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** — Compute $P = uG$ **Bob** — Compute $Q = vG$
Public exchange of values **Alice** $\xrightarrow{P}$ **Bob**   **Alice** $\xleftarrow{Q}$ **Bob**
Further private computations **Alice** — Compute uQ **Bob** — Compute vP The shared secret is $uQ = u(vG) = v(uG) = vP$

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

Double-and-add-Method

- Input $P \in E(F_p)$, $d \in \mathbb{N}$ Complexity of $\mathcal{O}(log_2(d) \cdot log_2(p))$
- 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 $d = d_0 + d_1 2 + ... + d_m 2^m \quad d_i \in \{0, 1\}$

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 - Implementation without memory optimization

Memory optimization - Implementation with memory optimization

Comparison with OpenSSL

Jacobian coordinates - Algorithmic changes, jacobian coordinates

Final - Final performance optimization

Performance

Algorithm

Performance

Optimizations

Overview

- **Baseline**
  - Improved memory allocation
  - Change the base from 8 bit to 64 bit for the big integers
- **Memory optimization**
  - Change the base from 2 bit to 64 bit for Montogomery
  - Precomputation of $2^k \cdot G$ where $k \in \{1, \ldots, \log_2(p)\}$ and $G \in E$
  - Introduction of Jacobian coordinates
  - Vectorize shifting using AVX2
- **Jacobian coordinates**
  - function stiching
  - Inlining functions
  - Unrolling

Intel ADX

C Code

```c
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=CORE-AVX2 -O3

```asm
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

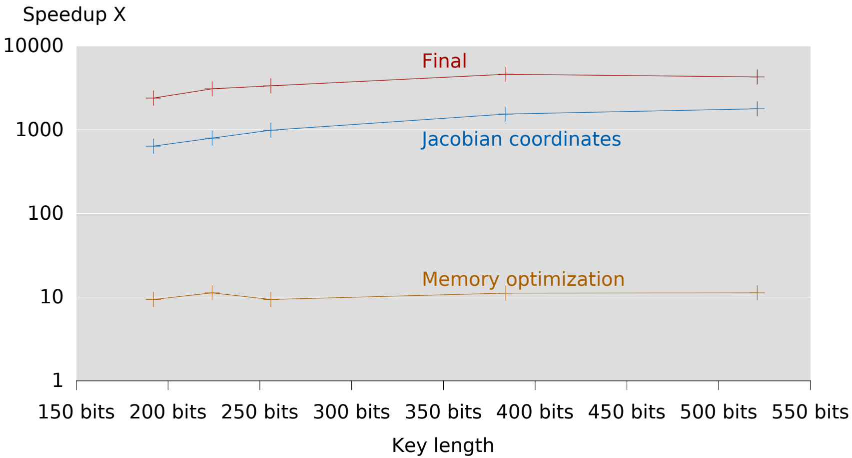
Approach	**Lower bound**	**Bottleneck**
ADX	8 cycles/iteration	ADX throughput
AVX2 (base 32) AVX2 (base 64)	10 cycles/iteration 24 cycles/iteration	Emulation of carry flag

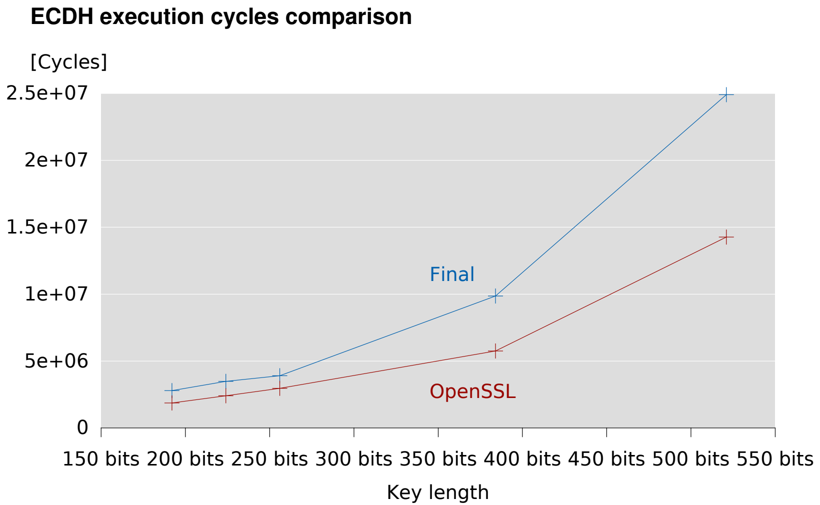
- Further AVX2 downsides
  - higher mul latencies
  - unfriendly data layout
  - multiplications not always independent

Experiment result

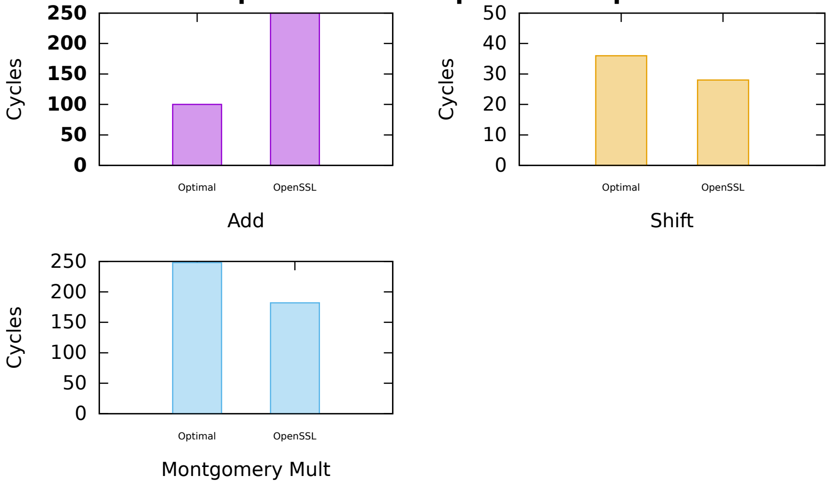
- Environment
  - Platform:Arch Linux 64 bit, GCC 6.1.1 compiler
  - Skylake i7-6600U CPU @ 3GHz
  - 64 bit multiplication (mul, mulx): 1 op/cycle
  - 64 bit addition/subtraction (add, sub): 4 op/cycle
  - 64 bit addition with carry (adc, adcx, adox): 1 op/cycle
  - Carry addition only: peak performance of 2 ops/cycle 6 Gops/s on 1 core
  - Compile flag: -O3 -mavx2 -mbmi2 -madx

Speedup plot compared to Baseline

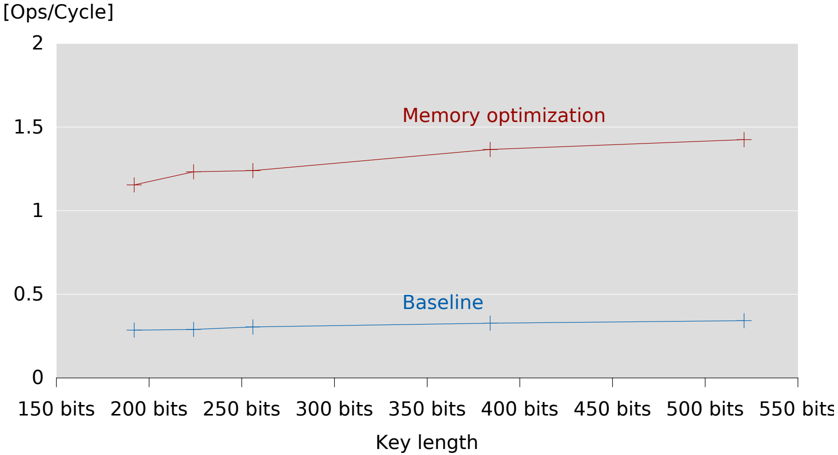



ECDH execution cycles comparison
[Cycles]
2.5e+07
2e+07
1.5e+07
1e+07
5e+06
0
Final
OpenSSL
150 bits 200 bits 250 bits 300 bits 350 bits 400 bits 450 bits 500 bits 550 bits
Key length

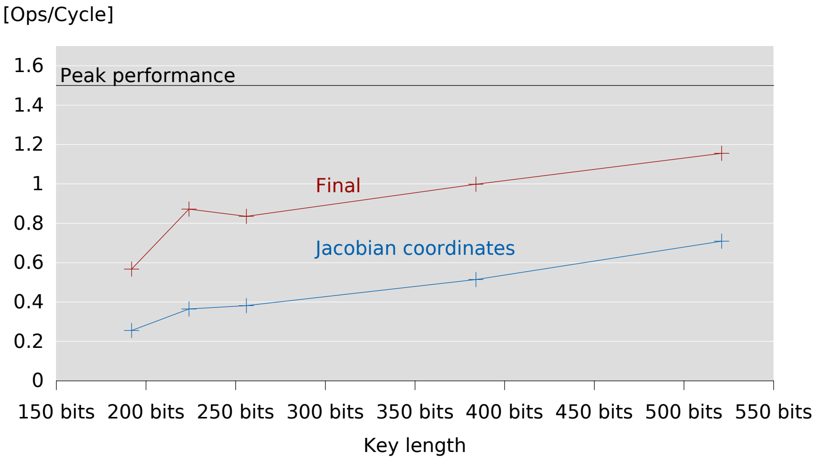
Final optimization compare with OpenSSL



Performance plot Part 1



Performance plot Part 2



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