Exchanging Shared Secrets: Elliptic-Curve Diffie-Hellman

Introduction

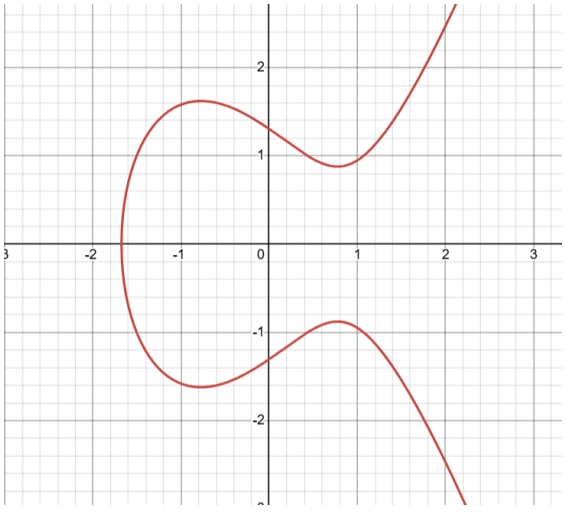
Let's imagine that I want to send a secret message to you, the reader, over an unsecure medium such as the internet. Now, I could simply just send this message to you without any sort of modification, but that would mean that anyone who intercepted the data packets could read the secret message. To prevent this, we need to encrypt the data using a symmetric encryption key so that unauthorized parties cannot read the message if it is intercepted.

Encrypted data may look something like this:
a69f73cca23a9ac5c8b567dc185a756e97c982164fe25859e0d1dcc1475c80a615b2123af1f5f94c11e3
e9402c3ac558f500199d95b6d3e301758586281dcd26

When this ciphertext is sent to the other party, the receiver needs a decryption key. However, like with the message itself, we cannot securely transmit the key over the internet to the other party. Therefore, we need to use a secure key exchange protocol to establish a shared secret (data known by both parties which can be used to generate encryption/decryption keys). One protocol to do this is the Diffie-Hellman Key-Exchange which, proposed by Whitfield Diffie and Martin Hellman in 1976, was the first public key cryptography method of exchanging a secret key over a public channel. Although there are multiple ways of implementing this key exchange protocol, we will be focusing on the version which makes use of elliptic curves known as the Elliptic-Curve Diffie-Hellman (ECDH) key exchange.

A background on elliptic curve point addition

An elliptic curve is a function which typically uses the "Weierstrass" form $y^2 = x^3 + ax + b$, where $4a^3 + 27b^2 \neq 0$. The discriminant of an elliptic curve is given by $\Delta = -16(4a^3 + 27b^2)$ and this condition must be satisfied to maintain a complex curve structure with no cusps or self-intersections. An example of an Weierstrass elliptic curve is shown below:

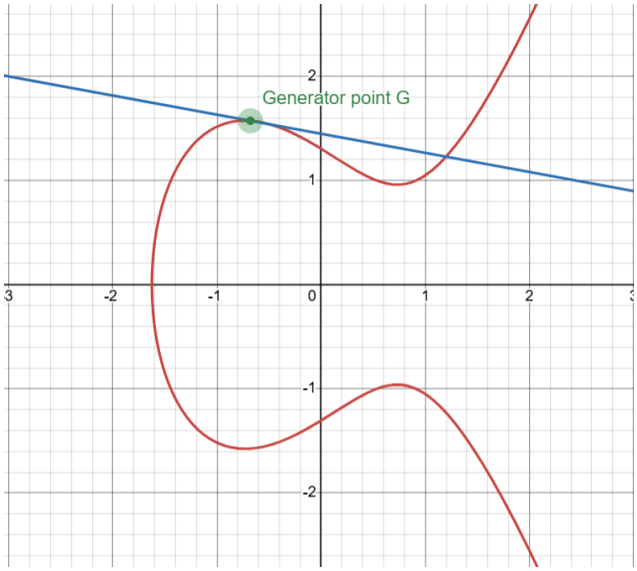


To find k x G under scalar multiplication (repeated point addition), we must use the following process:

Step 1) Draw a tangent to the curve at the general point G. Note that the tangent should cross the curve at a single other point.

To find the gradient (m) of the tangent, we can start by finding the first derivative using implicit differentiation and substituting the generator point G (x_1, y_1) into the gradient function

$m = (3x_1^2 + a)/2y_1$



Step 2) The point where the tangent intersects the graph for the second time is the point -2G.

Using our general formula for the tangent line and equating it to the elliptic curve equation to find points of intersection, we get:

$y = mx + c$ and $y^2 = x^3 + ax + b$

so $x^3 + ax + b = (mx + c)^2$

Rearranging this we get $x^3 - m^2x^2 + (a-2mc)x + (b-c^2) = 0$

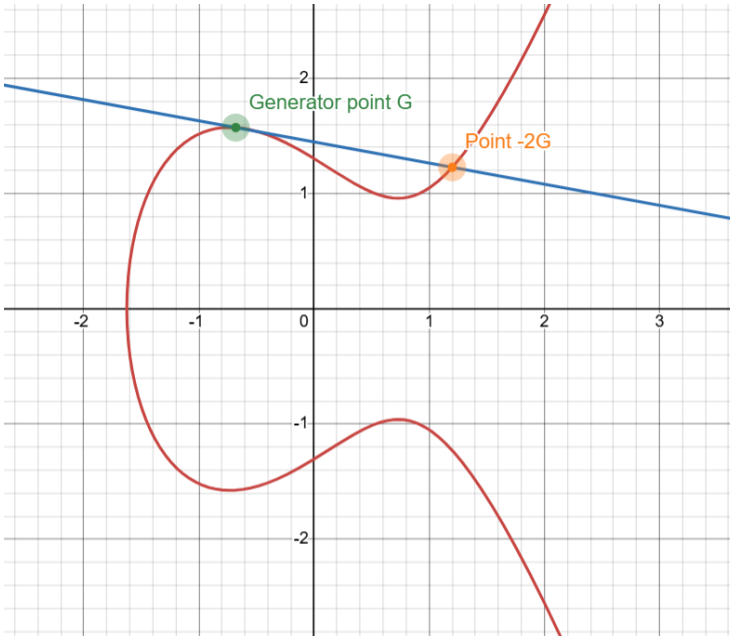
Using Vieta's formulas, we know that the sum of the roots equals -b/a in the cubic equation formed. Therefore, $x_1 + x_2 + x_3 = m^2$. As the point, at which the tangent is to, is a repeated root of the cubic equation, we know $x_1 = x_2$:

$x_3 = m^2 - 2x_1$

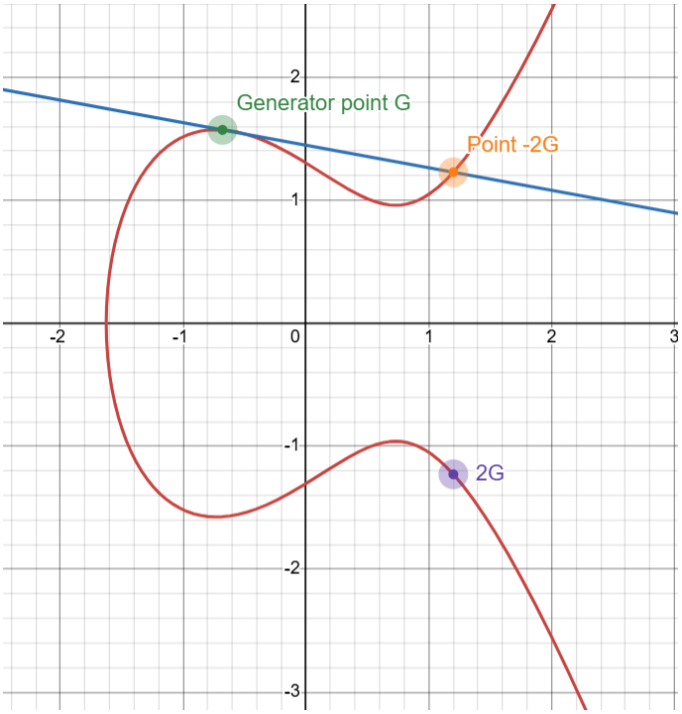
The corresponding y coordinate (y_3), given by substituting x_3 into the standard equation $y-y_1 = m(x-x_1)$ is:

$y_3 = m(x_1-x_3) - y_1$

Hence -2G is given as $y_3 = (m^2 - 2x_1, m(x_1-x_3) - y_1)$, where m is the gradient of the tangent calculated in step 1.

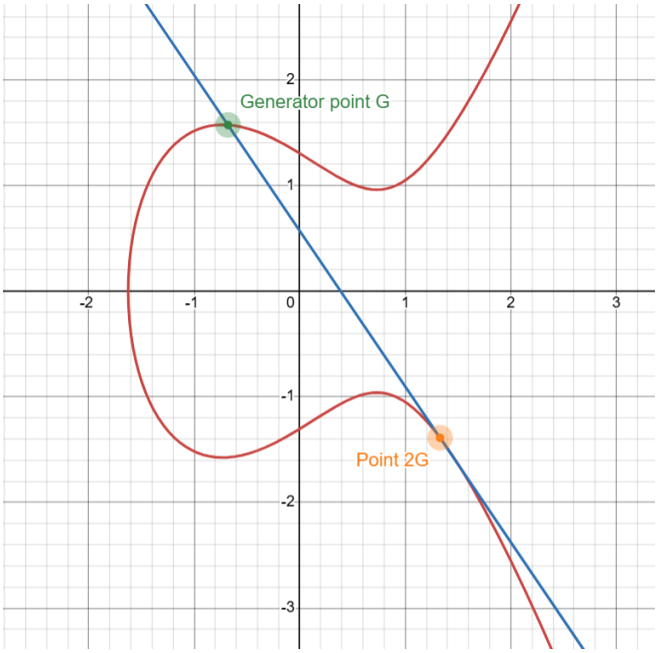


Step 3) Reflect point -2G in the line y=0 to get 2G

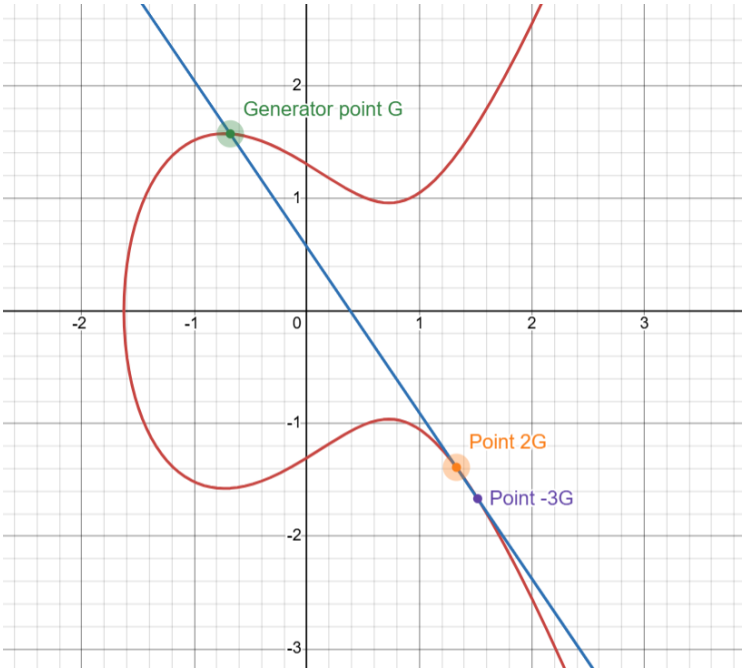


Step 4) Now that we have point 2G, using the "double and add" method, we can obtain 3G by adding 2G to G. Rather than drawing a tangent at G as we did in step 1, to find 3G we must now draw a chord through G and 2G.

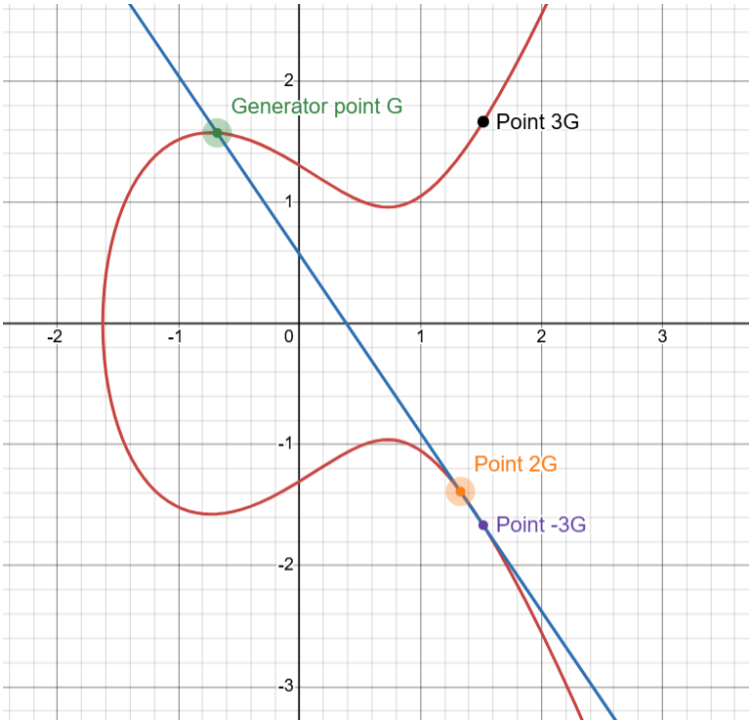
The gradient, m, is now $(y_2 - y_1)/(x_2 - x_1)$, where (x_1, y_1) is G and is 2G (x_2, y_2)



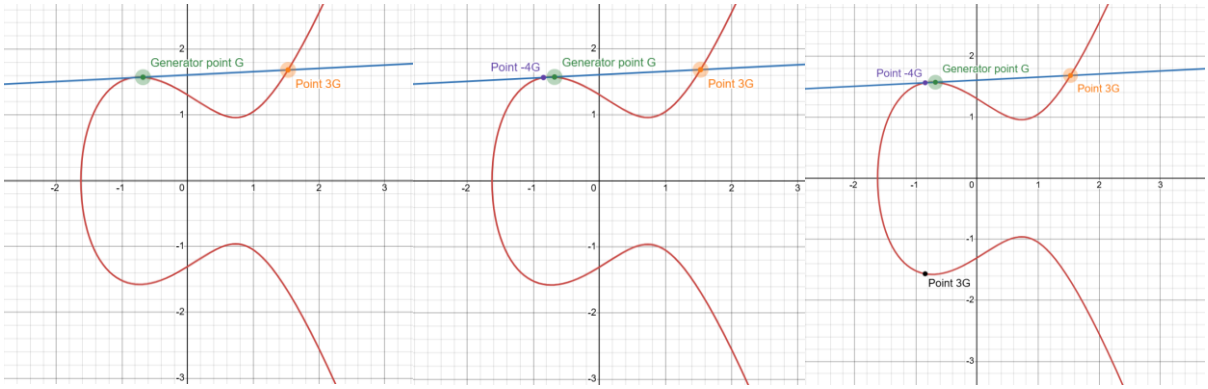
Step 5) The x component of the point of intersection of the chord with the curve, x_3 would now be $(m^2 - x_1 – x_2)$ because x_1 and x_2 are no longer equivalent. y_3 will still remain as $(m(x_1-x_3) – y_1)$.



Step 6) Reflect -3G in the x-axis to get 3G



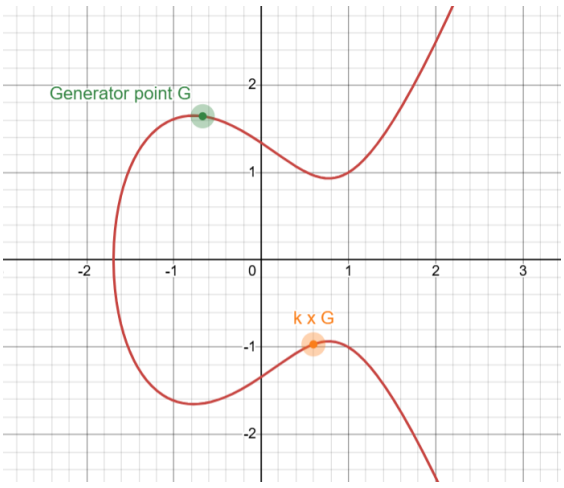
Step 7) Repeat steps 4-6 to obtain 4G, 5G, 6G and so on. Shown below is an example of finding 4G, which would use the double and add method with 3G and G. Likewise, 5G can be obtained by adding 4G and G together on the curve.



Application of Elliptic Curves in Diffie-Hellman key exchange

That's all well and good, but how does this point addition help with solving our initial problem I hear you ask. As previously mentioned, Elliptic Curves are used as the foundation for the Elliptic-Curve Diffie-Hellman key exchange protocol, co-invented in 1985 by Neal Koblitz and Victor Miller. This protocol relies on exchanging public keys to obtain a shared secret which can be used to generate encryption keys known only by the 2 intended parties.

Scalar multiplication on an elliptic curve is a trapdoor (one-way) function, which can easily be computed in 1 way but is incredibly difficult to reverse. Namely, it is very difficult to tell how many times a Generator point (point on the curve chosen as the starting point) has been added to itself using point addition to get to the result k x G. On the curve shown below, is almost impossible to deduce whether the point k x G is the result of adding the G to itself 5, 100 or even 10000 times (depending on the order of the curve). In order to solve this, you could find out every single multiple of the generator point under point addition on the curve, but this is computationally infeasible for curves with very large orders. This infeasibility provides security against attackers who obtain the public keys when they are sent over the internet.



Instead of "guess how many sweets are in the jar", why not have "guess the scalar multiple used to get to k x G from G under point addition"?

How does Elliptic-Curve Diffie-Hellman Key Exchange (ECDH) work?

It is important to note that, unlike the above example, we use the multiplicative group of numbers modulo the prime number p. Therefore, the curve used will be $y^2 = x^3 + ax + b(\text{mod } p)$, where p is a prime number. This ensures the points will be over a finite field between 0 and p-1, ensuring the x and y coordinates of the point will not be too large so that fewer bits are required to store the keys.

The order of the curve is the number of points in the subgroup produced when the generator point is repeatedly added to itself. The cofactor is the total number of valid co-ordinate points on the curve divided by the order of the curve.

Step 1) Alice and Bob both agree on:

-Which Elliptic Curve to use during the transmission (one example of this is secp256k1 which is used in Bitcoin transactions)

-A generator point (G) to use as the starting point to generate a subgroup of points (or all of the points) on the elliptic curve

The order and cofactor of the curve are predefined depending on the generator point G and the curve chosen

Step 2) Alice and Bob both randomly generate their secret keys (α and β respectively), each choosing a number from 1 to n-1, where n is the order of the curve. If the point is greater than or equal to n, it will simply reduce modulo n to a number between 1 and n-1 due to the modulo p operation in the curve's equation.

Alice	The Internet	Bob
Randomly generate public key, $\alpha_{private}$ where $[1 \leq \alpha_{private} \leq n-1]$		Randomly generate private key, $\beta_{private}$ where $[1 \leq \beta_{private} \leq n-1]$

Step 3) The public key is generated as a point on the elliptic curve by multiplying the private key by the generator point (G) on the curve using point addition using Alice and Bobs respective private keys

Find the public key $\alpha_{public} = \alpha_{private} \times G$ using point addition		Find the public key $\beta_{public} = \beta_{private} \times G$ using point addition

Step 4) Alice and Bob exchange their public keys over the internet. It does not matter if these keys are intercepted, as the point generated as the public key gives away little to no information about the private key and the corresponding number of point additions used to get from the generator point to the point representing Alice and Bob's respective public keys. This is why point multiplication on the curve is such a good trapdoor function. Interestingly, we can also compress the public keys by replacing the y coordinate of the public key with information regarding whether it is positive or negative (remembering that each point with a given x value can either be above or below the x-axis due to the x-axis symmetry of the curve). This reduces the amount of data that must be transferred by (almost half), but more operations will be needed later to solve the curve equation for the y co-ordinate again.

Step 5) Validate that the public key point supplied by the other party actually lies on the curve for security reasons (covered shortly).

Step 6) Calculate the shared secret by taking the point of the other person's public key and multiplying it by your own private key.

Step 7) The shared secret will be a point on the curve, where the y value can be ignored, to give a single x coordinate value known only by both parties. This shared secret can then be used to generate symmetric keys which can be used to encrypt and decrypt messages sent by both parties to each other. The private key of each party should then be permanently erased unless using static ECDH (where the same key is used multiple times).

Improving the security of the exchange

As previously mentioned, we must validate the point given by the other party to prevent small subgroup attacks. If the public key is changed, the value provided by the other may be one which is located on another less secure curve, which when multiplied by our private key, can leak information about our key. For example, if curve the point supplied by the attacker lies on a curve with order 2, the result of our private key multiplied by the public key supplied by the other party will be one of 2 values on the unsecure curve, which gives away the last binary digit of our private key. This can be repeated to find our private key one bit at a time.

Secondly, we can use Ephemeral (rather than static) ECDH. This can be done by generating a new shared secret (by repeating the process of exchanging public keys and doing elliptic curve point additions) for each session, meaning the same keys are not used more than once. This has the advantage of enabling Forward Secrecy, so that even if the shared secret is obtained by the attacker, old connections between the parties cannot be decrypted, even if the attacker obtains the private key for one of the communications. It also helps to use a curve with good security and smaller necessary key sizes, such as Curve25519 (used in TLS) and Curve448.

The future of ECDH

So how secure will ECDH be in the future? With significant advancements in quantum computing in the past 10 years, ECDH is vulnerable to quantum computing algorithms. One of these such algorithms is Shor's algorithm, which can be used to crack the elliptic curve discrete logarithm problem (which the protocol relies on) in polynomial time. Subsequently, quantum-resistant

algorithms such as KEMs (lattice-based **k**ey-**e**ncapsulation **m**echanisms) may completely replace ECDH in the future.

Thank you for reading.