Sec256k1 in Swift

https://github.com/efz/sec256k1

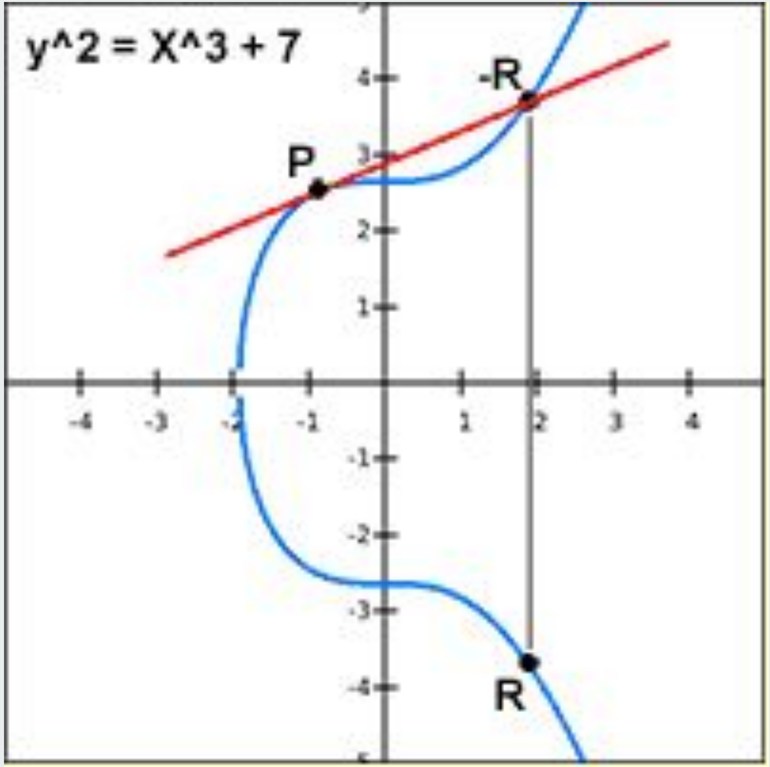

Image: https://steemit.com/ellipticcurve/@sso/calculate-bitcoin-publickey

Sec256k1 public & private keys

- (a, a*G) => private/public key pair.
- G => special point on the curve
- a => Private key => random 256 bit unsigned integer
- a * G = G + G + G...+ G => public key
- Very easy to create session keys,
  - (a, a*G) => A's keys
  - (b, b*G) => B's keys
  - b*(a*G) = (a*b)*G = a* (b*G) => Session key
- 2 prime modulo number systems
  - Coordinate system - field
  - Private key - scalar (based on order of point G)

SWIFT implementation

- Use Swift Structs.
- 4 x 64 bit integers to represent 256 bit numbers
  - C version used 5 x 52, 10 x 26, etc, encodings for field integers. Use GCC 128 bit int.
- Single Group point representation in 3 coordinates (X, Y, Z)
- All numbers represented as normalized modulo P numbers ($0 <= n < P$)

Benchmark

	C	SWIFT
Add scalar/field	0.009	0.006
Mul scalar (field)	0.03 (0.02)	0.03 (0.02)
Inverse scalar (field)	11.1 (4.7)	8.5 (4.8)
Point Add z == 1 (z != 1)	0.22 (0.26)	0.19 (0.25)
Point Double	0.12	0.12
Sha256	0.20	0.22
Sign	36 (not comparable)	28.9
Verify	61.3	63.2

Performance improvements

- Swift Tuples instead of fixed size arrays
- Inlining small routines
- Module exponents calculation
- Fused operations
  - E.g. multiple additions, small integer multiplications, etc. before reduction
  - Numbers doesn’t need full 256bit range
- Avoid modulo division
- a*G computation with precomputed table
- b*Y computation
- b*Y + a*G calculation

Modulo exponent

- A^(P-2) => modulo inverse, A^(P+1/4) => modulo square root for field P
- Naive method,
  - A^27 = A * A * A...* A => 26 Operations
- By expanding exponent in binary,
  - A^(11011) = A^(2^8 + 2^4 + 2^1 + 2^0) = (A^16) * (A^8) * (A^2)* (A^1) => 7 Operations
- Reusing common segments of 1s (When exponent is known.),
  - A^11 => 2 operations
  - A^11000 = (A^11) ^8 = square (A^11) 3 times
  - A^11011 = (A^11000) * (A^11) => 6 Operations => 1 Operation saving.
  - Larger operation count reductions when exponent has large segments of 1s as in primes used in sec256k1

Reduction and fused operations

- A (256 bit) * B (256 bit) => R (512 bit)
- Reduce R back to 256 bits and 0 <= R < P
- R = m * 2^256 + r = m *(2^256 - P) + r = m * (P's 2s complement) + r
- Longs segment of leading 1's in P => lots of leading 0's in ~P
- A * B + C * D => two 512 bits to 256 bit reductions
  - Partially reduce A*B and C*D
  - Add partially reduced A*B and C*D and reduce final result to 256bits
- Similarly partially reduces results can be multiplied by small integers, etc.

a*G Computation

- Precomputed table.
- 4 bit table example
  - (0001) * G, (0010) * G, (0011) * G, … (1111) * G
  - (0001 0000) * G, (0010 0000) * G, (0011 0000) * G, … (1111 0000) * G
  - …
  - a = … 0011 0010
  - a * G = … (0011 0000) * G + (0010) * G
- Space Vs pre-computation time Vs final calculation time
  - 4 bit => 16 * 64 entries => 1024 precomputed values, 64 additions
  - 8 bit =>256 * 32 entries => 8192 precomputed values. 32 additions

Avoid modulo division

- Modulo division is expensive. Only do it in last step in multi step computations.
- Keep divisor in separate Z coordinate. (x, y, z)
- (x, y) => (x, y, 1)
- Curve equation:
  - y^2 = x^3 + 7 => y^2 = x^3 + 7* z^6
- (x, y, z) => (x/z^2, y/z^3, 1) => (x, y)

b*T Computation

- Point T is unknown until runtime.
- Decompose “b” in binary. Double T in each step and add to the result.
  - 27 * T => (11011) * T => (2^5) * T + (2^4) * T + (2^1) * T + (2^0) * T
  - Max about 255 + 255 operations
- Full multiplication table computation need more operations than (255 + 255)
- Build partial 4bit table
  - (0001) * T, (0010)*T, …, (1111)*T => 16 entries
  - Only need to keep leading bit 1 entries. (1000)*T , (1001)*T,.., (1111)*T
- Scan “b” bit pattern from most significant end for 1s and extract precomputation table keys.
  - About 255 doubling operations + max 67 additions + 16 additions for precomputation => max about 367 ops

b*T + a*G Computation

- Compute b*T and a*G separately and add.
  - About 367 operations for b*T + 64 operations for a * G
- 64 operations for a*G can be reduced in most cases by following the approach used in b*T computation.
- Exploit associativity,
  - b*T+ a*G = (T + T ...+ T) + (G + G +...+ G) = (T + G) + (2*T+ G) + ... T + G + G ...
- 256 doubling operation anyways required for b*T. No need to do any more doubling.
  - Parallel scan “b” and “a” bit patterns for segments start with 1.
  - Do required number of doubling on result before adding table lookup values.

Few interesting observations

- Swift compiler/optimizer doesn't do aggressive inlining.
    - Explicite inline annotations on small routines make big difference in runtime.
- Swift tuples are much faster than fixed size arrays.
    - Assume swift array boundary checks makes array accesses slower compared to C
- Structs in Swift are very performant and easy to use.
    - Pass by value, cow
- Manual code tweaking can make big difference in speed.
    - Sha256 implementation.
- Wider performance difference between C & Swift libraries in Linux vs MacOs
    - 5% difference in verify benchmark in MacOs
    - 20% difference in verify benchmark in Linux
    - 27% difference in verify benchmark in Linux with assembly optimization