Elliptic Curve Cryptography
Fast implentation of the Diffie-Hellman key exchange

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

ETH

June 1, 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: $d \cdot P \in E(F_p)$

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

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

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

Implementation

▶ Bigint

```
1   typedef uint64_t block;
2
3   typedef struct
4   {
5       uint64_t significant_blocks;
6       block blocks[BIGINT_BLOCKS_COUNT];
7   } __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
  ▶ $C = C_{add} + C_{mult} + C_{shift}$
  ▶ Code generated operations counts

Optimizations

Stages

Baseline - Implementation without memory optimization

Memory optimization - Implementation with memory optimization

Comparison with OpenSSL

Precomputation /Jacobian coordinates - Algorithmic changes, jacobian coordinates

Final - Final performance optimization

Performance

Algorithm

Performance

Optimizations

Overview

- ► Code Optimizations
    - ► combine instructions
    - ► change the base from 8 bit to 64 bit for the big integers
    - ► Precomputation of $2k \cdot G$ where $k \in \mathbb{N}$ and $G \in E$
    - ► change the base from 2 bit to 64 bit in the montgomery multiplication
    - ► Inlining functions
    - ► better memory allocation

- ► Algorithmic Optimizations
    - ► Introduction of Jacobian coordinates

Intel ADX

C Code

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

Created Assembly code
x86 icc 13.0.1 -m64 -march=CORE-AVX2 -O3

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

http://gcc.godbolt.org/

Intel ADX

C Code

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

Created Assembly code
x86 gcc 5.3 -m64 -march=haswell -O3

```
1   mulx    48(%rsi), %r9, %r10
2   addq    %r9, %r11
3   movq    %r10, %r9
4   setc    %bpl
5   addb    $-1, %bl
6   adcq    48(%rdi), %r11
7   movq    %r11, 2288(%rax)
8   setc    %bl
9   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)	10 cycles/iteration	Emulation of carry
AVX2 (base 64)	24 cycles/iteration	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
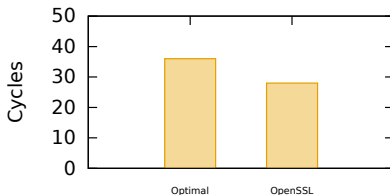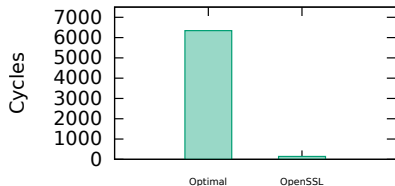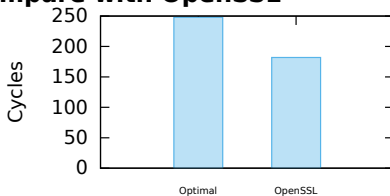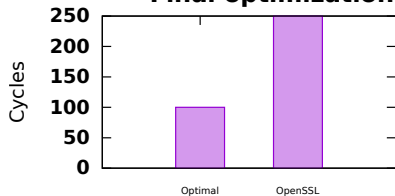
Speedup plot compared to Baseline

ECDH execution cycles comparison
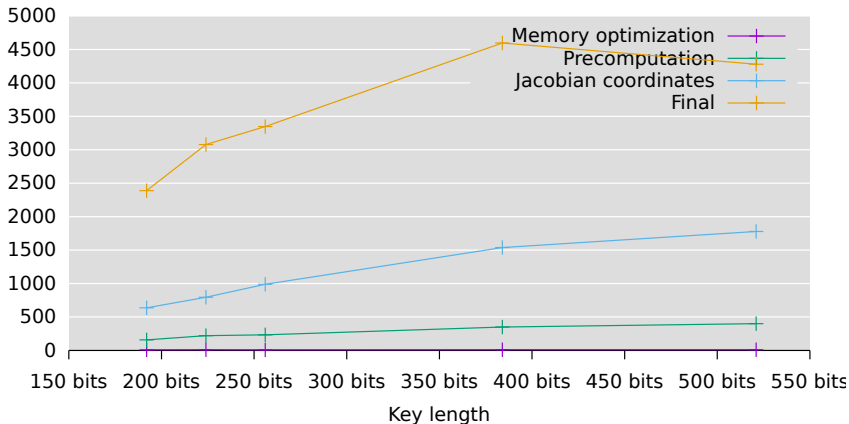
Final optimization compare with OpenSSL

Performance plot Part 1

[Ops/Cycle]

Algorithm
000

Implementation
0

Cost Analysis
0

Optimizations
00000

Results
00000●00
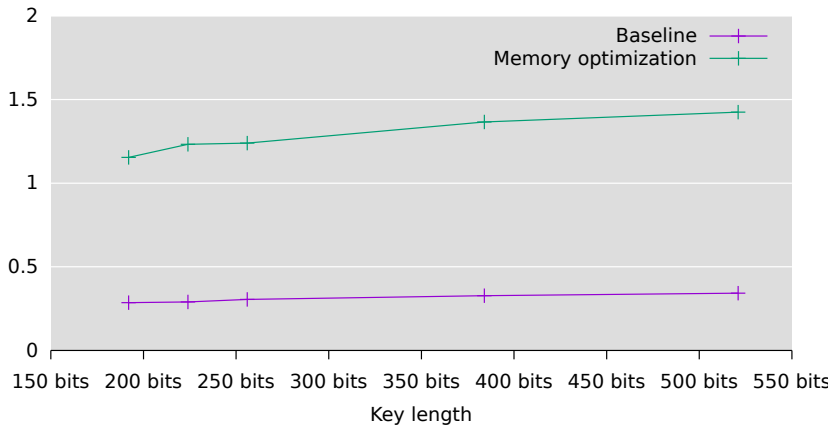
Performance plot Part 2

[Ops/Cycle]

Algorithm
000

Implementation
0

Cost Analysis
0

Optimizations
00000

Results
0000000●0
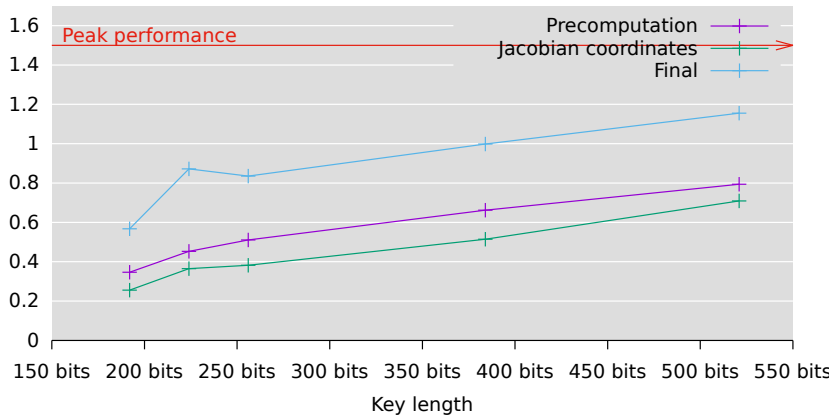
Performance plot operation counts

Key size	Baseline	Memory optimization	Precomputation	Jacobian coordinates	Final
192	2117237004	912515528	16277780	2977446	1585686
224	3666675252	1384425854	25816754	5796852	3035341
256	4901227919	2122930314	35137355	6206895	3257705
384	18873323391	7047549105	109331889	19264827	9848295
521	48749705798	18063182851	282776551	56794800	28765815

Speedup plot with varied private key form, compared to OpenSSL

Speedup/OpenSSL X

Key length