www.iacr.org/archive/ches2014/87310103/87310103.pdf) (PDF). Cryptographic Hardware and Embedded Systems – CHES 2014. Lecture Notes in Computer Science. **8731**. Springer. pp. 72–95. doi:10.1007/978-3-662-44709-3_5 (https://doi.org/10.1007%2F978-3-662-44709-3_5). ISBN 978-3-662-44708-6.
3. Montgomery, Peter L. (1987). "Speeding the Pollard and elliptic curve methods of factorization" (https://doi.org/10.2307%2F2007888). *Math. Comp.* **48** (177): 243–264. doi:10.2307/2007888 (https://doi.org/10.2307%2F2007888). JSTOR 2007888 (https://www.jstor.org/stable/2007888). MR 0866113 (https://www.ams.org/mathscinet-getitem?mr=0866113).
4. Yarom, Yuval; Benger, Naomi (2014). "Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack" (https://eprint.iacr.org/2014/140). *IACR Cryptology ePrint Archive*.