Information Security Lab
Autumn Semester 2022
Module 1, Week 1 – Elliptic Curve Cryptography

Kenny Paterson (@kennyog)

Applied Cryptography Group

https://appliedcrypto.ethz.ch/

Overview of today's lectures

- Elliptic Curves
- Cryptography from Elliptic Curves
- ECDSA and friends

(Next week: cryptanalysis of ECDSA with partially known nonces.)

Overview of module's labs

- **This week**, you will be implementing ECC in Python, starting from a BigNum package.
- That is, we give you "mod *p*" arithmetic for large *p* for free, but the rest is up to you!
- In fact, you will implement ECDSA, a signature scheme based on ECC.
- **Next week**, you will see how vulnerable ECDSA is to side-channel attacks and implementation errors by breaking it using *lattice cryptanalysis.*
- There, we will give you tools for performing lattice reduction, but the rest will be up to you!

Elliptic Curves

Classical Discrete Log Cryptography

- Recall: set p a large prime; q a prime divisor of p-1; g an element of order q mod p.
- So g generates G_q, a cyclic group of prime order q in the set of integers modulo q:

$$G_q = \{g^0 = 1, g^1, g^2, \ldots, g^{q-1}\}.$$

- (p,q,g) are **public parameters** for discrete log based crypto.
- E.g. Ephemeral Diffie-Hellman Key Exchange (EDHKE/DHE):
  - Alice and Bob agree on (p,q,g), e.g. get them from a standard.
  - Alice selects x uniformly at random from {0,1,…,q-1}, and sends Bob $X = g^x$ mod p.
  - Bob selects y uniformly at random from {0,1,…,q-1}, and sends Alice $Y = g^y$ mod p.
  - Alice and Bob can both now compute $Y^x = X^y = g^{xy}$ mod p and use this value to derive a shared key.

Classical Discrete Log Cryptography

Security in the classical discrete log setting depends on the Discrete Logarithm Problem and variants:

The discrete logarithm problem (DLP) in G_q:

Let (p, q, g) be as above. Given as input (p, q, g) and $\boldsymbol{y = g^x \bmod p}$, where x is a uniformly random value in {0,1,…,q-1}, **find $\boldsymbol{x}$**.

Problem: making the DLP in G_q sufficiently hard in the face of the best known algorithms means using large q and p.

e.g. For "128-bit security", p should have 3072 bits and q should have 256 bits.

This makes cryptographic algorithms using the DLP in this setting slow and bandwidth hungry.

Elliptic Curves

- An elliptic curve over a field F is a set of pairs $(x,y) \in F \times F$ called *points*, defined by some equation in x and y defined over F.
- Think of F as the integers mod p for some large prime p (typically 256 bits).
- A common form for the equation of an elliptic curve is

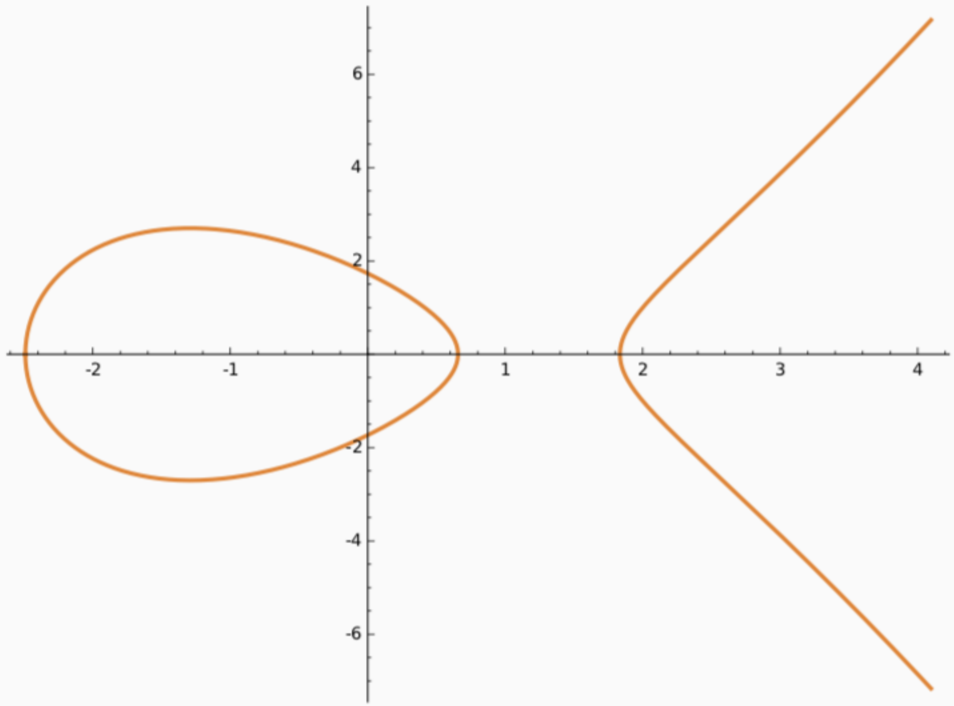
$$E = \{ (x,y) \in F \times F \mid y^2 = x^3 + ax + b \} \cup \{ O \}$$

  where $4a^3+27b^2 \neq 0$ over F.
- Here a and b are coefficients from F that can vary to give us different curves.
- Here "point at infinity" O is a special curve point that does **not** have a representation as a pair $(x,y) \in F \times F$.

Elliptic Curves

- This is called *short Weierstrass form using affine coordinates*; other common forms in cryptography include Montgomery form, Edwards form.
  - Different curve forms offer various trade-offs for implementation and security.
  - For example, Montgomery form makes it easy to do *constant-time scalar multiplication*, while Edwards form makes it easier to avoid branching in ECC code.
  - Entire books have been written on the subject; we will stick with Weierstrass form here.
- In applications, we usually work with one fixed, standardised curve whose properties are carefully evaluated by experts.

Elliptic Curve over the **Rationals** with $a = -5$, $b = 3$




```
sage: E = EllipticCurve([-5, 3])
```

$y^2 = x^3 + 2x + 4$ modulo 5

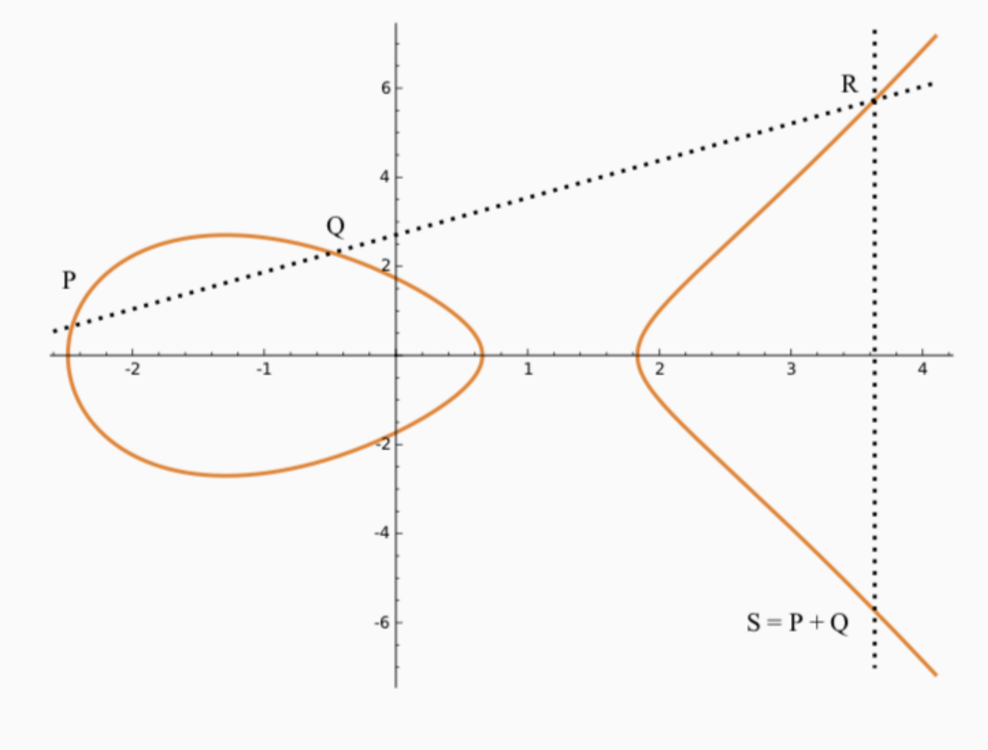
x	0	1	2	3	4
x^3	0	1	3	2	4
$2x$	0	2	4	1	3
4	4	4	4	4	4
y^2	4	2	1	2	1
y	2,3		1,4		1,4

- Here, we see fairly typical behaviour of elliptic curve over a finite field (using artificially small p=5).
- $x^3 + 2x + 4$ takes on 3 distinct values; of these 2 values have square roots mod 5, leading to points (0,2), (0,3), (2,1), (2,4), (4,1), (4,4).
- Including O, we get a total of 7 points on our curve E.

Addition of Points on an Elliptic Curve

- Any pair of points on an elliptic curve can be added to obtain a third point.
- The point at infinity O acts as an (additive) identity for this addition operation.
  - $P + O = O + P = P$ for all elliptic curve points P.
- Each point P has an (additive) inverse denoted $-P$.
  - O is its own inverse: $O + O = O$. (NB Symbols not numbers here!)
  - If $P = (x, y)$ then $-P = (x, -y)$.
  - $P + (-P) = O$.

Addition of Points on an Elliptic Curve



- There is a geometric interpretation of the addition process: to find $P + Q$, draw a straight line through P and Q, find the point of intersection with the curve, and project through the x-axis.
- Special case when $P=Q$: use tangent line at P.

Addition of Points on an Elliptic Curve

- We provide explicit formulae for point addition (P+Q) and point doubling (P+P).
- These are based on the geometric interpretation from the previous slide; they are valid for Weierstrass form.
- Special cases are needed when one or both of P, Q is point at infinity, O, and when Q=-P.
- To **add** two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ with $x_1 \neq x_2$:
  1. $\lambda = (y_2 - y_1)/(x_2 - x_1)$ (slope of line between P and Q).
  2. $x_3 = \lambda^2 - x_1 - x_2$ (x-coord of intersection of that line with curve)
  3. $y_3 = \lambda(x_1 - x_3) - y_1$ (negative of y-coord of same)
  4. return (x_3, y_3)
- To **double** a point $P = (x, y)$, i.e. to compute $P + P$:
  1. $\lambda = (3x^2 + a)/(2y)$ (slope of tangent at P).
  2. $x' = \lambda^2 - 2x$
  3. $y' = \lambda(x - x') - y$
  4. return (x', y')
- You will be implementing these formulae "from scratch".

Elliptic Curves as Groups

- This addition law turns the set of points on an elliptic curve over a field into a **group**.
- (This requires a proof, particularly for the associative law, namely $(P+Q)+R = P+(Q+R)$ for any three points P, Q, R, but we do not provide one here.)
- The group operation is written as "+", and we speak of **adding** two points.
- The identity in the group is the special point O (the point at infinity).
- The group **order** is the number of points on the curve.
- By carefully choosing E, we can ensure that the group has prime or nearly prime order, good for doing crypto.

Example: Elliptic Curves as Groups

- Recall the curve E with equation $y^2 = x^3 + 2x + 4$ over $F = F_5$.
- This curve has points O, (0,2), (0,3), (2,1), (2,4), (4,1), (4,4).
- So the group order is 7, a prime.
- Take $P = (0,2)$.
- Then it so happens that P, $P+P$, $P+P+P$,… gives all 7 group elements.
- So P is a **generator** of the group of points on E.
- Compare with $\{1, g, g^2, \ldots g^{q-1}\}$ in the usual discrete logarithm setting (where we have a subgroup of order q mod p).
- NB group order is not usually prime, and group does not usually have a single generator.

Projective Coordinates

- The equations for adding and doubling points in affine coordinates require computations of modular inverses.
- These are expensive to do.
- Use of projective coordinates allow modular inverses to be avoided except when converting between coordinate systems.
- Curve equation becomes: $Y^2Z = X^3 + aXZ^2 + bZ^3$ with points $(x,y,z) \in F \times F \times F$: now a *homogeneous* equation of degree 3.
- Any point (X,Y,Z) on this curve can be mapped to a point $(X/Z, Y/Z)$ on the affine curve, provided $Z \neq 0$.
- Similarly (x,y) on the affine curve can be mapped to $(x,y,1)$ on the projective version of the curve.
- The point at infinity is represented by the point (0,1,0) on the projective curve (and does not map to a point on the affine curve).
- For "extra fun" in the lab, you may wish to work with projective coordinates. Explicit formulae given in Section 2.2 of Cohen-Miyaji-Ono, Asiacrypt 1998 (https://link.springer.com/content/pdf/10.1007%2F3-540-49649-1_6.pdf)

Scalar Multiplication

- We write $[k]P$ for the operation of adding P to itself k times.
- This is called *scalar multiplication by k*.
- This is the analogue of exponentiation in the usual discrete log setting, i.e. $[k]P$ on curve roughly equivalent to $g^x \bmod p$.
- In our example, P, $[2]P$, $[3]P$,… gives us the full set of points on the curve.
- NB1: $[7]P = O$, c.f. $g^q = 1 \bmod p$ in classical DL setting.
- NB2: If $P = (x,y)$, then $[k]P \neq (kx, ky)$ in general!!!

Scalar Multiplication

- To compute some scalar multiple $[k]P$ of a point P we use an analogue of square-and-multiply called *double-and-add.*
- Suppose we want to compute $[11]P$.
- $11_{10} = 1011_2$, so we compute $[11]P$ by the following chain:

1: P

0: Double: $[2]P$

1: Double and add: $[4]P + P = [5]P$

1: Double and add: $[10]P + P = [11]P$

Scalar Multiplication

- In general, if the scalar k has t bits, then $[k]P$ can be computed in at most t doublings and t additions of points.
- But notice that "D" and "D+A" steps require different numbers of mod p operations.
- Whether "D" or "D+A" is done leaks bits of scalar k.
- Also, total running time is less if k is "small", e.g. MSBs are zero.
- This open up avenues for *side-channel* attacks.
- There is a vast literature involved in making $[k]P$ go fast and be resistant to side-channel attacks.
- Most major crypto libraries now do a fairly good job of this.

Cryptography from Elliptic Curves

The Elliptic Curve Discrete Logarithm Problem

Recall the (classical) discrete logarithm problem:

The Discrete Logarithm Problem in G_q:

Let (p, q, g) be group parameters (so q divides p-1; g has order q mod p).

Set $y = g^x \bmod p$, where x is a uniformly random value in {0,1,…,q-1}.

Given (p, q, g) and y, find x.

The Elliptic Curve Discrete Logarithm Problem (ECDLP):

Let E be an elliptic curve over the field F of prime order p.

Let P be a point of prime order q on E.

Set $Q = [x]P$ where x is a uniformly random value in {0,1,…,q-1}.

Given E and points P, Q, find x.

The Elliptic Curve Discrete Logarithm Problem

- The essence of Elliptic Curve Cryptography is that, except for some special cases, the best algorithms for solving ECDLP run in time $O(q^{1/2})$ where q is the order of the generator P.
- These are in fact *generic* algorithms that work in any finite abelian group.
  - Baby-steps-Giant-Steps, Pollard lambda algorithm, Pollard rho algorithm, Method of Wild and Tame Kangaroos,…
  - These all require running time (and, in some cases, space) that are **exponential** in $\log_2 q$, the bit-size of q.
- The $O(q^{1/2})$ behaviour enables us to choose much smaller parameters than are needed in "normal" discrete-log-based cryptography.
- This results in more compact keys, ciphertexts, etc, and faster cryptographic operations.
- For 128-bit security, we want $O(q^{1/2}) \approx 2^{128}$, so we need q (and hence p) to have 256 bits.

Cryptography from ECDLP

- Most schemes for the DLP setting can be translated easily into the ECDLP setting.
- Example: ECDHE (Elliptic Curve Diffie-Hellman Ephemeral).
  - Alice and Bob agree on a curve E and a base-point P of prime order q.
  - Alice chooses x uniformly at random from $\{0,1,..,q\text{-}1\}$, and sends Bob $[x]P$.
  - Bob chooses y uniformly at random from $\{0,1,..,q\text{-}1\}$, and sends Alice $[y]P$.
  - Both sides can now compute $[xy]P$: Alice computes $[x]([y]P)$ and Bob computes $[y]([x]P)$.

ECC Setup

To set up a system for using elliptic curve cryptography:

- We need to decide on a field F (usually a prime field for some prime p).
- We need to decide on a curve E over that field.
- We need to find a base point P on the curve of known and large prime order q.
- We need to support the new arithmetic of scalar multiplication on our curve, in a fast and secure manner.

Given the additional complexity of the new operations, there is lots of scope for errors and new attack vectors!

- Example: basic doubling and adding operations use different formulae, leading to timing side channels.
- Example: computing $[k]P$ may be faster if MSBs of k are zero, again resulting in timing side channels (and possible leak of ECDSA private key).

Curve Selection

- For the field F of prime order p, a curve E over F has n points where:

$$p + 1 - 2\sqrt{p} \leq n \leq p + 1 + 2\sqrt{p}$$

- This is known as the Hasse-Weil bound; for large p, it means that the *bit-size* of n is the same as that of p.
- Prime order curves (where $n=q$ is prime) are popular and enjoy some implementation advantages.
- Otherwise, we typically ensure $n = h.q$ where h (called the co-factor) is small and q is prime.
- The Schoof-Elkies-Adkin (SEA) algorithm can be used to compute the number of points on an elliptic curve in a fairly efficient manner.
- Easier and safer to rely on curves that are *standardised*.

An example standardised curve: NIST P-256

- $p = 2^{224}(2^{32} - 1) + 2^{192} + 2^{96} - 1$.
- $a = -3$
- b := 5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e 27d2604b.
- $h = 1$; q = FFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC6325
- A base point *P* is also specified.
- NIST P-256 is a curve of prime order *q*; special *sparse form* of *p* potentially makes mod *p* arithmetic faster.
- Very widely supported in crypto libraries.
- *p* and *q* have 256 bits, so complexity of solving ECDLP is about 2^{128}.

An example standardised curve: Curve25519

- Introduced by Bernstein in 2005/2006.
- $p = 2^{255} - 19$, allowing very fast modular reduction.
- Curve equation: $y^2 = x^3+486662x^2+x$.
- Curve has Montgomery form, allowing ECDH operations to be done using only x coordinates in a side-channel resistant manner.
- Group order: $8(2^{252}+ 27742317777372353535851937790883648493)$ – co-factor of 8.
- "Minimal" curve satisfying various security/performance criteria.
- Offers a bit less than 128-bit security, improved speed compared to, e.g. NIST P-256.
- Adopted for use in TLS 1.3 (along with NIST P-256, NIST P-384, NIST P-521 and Curve448-Goldilocks).
- See https://cr.yp.to/ecdh/curve25519-20060209.pdf and RFC 7748 for further details.

Base Point Selection

- Standardised curves normally come with specified base points, so base point selection is not needed in practice.
- Suppose E defined over F has n points where n has a large prime divisor q.
- Choose a non-O point P so that P has order q, i.e. check that $[q]P = O$.
- If $n = q$, then every point P on the curve will have this property; otherwise take a random point P' and compute $[h]P'$ and check $[h]P' \neq O$.
- How to find a random point on the curve?
  - Pick a random x, compute $x^3 + ax + b$, and try to solve for y in the curve eqn:

$$y^2 = x^3 + ax + b \bmod p.$$

  - Requires an algorithm for taking square roots mod p – use Tonelli-Shanks.
  - This algorithm will succeed roughly half the time (half of the non-zero elements mod p are squares).

Point Compression

- The point P can be represented by a pair (x, y) in $F \times F$.
- It then looks as if 2 field elements are needed to represent a point, requiring $2\log_2 p$ bits.
- This can be reduced to $\log_2 p$ bits using **point compression**.
  - Use $\log_2 p$ bits to define the x-coordinate, and 1-bit to represent the "sign" of y.
  - Can always extract two candidates (x, y) and $(x, p\text{-}y)$ for the point given x, by solving $y^2 = x^3 + ax + b \bmod p$.
  - Use the "sign" bit to decide between the two.

Key Pair Generation

- Suppose E defined over F has n points where n has a large prime divisor q; let P be a point of order q.
- To generate key pair for ECC:
  - Choose a random scalar k in $\{0,1,\ldots,q\text{-}1\}$.
  - Set $Q = [k]P$.
  - The private key is k; the public key is Q.
- The problem of extracting the private key from the public key is the ECDLP.
- We've already seen how to use this set up to do an elliptic-curve analogue of ephemeral Diffie-Hellman (ECDHE).

ECDSA and friends

ECDSA

- ECDSA is a translation of DSA from the standard DL setting to the elliptic curve setting.
- DSA was designed to avoid patents on Schnorr's scheme.
- ECDSA (and DSA) specified in: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf and ANSI X9.62 (not free!).
- Detailed description in Boneh-Shoup, Section 19.3, but using different notation: http://toc.cryptobook.us/book.pdf
- Signatures are pairs (r,s) where r and s are integers mod q, the order of base point P; hence 512 bit signatures at the 128-bit security level.
- Public key is a point Q on an elliptic curve, requiring 256 bits at 128-bit security level.
- This is pretty compact!

ECDSA – The Gory Details

Parameters: (E, p, n, q, h, P, H) defining a curve E over field F_p with $n = q \cdot h$ points, subgroup of prime order q and generator P of order q; H is a hash function, e.g. SHA-256 (here we assume output of H is at least bit-size of q).

<u>**KeyGen**</u>:

Set $Q = [x]P$ where x is uniformly random from $\{1,\ldots,q\text{-}1\}$.

Output verification key: $\boldsymbol{Q}$; signing key: $\boldsymbol{x}$.

<u>**Sign**</u>: Inputs (x, m) // x is private key; m is the message to be signed

h = bits2int(H(m)) mod q. // take len(q) MSBs of H(m), cast to BigInt, reduce mod q.

Do:

1. Select k uniformly at random from $\{1,\ldots,q\text{-}1\}$.
2. Compute r = x-coord([k]P) mod q. // [k]P is a point on E; its x-coord is in F_p; we consider that as an integer and reduce mod q.
3. Compute $s = k^{-1}(h + xr)$ mod q.

Until $r \neq 0$ and $s \neq 0$. // works first try w.h.p.

Output (r,s).

ECDSA – The Gory Details

Verify: Inputs (Q, m, (r,s)) // Q is verification key; m is message; (r, s) is claimed signature.

1. check that $1 \leq r \leq q-1$ and $1 \leq s \leq q-1$.
2. compute $w = s^{-1} \bmod q$.
3. compute $h = \text{bits2int}(H(m)) \bmod q$.
4. compute $u_1 = w \cdot h \bmod q$ and $u_2 = w \cdot r \bmod q$.
5. compute $Z = [u_1]P + [u_2]Q$.
6. If (x-coord(Z) mod q == r) then output 1 else output 0.

Correctness:

Suppose (r,s) is a signature for message m under key Q. Then:

$$Z = [u_1]P + [u_2]Q = [s^{-1}h]P + [s^{-1}r]Q = [s^{-1}(h + xr)]P = [k]P.$$

Here we used $s = k^{-1}(h + xr) \bmod q$ from the signing algorithm to obtain $s^{-1}(h + xr) = k \bmod q$.

Recalling that r = x-coord($[k]P$) mod q completes the argument.

ECDSA Security and Implementation Pitfalls

- Implementation requires:
  - Various fiddly conversions of bit-strings to integers, etc: bits2int() and conversion of mod p integers to mod q integers.
  - Uniform sampling of integers k in the range $\{1,\ldots, q\text{-}1\}$ – use rejection sampling (sample from $[0,2^t]$ for t = bitsize(q), until result is in $\{1,\ldots, q\text{-}1\}$).
  - Computation of multiplicative inverses mod q: k^{-1}, s^{-1}.
  - Scalar multiplications: $Q = [x]P$; $[k]P$; $[u_1]P + [u_2]Q$.
  - Sanity checks on r, s.
- There are lots of ways to get some or all of this wrong!
  - Sampling k wrongly, e.g. choose k from $[0,2^t]$ where t is bit-size of q, and reduce mod q.
  - Repeating k, or k being predictable due to bad RNG.
  - Leaking some or all of k through a side-channel attack, e.g. running time of $[k]P$ or computation of k^{-1} mod q.
  - More in next week's lectures…

ECDSA Variants

- ECDSA is very sensitive to randomness failures, e.g. Sony Playstation fail, various cryptocurrency incidents.
- The key issue: if the same *k* is ever used twice by a signer on two different messages, then an attacker can detect this and recover the private key.
- RFC 6979: de-randomisation technique for ECDSA.
  - Essentially, set *k* as PRF(*sk*, *m*) where *sk* is a second private key component.
  - Values *k* are "as good as random" but now no reliance on (P)RNG; highly unlikely that *k* values will repeat if PRF is good.
  - De-randomised signatures are indistinguishable from standard ECDSA signatures, except if same message is signed twice (then same *k* is used and same signature results).
  - Similar trick used in EdDSA (along with a specific curve, and different signing and verification equations).

ECDSA – Formal Security

- ECDSA has an unfortunate malleability property: if (r,s) is a valid signature for message m and verification key Q, then so is (r,-s).
- Hence ECDSA is not SUF-CMA secure; proven to be UF-CMA secure in generic group model (see: D. R. Brown. Generic groups, collision resistance, and ECDSA. Designs, Codes and Cryptography, 35(1):119–152, 2005).

Before the Lab

- Review this lecture, check that you understand what ECC is all about.
- Brush up your mod p arithmetic, revisit/learn how to do modular inversion mod p using Extended Euclidean Algorithm.
- Sharpen up your Python programming skills.
- Read the lab material.
- Start programming!