Cryptography
Week #10:
Elliptic Curve Cryptography

Rogério Reis, rogerio.reis@fc.up.pt
MSI/MCC/MIERSI - 2024/2025
DCC FCUP

November, 29th 2024

Elliptic Curve Cryptography (*ECC*)

The introduction of elliptic curve cryptography (ECC) in 1985 revolutionized the way we do public-key cryptography. ECC is more powerful and efficient than alternatives like RSA and classical Diffie-Hellman (ECC with a 256-bit key is stronger than RSA with a 4096-bit key), but it's also more complex.

Although first introduced in 1985, ECC wasn't adopted by standardization bodies until the early 2000s, and it wasn't seen in major toolkits until much later: OpenSSL added ECC in 2005, and the OpenSSH secure connectivity tool waited until 2011. But modern systems have few reasons not to use ECC.

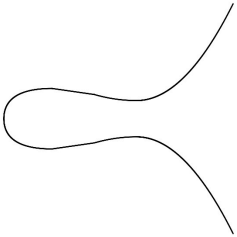
What are Elliptic Curves?

Definition (Elliptic Curve)

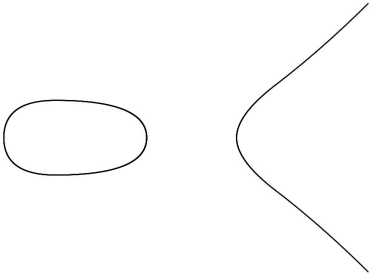
An elliptic curve is the set of solutions to an equation of the form

$$Y^2 = X^3 + AX + B.$$

Equations of this type are called Weierstrass equations.



$E_1 : Y^2 = X^3 - 3X + 3$

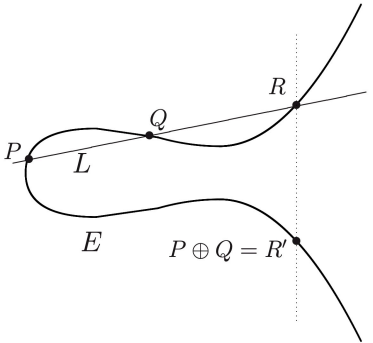


$E_2 : Y^2 = X^3 - 6X + 5$

Addition of two points in a EC

Let P and Q be two points on an elliptic curve E. We start by drawing the line L through P and Q. This line L intersects E at three points, namely P, Q, and one other point R. We take that point R and reflect it across the x-axis (i.e., we multiply its Y-coordinate by -1) to get a new point R'. We write

$$P \oplus Q = R'.$$



An example (1)

Let E be the elliptic curve

$$E : Y^2 = X^3 - 15X + 18.$$

The points $P = (7, 16)$ and $Q = (1, 2)$ are on the curve E. The line L connecting them is given by the equation

$$L : Y = \frac{3}{7}X - \frac{1}{3}.$$

Using this equation in E we get

$$\begin{aligned} \left(\frac{7}{3}X - \frac{1}{3}\right)^2 &= X^3 - 15X + 18 \\ \frac{40}{9}X^2 - \frac{14}{9} + \frac{1}{9} &= X^3 - 15X + 18 \\ 0 &= X^3 - \frac{49}{9}X^2 - \frac{121}{9}X + \frac{161}{9}. \end{aligned}$$

We need to find the roots of this cubic polynomial. In general, finding the roots of a cubic is difficult. However, in this case we already know two of the roots, namely $X = 7$ and $X = 1$, since we know that P and Q are in the intersection $E \cap L$. It is then easy to find the other factor,

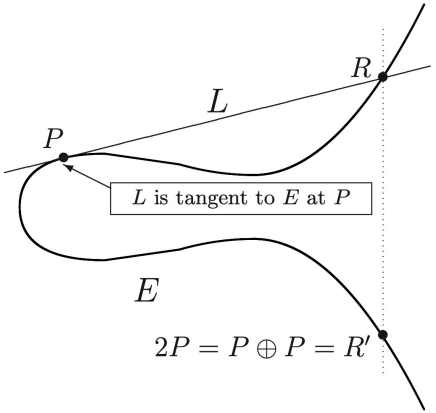
$$X^3 - \frac{49}{9}X^2 - \frac{121}{9}X + \frac{161}{9} = (X - 7)(X - 1)\left(X + \frac{23}{9}\right).$$

Computing the corresponding value of Y we get $R = \left(-\frac{23}{9}, \frac{170}{27}\right)$. And thus

$$P \oplus Q = \left(-\frac{23}{9}, -\frac{170}{27}\right).$$

What if $P = Q$?

Imagine what happens to the line L connecting P and Q if the point Q slides along the curve and gets closer and closer to P. In the limit, as Q approaches P, the line L becomes the tangent line to E at P.



Example (2)

Using the previous example let us compute $P \oplus P$.

$$2Y\frac{dY}{dX} = 3X^3 - 15, \quad \text{so} \quad \frac{dY}{dX} = \frac{3X^2 - 15}{2Y}.$$

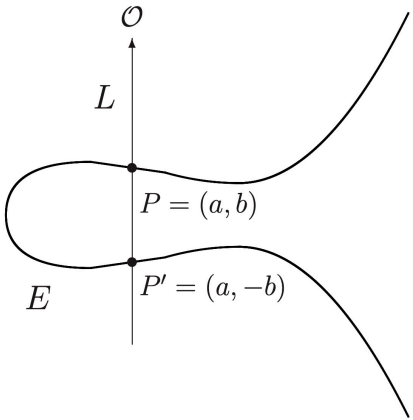
Using the coordinates of $P = (7, 16)$ gives slope $\lambda = 33$, so the tangent line to E at P is given by the equation

$$L : Y = \frac{33}{8}X - \frac{103}{8}.$$

$$\begin{aligned}
\left(\frac{33}{8}X - \frac{103}{8}\right)^2 &= X^3 - 15X + 8 \\
X^3 - \frac{1089}{64}X^2 + \frac{2919}{32}X - \frac{9457}{64} &= 0 \\
(X-7)^2\left(X - \frac{193}{64}\right) &= 0.
\end{aligned}$$

Thus, $2P = P \oplus P = \left(\frac{193}{64}, \frac{223}{512}\right)$.

What about $(a, b) \oplus (a, -b)$?



Simply add a point to every Elliptic Curve: $P \oplus P' = \mathcal{O}$.

Theorem

Let E be an elliptic curve. The addition law on E has the following properties:

i) $P + \mathcal{O} = \mathcal{O} + P = P \ (\forall P)$

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

iii) $(P + Q) + R = P + (Q + R) \ (\forall P, Q, R)$

iv) $P + Q = Q + P \ (\forall P, Q)$

Thus and Elliptic Curve is an Abelian group.

Elliptic Curve Addition Algorithm

Let

$$E : Y^2 = X^3 + AX + B$$

be an elliptic curve and let P_1 and P_2 be points on E.

1. If $P_1 = \mathcal{O}$, then $P_1 + P_2 = P_2$.
2. Otherwise, if $P_2 = \mathcal{O}$, then $P_1 + P_2 = P_1$.
3. Otherwise, let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$.
4. If $x_1 = x_2$ and $y_1 = -y_2$, then $P_1 + P_2 = \mathcal{O}$.
5. Otherwise,

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq P_2, \\ \frac{3x_1^2 + A}{2y_1} & \text{if } P_1 = P_2. \end{cases} \qquad \begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \end{aligned}$$

$P_1 + P_2 = (x_3, y_3)$.

Multiplication (by an integer)

$$nP = \underbrace{P + P + P + \cdots + P}_{n \text{ copies}}.$$

To compute kP efficiently the naive technique of adding P by applying the addition law k–1 times is far from optimal. For example, if k is large (of the order of, say, 2^{256}) as it occurs in elliptic curve–based crypto schemes, then computing k–1 additions is downright infeasible. But there's a trick: you can gain an exponential speed-up by adapting the technique discussed in "Fast Exponentiation Algorithm" to compute $x^e \pmod{n}$. For example, to compute $8P$ in three additions instead of seven using the naive method, you would first compute $P_2 = P + P$,then $P_4 = P_2 + P_2$, and finally $P_4 + P_4 = 8P$.

Elliptic curves over finite fields

In order to apply the theory of elliptic curves to cryptography, we need to look at elliptic curves whose points have coordinates in a finite field $\mathbb{F}_p$. We simply define an elliptic curve over $\mathbb{F}_p$ to be an equation of the form

$$E : Y^3 = X^2 + AX + B \quad \text{with } A, B \in \mathbb{Z}_p \text{ and } 4A^3 + 27B^2 \neq 0$$

and

$$E(\mathbb{F}_p) = \{\langle x, y\rangle \in \mathbb{F}_p \mid y^2 = x^3 + Ax + B\} \cup \{\mathcal{O}\}.$$

Example (3)

Consider the elliptic curve

$$E : Y^2 = X^3 + 3X + 8 \quad \text{over the field } \mathbb{F}_{13}.$$

We can find the points of $E(\mathbb{F}_{13})$ by substituting in all possible values $X = 0, 1, 2, \ldots, 12$ and checking for which X values the quantity $X^3 + 3X + 8$ is a square modulo 13. For example, putting $X = 0$ gives 8, and 8 is not a square modulo 13. Next we try $X = 1$, which gives $1 + 3 + 8 = 12$. It turns out that 12 is a square modulo 13; in fact, it has two square roots,

$$52 \equiv 12 \pmod{13} \text{ and } 82 \equiv 12 \pmod{13}.$$

This gives two points $\langle 1, 5 \rangle$ and $\langle 1, 8 \rangle$ in $E(\mathbb{F}_p)$.

Continuing in this fashion, we end up with a complete list,

$$E(\mathbb{F}_{13}) = \{\mathcal{O}, \langle 1,5\rangle, \langle 1,8\rangle, \langle 2,3\rangle, \langle 2,10\rangle, \langle 9,6\rangle, \langle 9,7\rangle, \langle 12,2\rangle, \langle 12,11\rangle\}\,.$$

Thus $E(\mathbb{F}_{13})$ consists of nine points.

	$\mathcal{O}$	$(1,5)$	$(1,8)$	$(2,3)$	$(2,10)$	$(9,6)$	$(9,7)$	$(12,2)$	$(12,11)$
$\mathcal{O}$	$\mathcal{O}$	$(1,5)$	$(1,8)$	$(2,3)$	$(2,10)$	$(9,6)$	$(9,7)$	$(12,2)$	$(12,11)$
$(1,5)$	$(1,5)$	$(2,10)$	$\mathcal{O}$	$(1,8)$	$(9,7)$	$(2,3)$	$(12,2)$	$(12,11)$	$(9,6)$
$(1,8)$	$(1,8)$	$\mathcal{O}$	$(2,3)$	$(9,6)$	$(1,5)$	$(12,11)$	$(2,10)$	$(9,7)$	$(12,2)$
$(2,3)$	$(2,3)$	$(1,8)$	$(9,6)$	$(12,11)$	$\mathcal{O}$	$(12,2)$	$(1,5)$	$(2,10)$	$(9,7)$
$(2,10)$	$(2,10)$	$(9,7)$	$(1,5)$	$\mathcal{O}$	$(12,2)$	$(1,8)$	$(12,11)$	$(9,6)$	$(2,3)$
$(9,6)$	$(9,6)$	$(2,3)$	$(12,11)$	$(12,2)$	$(1,8)$	$(9,7)$	$\mathcal{O}$	$(1,5)$	$(2,10)$
$(9,7)$	$(9,7)$	$(12,2)$	$(2,10)$	$(1,5)$	$(12,11)$	$\mathcal{O}$	$(9,6)$	$(2,3)$	$(1,8)$
$(12,2)$	$(12,2)$	$(12,11)$	$(9,7)$	$(2,10)$	$(9,6)$	$(1,5)$	$(2,3)$	$(1,8)$	$\mathcal{O}$
$(12,11)$	$(12,11)$	$(9,6)$	$(12,2)$	$(9,7)$	$(2,3)$	$(2,10)$	$(1,8)$	$\mathcal{O}$	$(1,5)$

How many points can we expect in a *EC*?

Theorem (Hasse)

Let E be an elliptic curve over $\mathbb{F}_p$. Then

$$|E(Fp)| = p + 1 - tp \quad \textit{with } t_p \textit{ satisfying } |t_p| \leq 2\sqrt{p}.$$

The *ECDLP* Problem

We defined before *DLP*: that of finding the number y given some base number g, where $x = g^y \pmod{p}$ for some large prime number p.

Cryptography with elliptic curves has a similar problem: the problem of finding the number k given a base point P where the point $Q = kP$. This is called the elliptic curve discrete logarithm problem, or *ECDLP*.

One important difference between *ECDLP* and the classical *DLP* is that *ECDLP* allows you to work with smaller numbers and still enjoy a similar level of security.

One way of solving *ECDLP* is to find a collision between two outputs, $c_1P + d_1Q$ and $c_2P + d_2Q$. The points P and Q in these equations are such that $Q = kP$ for some unknown k, and c_1, d_1, c_2, and d_2 are the numbers you will need in order to find k.
A collision occurs when two different inputs produce the same output. Therefore, in order to solve ECDLP, we need to find points where the following is true: $c_1P + d_1Q = c_2P + d_2Q$. In order to find these points, we replace Q with the value kP, and we have the following:

$$c_1P + d_1kP = (c_1 + d_1k)P = (c_2 + d_2k)P = c_2P + d_2kP.$$

Thus $c_1 + d_1k = c_2 + d_2k$ modulo the number of the points in the curve, **which is not a secret**.

Thus

$$\begin{aligned} d_2 k - d_1 k &= c_1 - c_2 \\ k(d_2 - d1) &= c_1 - c_2 \\ k &= \frac{c_1 - c_2}{d_2 - d_1} \end{aligned}$$

In practice, elliptic curves extend over numbers of at least 256 bits, which makes attacking elliptic curve cryptography by finding a collision impractical because doing so takes up to 2^{128} operations (the cost of finding a collision over 256-bit numbers.

Diffie–Hellman Key Agreement over Elliptic Curves

The elliptic curve version of *DH* is identical to that of classical *DH* but with different notations. In the case of *ECC*, for some fixed point G, Alice picks a secret random number d_A, computes $P_A = d_A G$ (the point G multiplied by d_A), and sends P_A to Bob. Bob picks a secret random d_B, computes the point $P_B = d_B G$, and sends it to Alice. Then both compute the same shared secret, $d_A P_B = d_B P_A = d_A d_B G$. This method is called *elliptic curve Diffie–Hellman*, or *ECDH*.

Signing with Elliptic Curves

The standard algorithm used for signing with *ECC* is *ECDSA*, which stands for *elliptic curve digital signature algorithm*. This algorithm has replaced *RSA* signatures and classical *DSA* signatures in many applications. It is, for example, the only signature algorithm used in *Bitcoin* and is supported by many *TLS* and *SSH* implementations.

As with all signature schemes, *ECDSA* consists of a signature generation algorithm that the signer uses to create a signature using their private key and a verification algorithm that a verifier uses to check a signature's correctness given the signer's public key. The signer holds a number, d, as a private key, and verifiers hold the public key, $P = dG$. Both know in advance what elliptic curve to use, its order (n, the number of points in the curve), as well as the coordinates of a base point, G.

ECDSA Signature Generation

In order to sign a message, the signer first hashes the message with a cryptographic hash function such as *SHA-256* or *BLAKE2* to generate a hash value, h, that is interpreted as a number between 0 and $n - 1$. Next, the signer picks a random number, k, between 1 and n–1 and computes kG, a point with the coordinates $\langle x, y \rangle$.
The signer now sets $r = x \pmod{n}$ and computes $s = (h + rd)/k \pmod{n}$, and then uses these values as the signature $\langle r, s \rangle$. The length of the signature will depend on the coordinate lengths being used. For example, when you're working with a curve where coordinates are 256-bit numbers, r and s would both be 256 bits long, yielding a 512-bit-long signature.

ECDSA Signature Verification

In order to verify an *ECDSA* signature $\langle r, s \rangle$ and a message's hash, h, the verifier first computes $w = 1/s$, the inverse of s in the signature, which is equal to $k/(h + rd) \pmod{n}$, since s is defined as $s = (h + rd)/k$. Next, the verifier multiplies w with h to find u according to the following formula:

$$wh = hk(h + rd) = u.$$

Then multiplies w with r to find v:

$$wr = rk(h + rd) = v.$$

Having u and v, the verifier computes the point Q:

$$Q = uG + vP$$

where P is the signer's public key, which is equal to dG, and the verifier only accepts the signature if the x coordinate of Q is equal to the value r from the signature.

This process works because, as a last step, we compute the point Q by substituting the public key P with its actual value dG:

$$uG + vdG = (u + vd)G.$$

When we replace u and v with their actual values, we obtain the following:

$$\begin{aligned} u + vd &= hk(h + rd) + drk/(h + rd) \\ &= (hk + drk)/(h + rd) \\ &= k(h + dr)/(h + rd) = k. \end{aligned}$$

This tells us that $(u + vd)$ is equal to the value k, chosen during signature generation, and that $uG + vdG$ is equal to the point kG.

ECDSA vs *RSA* Signatures

When comparing *RSA* and *ECC's* signature algorithms, recall that in *RSA* signatures, the signer uses their private key d to compute a signature as $y = xd \pmod{n}$, where x is the data to be signed and y is the signature. Verification uses the public key e to confirm that $y^e \pmod{n}$ equals x — a process that's clearly simpler than that of *ECDSA*.
RSA's verification process is often faster than *ECC's* signature generation because it uses a small public key e. But *ECC* has two major advantages over *RSA*:

- shorter signatures
- signing speed.

EdDSA and Ed25519

EdDSA is simpler, **faster** in both signature and verification and it is **deterministic**, eliminating the risks of flawed randomness.
The "ingredients" for EdDSA are:

- a finite field $\mathbb{F}_q$, where q is a power of an odd prime;
- an elliptic curve E over $\mathbb{F}_q$, with order $|E(\mathbb{F}_q)| = 2^c p$, where p is a (large) prime;
- a base point G on $E(\mathbb{F}_q)$ with order p;
- an cryptographic hash function *Hash* with $2b$-bits output, where $2^{b-1} > q$, allowing elements of $\mathbb{F}_q$ and coordinates of points of the curve to be represented as strings of b bits.

EdDSA

The private key is a random string σ of b bits. Then $k_b||s = Hash(\sigma)$, with $|k_B| = |s| = b$ bits. Then the private key is k_B and the public key is $K_B = k_b G$.

EdDSA signature

Bob wants to sign a message m.

1. $R = rG$, where $r = Hash(s||m)$;
2. $S \equiv r + Hash(R||K_B|m)k_B \pmod{p}$. This satisfies

$$\begin{aligned} 2^c SG &= 2^c(r + Hash(R||K_B||m)k_B)G \\ &= 2^c rG + 2^c Hash(R||K_B||m)k_B G \\ &= 2^c R + 2^c Hash(R||K_B||m)K_B; \end{aligned}$$

3. Bob send $S||R||m$ to Alice.

EdDSA verification

Alice, have receiving $S||R||m$ from Bob, proceed as following:

1. Verifies that
$$2^c SG = 2^c R + 2^c Hash(R||K_B||m)K_B.$$

Compared with ECDSA this avoids the computation of a modular inverse. Like ECDSA one needs two scalar-point multiplications.

Ed25519

This is a specific intance of EdDSA:

- a twisted Edwards curve based on Curve25519;
- SHA-512 as hash function;
- a base point G to optimise efficiency.

Encrypting with Elliptic Curves

Although elliptic curves are more commonly used for signing, you can still encrypt with them. But you'll rarely see people do so in practice due to restrictions in the size of the plaintext that can be encrypted: you can fit only about 100 bits of plaintext, as compared to almost 4000 in *RSA* with the same security level.

One simple way to encrypt with elliptic curves is to use the Integrated Encryption Scheme (*IES*), a hybrid asymmetric–symmetric key encryption algorithm based on the Diffie–Hellman key exchange. Essentially, *IES* encrypts a message by generating a Diffie–Hellman key pair, combining the private key with the recipient's own public key, deriving a symmetric key from the shared secret obtained, and then using an authenticated cipher to encrypt the message.

When used with elliptic curves, *IES* relies on *ECDLP's* hardness and is called *Elliptic-Curve Integrated Encryption Scheme* (*ECIES*). Assuming one has:

- a key derivation function (*KDF*);
- a message autentication code function (*MAC*);
- a symmetric encryption scheme (E);
- an elliptic curve over a finite field of order n, and a base point G on it;
- Bob has a private key $k_B \in [1, n-1]$ and a public one $K_B = k_b G$;
- optionally, Alice and Bob share some additional data S_1 and S_2.

Enciphering with ECIES

Alice has a message m to send to Bob.

1. $r \leftarrow [1, n-1], R = rG$;
2. $S = P_x$, where $P = (P_x, P_y) = rK_B$ (ensuring that $P \neq \mathcal{O}$);
3. $k_E || k_M = KDF(S || S_1)$;
4. $c = E_{k_E}(m)$;
5. $d = MAC_{k_M}(c || S_2)$;
6. Alice sends Bob $R || c || d$.

Deciphering with ECIES

Bob has received $R||c||d$ from Alice.

1. $S = P_x$, where $P = (P_x, P_y) = k_B R$

$$P = k_B R = k_B rG = rk_B G = rK_B;$$

2. $k_E||k_M = KDF(S||S1)$;
3. fails if $d \neq MAC_{k_M}(c||S_2)$;
4. $m = E^{-1}_{k_E}(c)$.

Pseudo Random Number Generator with ECC (NIST SP 800-90)

Infamous because of the worst reasons... (thank you **Snowden**!)
Let p and Q be two points on an elliptic curve, $s_0 \in [0, n-1]$ (being n the prder of the curve).
The following code will produce $240k$-bit length pseudo-random sequence.

1: **procedure** DECPRNG
2: **for** i=1 to k **do**
3: $s_i \leftarrow x(s_{i-1}P)$
4: $r_i \leftarrow lsb_{240}(x(s_iQ))$
5: **return** $r_1 \cdots r_k$

There are no good reasons to use this method... but it is good to know it for history sake.

Choosing a Curve

When making your selection of the curve, be sure to choose coefficients a and b in the curve's equation $y^2 = x^3 + ax + b$ carefully; otherwise, you may end up with an insecure curve.

- The order of the group should not be a product of small numbers; otherwise solving *ECDLP* becomes much easier.
- Adding points $P + Q$ requires a specific addition formula when Q = P. Unfortunately, treating this case differently from the general one may leak critical information if an attacker is able to distinguish doublings from additions between distinct points. Some curves are secure because they use a single formula for all point addition. (When a curve does not require a specific formula for doublings, we say that it admits a unified addition law.)

- If the creators of a curve don't explain the origin of a and b, they may be suspected of foul play because you can't know whether they may have chosen weaker values that enable some yet-unknown attack on the cryptosystem.

NIST Curves

- The finite field $\mathbb{F}_p$ is chosen with

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96};$$

- equation of the curve is

$$y^2 = x^3 - 3x + b,$$

with $b = \text{SHA1}(c49d3608\,86e70493\,6a6678e1\,139d26b7\,819f7e90)$.

Although no actual weakness has been detected, NIST curves are seen by many with some suspicion. There are also version of NIST curves with 192, 224, 384 and 521 bits.

Curve25519

The curve given by

$$y^2 = x^3 + 486662x^2 + x,$$

over $\mathbb{F}_p$, with $p = 2^{255} - 19$.
This is a **very fast** curve being the "smallest" curve that satisfies Berstein's security criteria (RFC8032).

How Things Can Go Wrong

Elliptic curves have their downsides due to their complexity and large attack surface. Their use of more parameters than classical Diffie–Hellman brings with it a greater attack surface with more opportunities for mistakes and abuse, and possible software bugs that might affect their implementation. Elliptic curve software may also be vulnerable to side-channel attacks due to the large numbers used in their arithmetic. If the speed of calculations depends on inputs, attackers may be able to obtain information about the formulas being used to encrypt.

ECDSA with Bad Randomness

ECDSA signing is randomised, as it involves a secret random number k when setting $s = (h + rd)/k \pmod{n}$. However, if the same k is reused to sign a second message, an attacker could combine the resulting two values, $s1 = (h1 + rd)/k$ and $s2 = (h2 + rd)/k$, to get $s1 - s2 = (h1 - h2)/k$ and then $k = (h1 - h2)/(s1 - s2)$. When k is known, the private key d is easily recovered by computing the following:

$$(ks1 - h1)/r = ((h1 + rd) - h1)/r = rd/r = d.$$

Breaking *ECDH* Using Another Curve

Say that Alice and Bob are running *ECDH* and have agreed on a curve and a base point, G. Bob sends his public key $d_B G$ to Alice. Alice, instead of sending a public key $d_A G$ on the agreed upon curve, sends a point on a different curve, either intentionally or accidentally. Unfortunately, this new curve is weak and allows Alice to choose a point P for which solving *ECDLP* is easy. She chooses a point of low order, for which there is a relatively small k such that $kP = O$.

Now Bob, believing that he has a legitimate public key, computes what he thinks is the shared secret $d_B P$, hashes it, and uses the resulting key to encrypt data sent to Alice. The problem is that when Bob computes $d_B P$, he is unknowingly computing on the weaker curve. As a result, because P was chosen to belong to a small subgroup within the larger group of points, the result $d_B P$ will also belong to that small subgroup, allowing an attacker to determine the shared secret $d_B P$ efficiently if they know the order of P.

One way to prevent this is to make sure that points P and Q belong to the right curve by ensuring that their coordinates satisfy the curve's equation. Doing so would prevent this attack by making sure that you're only able to work on the secure curve.


TO BE CONTINUED