Cryptographic Engineering

Multiprecision arithmetic II and ECC

Radboud University, Nijmegen, The Netherlands

Spring 2019

Where were we...?

- Last lecture: arithmetic on big integers
- Conclusion at the end:
  - Can use a *redundant representation* for big integers
  - Carries get accumulated in “unused” upper parts of registers
  - Arithmetic becomes essentially polynomial arithmetic
  - Need to carry en bloc whenever coefficients become too large

Example: product-scanning multiplication

```
/* 256-bit integers in radix 2^16 */
typedef signed long long bigint[16];

void mul_prodscan(signed long long r[31],
                  const bigint x,
                  const bigint y)
{
  r[0]   = x[0] * y[0];
  r[1]   = x[1] * y[0];
  r[1]  += x[0] * y[1];
  r[2]   = x[2] * y[0];
  r[2]  += x[1] * y[1];
  r[2]  += x[0] * y[2];
  ...
  r[29]  = x[15] * y[14];
  r[29] += x[14] * y[15];
  r[30]  = x[15] * y[15];
}
```

Modular reduction

- We don't just need arithmetic on big integers
- We need arithmetic in finite fields
- In other words, we need reduction modulo a prime p
- Let's fix some p, say $p = 2^{255} - 19$
- We know that $2^{255} \equiv 19 \pmod{p}$
- This means that $2^{256} \equiv 38 \pmod{p}$
- Reduce 31-bit intermediate result `r` as follows:

```
for(i=0;i<15;i++)
  r[i] += 38*r[i+16];
```

- Result is in `r[0]`,...,`r[15]`

Primes are not rabbits

- "You cannot just simply pull some nice prime out of your hat!"
- In fact, very often we can.
- For cryptography we construct curves over fields of "nice" order
- Examples:
  - $2^{192} - 2^{64} - 1$ ("NIST-P192", FIPS186-2, 2000)
  - $2^{224} - 2^{96} + 1$ ("NIST-P224", FIPS186-2, 2000)
  - $2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ ("NIST-P256", FIPS186-2, 2000)
  - $2^{255} - 19$ (Bernstein, 2006)
  - $2^{251} - 9$ (Bernstein, Hamburg, Krasnova, Lange, 2013)
  - $2^{448} - 2^{224} - 1$ (Hamburg, 2015)
- All these primes come with (more or less) fast reduction algorithms
- More about *general primes* later
- For the moment let's stick to $2^{255} - 19$

Carrying after multiplication

```
long long c;
for(i=0;i<15;i++)
{
  c = r[i] >> 16;
  r[i+1] += c;
  c <<= 16;
  r[i] -= c;
}
c = r[15] >> 16;
r[0] += 38*c;
c <<= 16;
r[15] -= c;
```

- Coefficient `r[0]` may still be too large: carry again to `r[1]`

How about squaring?

```
#define bigint_square(R,X) bigint_mul(R,X,X)
```

How about squaring?

```
/* 256-bit integers in radix 2^16 */
typedef signed long long bigint[16];

void square_prodscan(signed long long r[31],
                     const bigint x)
{
  r[0]   = x[0] * x[0];
  r[1]   = x[1] * x[0];
  r[1]  += x[0] * x[1];
  r[2]   = x[2] * x[0];
  r[2]  += x[1] * x[1];
  r[2]  += x[0] * x[2];
  ...
  r[29]  = x[15] * x[14];
  r[29] += x[14] * x[15];
  r[30]  = x[15] * x[15];
}
```

How about squaring?

```
/* 256-bit integers in radix 2^16 */
typedef signed long long bigint[16];

void square_prodscan(signed long long r[31],
                     const bigint x)
{
  signed long long _2x[16];
  int i;
  for(i=0;i<16;i++)
    _2x[i] = 2*x[i];

  r[0]   =   x[0] * x[0];
  r[1]   = _2x[1] * x[0];
  r[2]   = _2x[2] * x[0];
  r[2]  +=   x[1] * x[1];
  ...
  r[29]  = _2x[15] * x[14];
  r[30]  = x[15] * x[15];
}
```

Squaring vs. multiplication

Multiplication needs

- 256 multiplications
- 225 additions

Squaring needs

- 136 multiplications
- 105 additions
- 15 additions or shifts or multiplications by 2 for precomputation

How about other prime fields?

- So far: reductions only modulo "nice" primes
- What if somebody just throws an ugly prime at you?
- Example: German BSI is pushing the "Brainpool curves", over fields $\mathbb{F}_p$ with

$$\begin{aligned} p_{224} = & 2272162293245435278755253799591092807334073\backslash \\ & 2145944992304435472941311 \\ = & 0xD7C134AA264366862A18302575D1D787B09F07579\backslash \\ & 7DA89F57EC8C0FF \end{aligned}$$

or

$$\begin{aligned} p_{256} = & 7688495639704534422080974662900164909303795\backslash \\ & 0200943055203735601445031516197751 \\ = & 0xA9FB57DBA1EEA9BC3E660A909D838D726E3BF623D\backslash \\ & 52620282013481D1F6E5377 \end{aligned}$$

- Another example: Pairing-friendly curves are typically defined over fields $\mathbb{F}_p$ where p has *some* structure, but hard to exploit for fast arithmetic

Montgomery representation

- We have the following problem:
  - We multiply two n-limb big integers and obtain a $2n$-limb result t
  - We need to find $t \mod p$
- Idea: Perform big-integer division with remainder (expensive!)
- Better idea (Montgomery, 1985):
  - Let R be such that $\gcd(R, p) = 1$ and $t < p \cdot R$
  - Represent an element a of $\mathbb{F}_p$ as $aR \mod p$
  - Multiplication of aR and bR yields $t = abR^2$ ($2n$ limbs)
  - Now compute *Montgomery reduction*: $tR^{-1} \mod p$
  - For *some* choices of R this is more efficient than division
  - Typical choice for radix-b representation: $R = b^n$

Montgomery reduction (pseudocode)

Require: $p = (p_{n-1}, \ldots, p_0)_b$ with $\gcd(p, b) = 1$, $R = b^n$,
$p' = -p^{-1} \mod b$ and $t = (t_{2n-1}, \ldots, t_0)_b$
Ensure: $tR^{-1} \mod p$

```
A ← t
for i from 0 to n − 1 do
    u ← a_i p' mod b
    A ← A + u · p · b^i
end for
A ← A / b^n
if A ≥ p then
    A ← A − p
end if
return A
```

Some notes about Montgomery reduction

- Some cost for transforming to Montgomery representation and back
- Only efficient if many operations are performed in Montgomery representation
- The algorithms takes $n^2 + n$ multiplication instructions
- n of those are "shortened" multiplications (modulo b)
- The cost is roughly the same as schoolbook multiplication
- Careful about conditional subtraction (timing attacks!)
- One can merge schoolbook multiplication with Montgomery reduction: "Montgomery multiplication"

Still missing: inversion

- Inversion is typically *much* more expensive than multiplication
- Efficient ECC arithmetic avoids frequent inversions
- ECC can typically not avoid *all* inversions
- We need inversion, but we do (usually) not need it often
- Two approaches to inversion:
  1. Extended Euclidean algorithm
  2. Fermat's little theorem

Extended Euclidean algorithm

- Given two integers a, b, the Extended Euclidean algorithm finds
  - The greatest common divisor of a and b
  - Integers u and v, such that $a \cdot u + b \cdot v = \gcd(a, b)$
- It is based on the observation that

$$\gcd(a, b) = \gcd(b, a - qb) \quad \forall q \in \mathbb{Z}$$

- To compute $a^{-1} \pmod{p}$, use the algorithm to compute

$$a \cdot u + p \cdot v = \gcd(a, p) = 1$$

- Now it holds that $u \equiv a^{-1} \pmod{p}$

Extended Euclidean algorithm (pseudocode)

Require: Integers a and b.
Ensure: An integer tuple (u, v, d) satisfying $a \cdot u + b \cdot v = d = \gcd(a, b)$

$u \leftarrow 1$
$v \leftarrow 0$
$d \leftarrow a$
$v_1 \leftarrow 0$
$v_3 \leftarrow b$
while $(v_3 \neq 0)$ **do**
  $q \leftarrow \lfloor \frac{d}{v_3} \rfloor$
  $t_3 \leftarrow d \mod v_3$
  $t_1 \leftarrow u - qv_1$
  $u \leftarrow v_1$
  $d \leftarrow v_3$
  $v_1 \leftarrow t_1$
  $v_3 \leftarrow t_3$
end while
$v \leftarrow \frac{d-au}{b}$
return (u, v, d)

Some notes about the Extended Euclidean algorithm

- Core operation are divisions with remainder
- This lecture: no details about big-integer division
- Version without divisions: **binary extended gcd**:

  Handbook of applied cryptography, Alg. 14.61
- The running time (number of loop iterations) depends on the inputs
- We usually do not want this for cryptography (timing attacks!)
- Possible protection: blinding
  - Multiply a by random integer r
  - Invert, obtain $r^{-1}a^{-1}$
  - Multiply again by r to obtain a^{-1}
- Note that this requires a source of randomness

Fermat's little theorem

Theorem

Let p be prime. Then for any integer a it holds that $a^{p-1} \equiv 1 \pmod{p}$

- This implies that $a^{p-2} \equiv a^{-1} \pmod{p}$
- Obvious algorithm for inversion: Exponentiation with $p - 2$
- The exponent is quite large (e.g., 255 bits), is that efficient?
- Yes, fairly:
  - Exponent is fixed and known at compile time
  - Can spend quite some time on finding an efficient addition chain (next lecture)
  - Inversion modulo $2^{255} - 19$ needs 254 squarings and 11 multiplications in $\mathbb{F}_{2^{255}-19}$

Inversion in $\mathbb{F}_{2^{255}-19}$

```
void gfe_invert(gfe r, const gfe x)
{
gfe z2, z9, z11, z2_5_0, z2_10_0, z2_20_0, z2_50_0, z2_100_0, t;
int i;
/* 2 */                gfe_square(z2,x);
/* 4 */                gfe_square(t,z2);
/* 8 */                gfe_square(t,t);
/* 9 */                gfe_mul(z9,t,x);
/* 11 */               gfe_mul(z11,z9,z2);
/* 22 */               gfe_square(t,z11);
/* 2^5 - 2^0 = 31 */ gfe_mul(z2_5_0,t,z9);
/* 2^6 - 2^1 */        gfe_square(t,z2_5_0);
/* 2^10 - 2^5 */       for (i = 1;i < 5;i++) { gfe_square(t,t); }
/* 2^10 - 2^0 */       gfe_mul(z2_10_0,t,z2_5_0);
/* 2^11 - 2^1 */       gfe_square(t,z2_10_0);
/* 2^20 - 2^10 */      for (i = 1;i < 10;i++) { gfe_square(t,t); }
/* 2^20 - 2^0 */       gfe_mul(z2_20_0,t,z2_10_0);
/* 2^21 - 2^1 */       gfe_square(t,z2_20_0);
/* 2^40 - 2^20 */      for (i = 1;i < 20;i++) { gfe_square(t,t); }
/* 2^40 - 2^0 */       gfe_mul(t,t,z2_20_0);
```

Inversion in $\mathbb{F}_{2^{255}-19}$

```
/* 2^41 - 2^1 */      gfe_square(t,t);
/* 2^50 - 2^10 */     for (i = 1;i < 10;i++) { gfe_square(t,t); }
/* 2^50 - 2^0 */      gfe_mul(z2_50_0,t,z2_10_0);
/* 2^51 - 2^1 */      gfe_square(t,z2_50_0);
/* 2^100 - 2^50 */    for (i = 1;i < 50;i++) { gfe_square(t,t); }
/* 2^100 - 2^0 */     gfe_mul(z2_100_0,t,z2_50_0);
/* 2^101 - 2^1 */     gfe_square(t,z2_100_0);
/* 2^200 - 2^100 */   for (i = 1;i < 100;i++) { gfe_square(t,t); }
/* 2^200 - 2^0 */     gfe_mul(t,t,z2_100_0);
/* 2^201 - 2^1 */     gfe_square(t,t);
/* 2^250 - 2^50 */    for (i = 1;i < 50;i++) { gfe_square(t,t); }
/* 2^250 - 2^0 */     gfe_mul(t,t,z2_50_0);
/* 2^251 - 2^1 */     gfe_square(t,t);
/* 2^252 - 2^2 */     gfe_square(t,t);
/* 2^253 - 2^3 */     gfe_square(t,t);
/* 2^254 - 2^4 */     gfe_square(t,t);
/* 2^255 - 2^5 */     gfe_square(t,t);
/* 2^255 - 21 */      gfe_mul(r,t,z11);
}
```

Multiprecision libraries

- Why would you write low-level arithmetic yourself?
- Aren't there some good libraries for this?
- There are:
  - GMP (`http://gmplib.org`), high-performance arithmetic on multiprecision numbers
  - NTL (`http://shoup.net/ntl/`), number-theory library, higher level than GMP, uses GMP
  - OpenSSL Bignum (`http://openssl.org`), low-level routines in OpenSSL
  - mp$\mathbb{F}_q$ (`http://mpfq.gforge.inria.fr/`), a finite-field library (generator)

Limitations of libraries

- Libraries don't know the modulus (except for $\texttt{mp}\mathbb{F}_q$), cannot optimize for a fixed modulus
- Libraries don't know the sequence of field operations you're computing (e.g., point addition), cannot use lazy reduction
- Libraries are not always timing-attack protected
- Consequence: ECC speed records are achieved with hand-optimized assembly implementations

Part II
Elliptic-curve cryptography
from a crypto-engineering perspective

Diffie-Hellman

- Let G be a cyclic, finite, abelian Group (written additively) and let P be a generator of G
- Alice chooses random $a \in \{0, \dots, |G|-1\}$, computes aP, sends to Bob
- Bob chooses random $b \in \{0, \dots, |G|-1\}$, computes bP, sends to Alice
- Alice computes joint key $a(bP)$
- Bob computes joint key $b(aP)$
- DLP in G: given $kP \in G$ and P, find k
- Solving the DLP breaks security of Diffie-Hellman

Groups with hard DLP

- Traditional answer: $\mathbb{Z}_p^*$ with large prime-order subgroup
- Modern answer: Elliptic curve over $\mathbb{F}_q$ with large prime-order subgroup
- Sophisticated answer (not in this lecture): hyperelliptic curves of genus 2

Typical view on elliptic curves

Definition

Let K be a field and let $a_1, a_2, a_3, a_4, a_6 \in K$. Then the following equation defines an elliptic curve E:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

if the discriminant Δ of E is not equal to zero. This equation is called the *Weierstrass form* of an elliptic curve.

Characteristic $\neq 2, 3$

If $char(K) \neq 2, 3$ we can use a simplified equation:

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

Characteristic 2

If $char(K) = 2$ we can (usually) use a simplified equation:

$$E : y^2 + xy = x^3 + ax^2 + b$$

Rational points

Setup for cryptography

- Choose $K = \mathbb{F}_q$
- Consider the set of $\mathbb{F}_q$-rational points:

$$E(\mathbb{F}_q) = \{(x, y) \in \mathbb{F}_q \times \mathbb{F}_q : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6\} \cup \{\mathcal{O}\}$$

- The element $\mathcal{O}$ is the "point at infinity"
- This set forms a group (together with addition law)
- Order of this group: $|E(\mathbb{F}_q)| \approx |\mathbb{F}_q|$

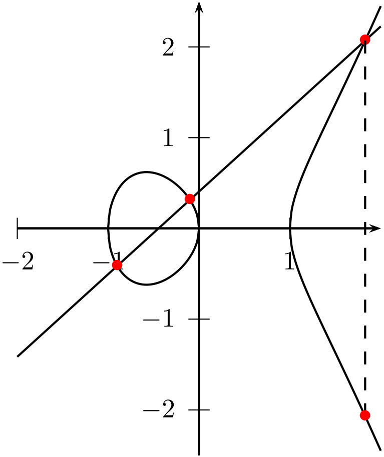
The group law

Example curve: $y^2 = x^3 - x$ over $\mathbb{R}$

Addition of points

- Add points $P = (-0,9; -0,4135)$ and $Q = (-0,1; 0,3146)$
- Compute line through the two points
- Determine third intersection $T = (x_T, y_T)$ with the elliptic curve
- Result of the addition: $P + Q = (x_T, -y_T)$

Graph of E over $\mathbb{R}$



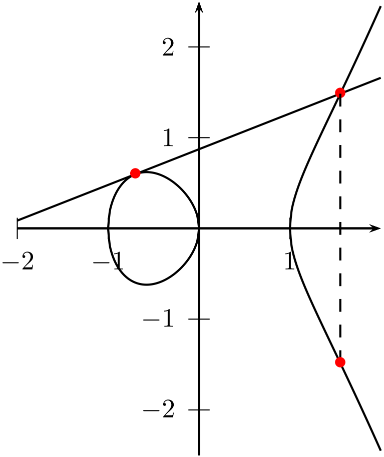
The group law

Example curve: $y^2 = x^3 - x$ over $\mathbb{R}$

Point doubling

- Double the point $P = (-0.7, 0.5975)$
- Compute the tangent on P
- Determine second intersection $T = (x_T, y_T)$ with the elliptic curve
- Result of the addition: $P + Q = (x_T, -y_T)$

Graph of E over $\mathbb{R}$



Group law in formulas

Curve equation: $y^2 = x^3 + ax + b$

Point addition

- $P = (x_P, y_P), Q = (x_Q, y_Q) \to P + Q = R = (x_R, y_R)$ with
- $x_R = \left(\frac{y_Q - y_P}{x_Q - x_P}\right)^2 - x_P - x_Q$
- $y_R = \left(\frac{y_Q - y_P}{x_Q - x_P}\right)(x_P - x_R) - y_P$

Point doubling

- $P = (x_P, y_P), 2P = (x_R, y_R)$ with
- $x_R = \left(\frac{3x_P^2 + a}{2y_P}\right)^2 - 2x_P$
- $y_R = \left(\frac{3x_P^2 + a}{2y_P}\right)(x_P - x_R) - y_P$

More Weierstrass curve group law

- Neutral element is $\mathcal{O}$
- Inverse of a point (x, y) is $(x, -y)$
- Note: Formulas don't work for $P + (-P)$, also don't work for $\mathcal{O}$
- Need to distinguish these cases!
- "Uniform" addition law in Hışıl's Ph.D. thesis, Section 5.5.2 (http://eprints.qut.edu.au/33233/):
  - Move special cases to other points
  - Not safe to use on arbitrary input points!
- Formulas for curves over $\mathbb{F}_{2^k}$ look slightly different, but same special cases

Finding a suitable curve

Security requirements for ECC

- $\ell = |E(\mathbb{F}_q)|$ must have large prime-order subgroup
- For n bits of security we need $2n$-bit prime-order subgroup
- Impossible to transfer DLP to less secure groups:
  - ℓ must not be equal to q
  - We need $\ell \nmid p^k - 1$ for small k

Finding a curve

- Fix finite field $\mathbb{F}_q$ of suitable size
- Fix curve parameter a (quite common: $a = -3$)
- Pick curve parameter b until E fulfills desired properties
- This requires efficient "point counting"
- This requires efficient factorization or primality proving

Standardized curves

"The nice thing about standards is that you have so many to choose from. " *– Andrew S. Tanenbaum*

- Various standardized curves, most well-known: NIST curves:
  - Big-prime field curves with 192, 224, 256, 384, and 521 bits
  - Binary curves with 163, 233, 283, 409, and 571 bits
  - Binary Koblitz curves with 163, 233, 283, 409, and 571 bits
- SECG curves (Certicom), prime-field and binary curves
- Brainpool curves (BSI), only prime-field curves
- FRP256v1 (ANSSI), one prime-field curve (256 bits)

Binary vs. big prime

Curves over big-prime fields

- Many fields of a given size $\Rightarrow$ many curves
- Efficient in software (can use hardware multipliers)
- Less efficient in hardware

Curves over binary fields

- Important for security: exponent k in $\mathbb{F}_{p^k}$ has to be prime
- Not many fields (not that many curves)
- More efficient in hardware
- Efficient in software only on some microarchitectures
- A hell to implement securely in software on some other microarchitectures

Putting it together

- Choose security level (e.g., 128 bits)
- Decide whether you want binary or big-prime field arithmetic, let's say big prime
- Pick corresponding standard curve, e.g., NIST-P256
- Implement field arithmetic
- Implement ECC addition and doubling
- Implement scalar multiplication (next lecture)
- You're done with BAD (!) ECDH software

Problem I: inversions

Inversions

- Adding $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ needs an inversion in $\mathbb{F}_q$
- Inversions are expensive
- Constant-time inversions are even more expensive

Solution: projective coordinates

- Store fractions of elements of $\mathbb{F}_q$, invert only once at the end
- Represent points in *projective coordinates*: $P = (X_P : Y_P : Z_P)$ with $x_P = X_P/Z_P$ and $y_P = Y_P/Z_P$
- The point $(1 : 1 : 0)$ is the point at infinity
- Also possible: weighted projective coordinates:
  - Jacobian coordinates: $P = (X_P : Y_P : Z_P)$ with $x_P = X_P/Z_P^2$ and $y_P = Y_P/Z_P^3$
  - López-Dahab coordinates (for binary curves): $P = (X_P : Y_P : Z_P)$ with $x_P = X_P/Z_P$ and $y_P = Y_P/Z_P^2$
- Important: Never *send* projective representation, always convert to affine!

Problem II: group-law special cases

- Addition of $P + Q$ needs to distinguish different cases:
  - If $P = \mathcal{O}$ return Q
  - Else if $Q = \mathcal{O}$ return P
  - Else if $P = Q$ call doubling routine
  - Else if $P = -Q$ return $\mathcal{O}$
  - Else use addition formulas
- Similar for doubling P:
  - If $P = \mathcal{O}$ return P
  - Else if $y_P = 0$ return $\mathcal{O}$
  - Else use doubling formulas
- Constant-time implementations of this are horrible
- Good news: Can avoid the checks when computing $k \cdot P$ and $k < |E(\mathbb{F}_q)|$
- Bad news: Side-channel countermeasures use $k > |E(\mathbb{F}_q)|$
- More bad news: Doesn't work for multi-scalar multiplication (next lecture)
- Baseline: *simple* implementations are likely to be wrong or insecure

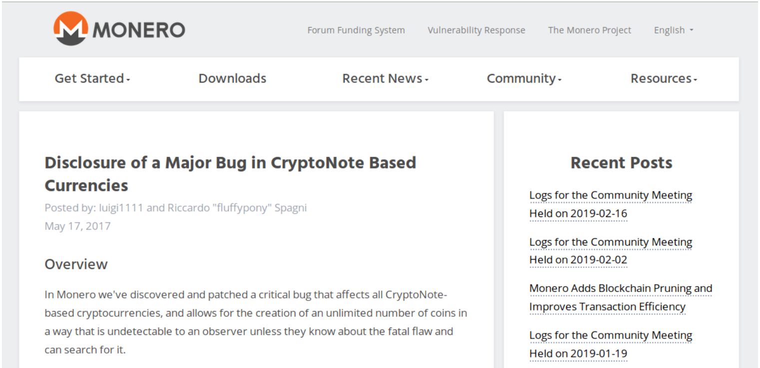
Solution I: Montgomery ladder

- Use Montgomery curve: $E_M : By^2 = x^3 + Ax^2 + x$.
- Use x-coordinate-only differential addition chain ("Montgomery ladder", next lecture)
- Advantages:
  - Works on all inputs, no special cases
  - Very regular structure, easy to protect against timing attacks
  - Point compression/decompression for free
  - Easy to implement, harder to screw up in hard-to-detect ways
  - Simple implementations are likely to be correct and secure
- Disadvantages:
  - Not all curves can be converted to Montgomery shape
  - Always have a cofactor of at least 4
  - Ladders on general Weierstrass curves are much less efficient
  - We only get the x coordinate of the result, tricky for signatures
  - Can reconstruct y, but that involves some additional cost

Solution II: (twisted) Edwards curves

- Edwards, 2007: New form for elliptic curves ("Edwards curves")
- Bernstein, Lange, 2007: very fast addition and doubling on these curves
- Bernstein, Birkner, Joye, Lange, Peters, 2008: generalize the idea to "twisted Edwards curves"
- Core advantage of (twisted) Edwards curves: **complete group law**
- No need to handle special cases
- No "point at infinity" to work with
- Can speed up doubling, but addition formulas work for $P + P$
- Efficient (for cryptography) transformation from Weierstrass to (twisted) Edwards only for some curves
- Always efficient: transformation between Montgomery curves and twisted Edwards curves
- Again: simple implementations are likely to be correct and secure
- Disadvantage: always have a cofactor of at least 4

So, what's the deal with the cofactor?



- Protocols need to be careful to avoid subgroup attacks
- Monero screwed this up, which allowed double-spending
- Elegant solution: "Ristretto" encoding by Hamburg, see: https://github.com/otrv4/libgoldilocks

Solution III: Complete group law on Weierstrass curves

- Bosma, Lenstra, 1995: complete group law for Weierstrass curves
- Problem: Extremely inefficient
- Renes, Costello, Batina, 2016: Much faster complete group law for Weierstrass curves
- Somewhat less efficient than (twisted) Edwards
- Covers all curves

Problem III: Wrong-curve attacks

ECDH attack scenario

- Alice sends point on different (insecure) curve with small subgroup
- Bob computes "shared key" in that small subgroup
- Alice learns "shared key" through brute force
- Alice learns Bob's secret scalar modulo the order of the small subgroup

Countermeasures

- Check that input point is on the curve (functional tests will miss this!)
- Send compressed points $(x, \text{parity}(y))$; decompression returns (x, y) on the curve or fails
- Send only x (Montgomery ladder); but: x could still be on the "twist" of E
- Make sure that the twist is also secure ("twist security")

Problem IV: Backdoors in standards?

“"I no longer trust the [NIST Elliptic Curves] constants. I believe the NSA has manipulated them through their relationships with industry.” – Bruce Schneier, 2013.

- It is pretty clear that NSA put a backdoor in Dual_EC_DRBG
- Constants of NIST curves have been obtained by hashing random values
- No-backdoor claim: We know the preimages
- Possible attack if you know a class of vulnerable curves: Generate random seeds until you have found a vulnerable (and seemingly secure) curve
- Fact: There are no known insecurities of NIST curves
- Fact: There is no proof that there are no intentional vulnerabilities in NIST curves
- For more details, see BADA55 elliptic curves

Choosing a safe curve

Overview of various elliptic curves and thorough security analysis by Bernstein and Lange:

https://safecurves.cr.yp.to

(doesn't list cofactor-1 curves, so best to combine with Ristretto)

Point representation and arithmetic

Collection of elliptic-curve shapes, point representations and group-operation formulas by Bernstein and Lange:

https://www.hyperelliptic.org/EFD/