Elliptic Curve Cryptography

Fast implentation of the Diffie-Hellman key exchange

M. Gollub C. Heiniger T. Rubeli H. Zhao

ETH

May 31, 2016

Elliptic Curve

- An elliptic curve $E(\mathbb{F}_p)$ consists of the set of the points $P(x, y)$, $x, y \in \mathbb{F}_p$ satisfying

$$y^2 \equiv x^3 + ax + b \pmod{p}$$

- Possible to define an addition rule to add points on E

Diffie Hellman key exchange

Public parameters	
$y^2 \equiv x^3 + ax + b \pmod{p}$	
$a, b \in \mathbb{F}_p$, a prime p and a base point G are known	
Private computations	
Alice	**Bob**
Compute $P = aG$	Compute $Q = bG$
Public exchange of values	
Alice $\xrightarrow{\quad P \quad}$ **Bob**	**Alice** $\xleftarrow{\quad Q \quad}$ **Bob**
Further private computations	
Alice	**Bob**
Compute aQ	Compute bP
The shared secret is $aQ = a(bG) = b(aG) = bP$	

Adaption of Table 2.2 J. Hoffstein et al., *An Introduction to Mathematical Cryptography*

Double-and-add-Method

- Input $P \in E(F_p)$, $d \in \mathbb{Z}$
- Output: $d \cdot P \in E(F_p)$

```
N <- P
Q <- O
for i from 0 to m do
  if d_i = 1 then
    Q <- point_add(Q, N)
  N <- point_double(N)
return Q
```

where $d = d_0 + d_1 2 + \ldots + d_m 2^m \quad d_i \in \{0, 1\}$

https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication#Double-and-add

Implementation

- Bigint

```
typedef uint64_t block;

typedef struct
{
    uint64_t significant_blocks;
    block blocks[BIGINT_BLOCKS_COUNT];
} __BigInt;
```

Corresponding operations (Addition, Multiplication, Division, ...)

- Elliptic Curve and ECDH
  - Elliptic Curve definition and key exchange mechanism
  - 5 predefined curves : 192, 224, 256, 384, 521 bits

Cost Anaylsis

- Index integer operation matters
- Cost measure
  - $C = C_{\text{add}} + C_{\text{mult}} + C_{\text{shift}}$
  - Code generated operations counts

Optimizations

Stages

Baseline 1 - Implementation without memory optimization

Performance

Baseline 2 - Implementation with memory optimization

Comparison with OpenSSL

Algorithm

Optimal 1 - Algorithmic changes, jacobian coordinates

Performance

Optimal 2 - Final performance optimization

Intel ADX

C Code

```
low_m1 = _mulx_u64(a->blocks[i], b, &hi_m1);
add_carry_m1 = _addcarryx_u64(add_carry_m1, carry_m1, low_m1, &temp_m1);
add_carry_1 = _addcarryx_u64(add_carry_1, res->blocks[i], temp_m1, tmp->blocks[i]);
carry_m1 = hi_m1;
```

Created Assembly code

x86 icc 13.0.1 -m64 -march=haswell -O3

```
mov     rdx, QWORD PTR [48+rsi]
mulx    rdx, rbx, rax
adox    rbp, rdx
adcx    rbp, QWORD PTR [48+rdi]
mov     QWORD PTR [2288+r9], rbp
```

http://gcc.godbolt.org/

Intel ADX

C Code

```
low_m1 = _mulx_u64(a->blocks[i], b, &hi_m1);
add_carry_m1 = _addcarryx_u64(add_carry_m1, carry_m1, low_m1, &temp_m1);
add_carry_1 = _addcarryx_u64(add_carry_1, res->blocks[i], temp_m1, tmp->blocks[i]);
carry_m1 = hi_m1;
```

Created Assembly code

x86 gcc 5.3 -m64 -march=haswell -O3

```
mulx    48(%rsi), %r9, %r10
addq    %r9, %r11
movq    %r10, %r9
setc    %bpl
addb    $-1, %bl
adcq    48(%rdi), %r11
movq    %r11, 2288(%rax)
setc    %bl
addb    $-1, %bpl
```

http://gcc.godbolt.org/

ADX vs AVX2

- Bottleneck operation: BigInt - block multiplication
- Unavoidable dependecies in carry chain -> vectorization by processing 4 multiplications in parallel

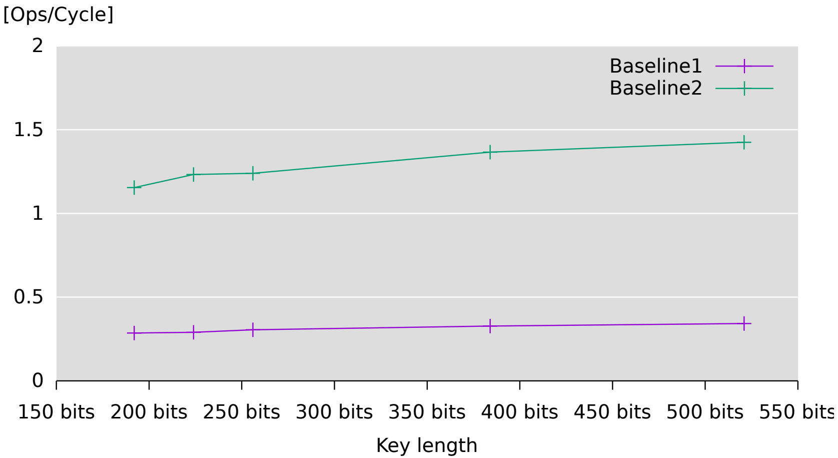
Approach	**Lower bound**	**Bottleneck**
ADX	8 cycles/iteration	ADX throughput
AVX (base 32) AVX2 (base 64)	10 cycles/iteration 24 cycles/iteration	Emulation of carry flag

- Further AVX2 downsides
  - higher mul latencies
  - unfriendly data layout
  - multiplications not always independent

Experiment result

- Environment
  - Platform:Arch Linux 64 bit, GCC 6.1.1 compiler
  - Skylake i7-6600U CPU @ 3GHz
  - 64 bit multiplication (mul, mulx): 1 op/cycle
  - 64 bit addition/subtraction (add, sub): 4 op/cycle
  - 64 bit addition with carry (adc, adcx, adox): 1 op/cycle
  - Carry addition only: peak performance of 2 ops/cycle 6 Gflops/s on 1 core

Performance plot Part 1



Performance plot Part 2