Part I: Introduction
The team

- This software optimization project is joint work
  - Shay Gueron and Vlad Krasnov

- Shay Gueron
  - Assoc. Professor at the department of mathematics, University of Haifa, Israel
    - shay@math.haifa.ac.il
  - Intel Principal Engineer, Intel Corporation, Israel Development Center, Haifa, Israel

- Vlad Krasnov
  - Intern, Intel Corporation, Israel Development Center, Haifa, Israel
  - Student, Department of Computer Science, Technion I.I.T

Optimizing cryptographic primitives

- Why?
  - The need for end-to-end security in the internet, constantly increases the world-wide number (and percentage) of SSL/TLS connections.
  - Why aren’t all connections https://?
    - The costs
  - Cryptographic algorithms that support secure communications become a critical computational load for servers
    - Therefore an important target for optimization.
Optimizing cryptographic primitives

• What are we doing to see today?
  – Discuss techniques for speeding up the software performance of several cryptographic primitives
  – Target the ubiquitous x86_64 architectures
    • Used in most server platforms
  – Optimizations for the 2nd Generation Intel® Core™ Processor Family (Codename “Sandy Bridge”)
    • [Link to Sandy Bridge information](http://software.intel.com/en-us/articles/sandy-bridge/)

Results to be shown

• AES in CTR mode at 0.83 C/B
• AES GCM at 2.59 C/B
  • Compared to 10.42 C/B in non AES-NI implementation
• SHA-1 at 5.18 C/B (in coming OpenSSL)
  • Compared to current 7.79 C/B
• SHA-256 at 13.64 C/B
  • Compared to 18.4 in OpenSSL
• SHA-512
• RSA1024 at 5908 sign/sec
  • Compared to 3646 sign/sec in OpenSSL 1.0.0e
• RSA2048 at 853 sign/sec
  • Compared to 519 sign/sec in OpenSSL 1.0.0e
• If time permits: some results on RSA authentication
Optimizing cryptographic primitives

- How?
  - Optimized algorithms
    - RSA
      - Save multiplications
      - Improve primitives
        » Use AMM
        » Improved square
    - SHA2
      - Utilize SIMD
  - Optimized code
    - Assembler

- Use new instructions
  - AES-NI
  - AVX
  - (nondestructive destination)

- Leverage micro architectural features
  - shld (throughput 1)
  - Decoded Instruction Cache
  - Faster adc of immediate 0
    » ADC reg, 0
  - Faster MUL
Software Optimizations for Cryptographic Primitives on General Purpose x86_64 platforms

Shay Gueron
University of Haifa
Department of Mathematics, Faculty of Natural Sciences, University of Haifa, Israel

Intel Corporation
Intel Corporation, Israel Development Center, Haifa, Israel

shay@math.haifa.ac.il

IndoCrypt 2011 - Tutorial
Software Optimizations for Cryptographic Primitives on General Purpose x86_64 platforms

Part II:
Measurements methodology

Shay Gueron
University of Haifa
Department of Mathematics, Faculty of Natural Sciences, University of Haifa, Israel

Intel Corporation
Intel Corporation, Israel Development Center, Haifa, Israel

shay@math.haifa.ac.il

IndoCrypt 2011, December 11-14, 2011 Chennai, India
Significance

• How to compare performance across systems?
  – CPU cycles?
  – System time?
  – Process time?
• What to report?
  – Best result?
  – Average?
  – Worst case?
• Other factors
  – Turbo boost
  – Multithreading

Our methodology

For the numbers here
• We measure in CPU cycles
  – Compare between different architectures in IPC terms
  – Scales linearly with clock speed
• We measure average performance
• Turbo off
• Multithreading off
X86 – the RDTSC instruction

- Read Time Stamp Counter:
  - Reads the time stamp counter directly from cpu into edx:eax registers
  - Provides accurate results for profiling
- Usage:
  - start_clk = RDTSC
  - Repeat n times { func}
  - end_clk = RDTSC
  - average_clk = (end_clk – start_clk)/n
- We use n = 100,000
  - Even n = 10,000 is good enough...
- Precede with “warm-up” to train cache and branch predictors

Code snippet

```c
#define REPEAT 100000
#define WARMUP (REPEAT/4)
unsigned long long RDTSC_start_clk, RDTSC_end_clk;
#define RDTSC_MEASURE_ITERATOR;
inline unsigned long long get_Clks(void)
{
    unsigned long long ret_val;
    __asm__ volatile
    ("cpuid
     rdtsc
     mov %%eax,\(\%0\)
     mov %%edx,4(\%0)"
     ::"rm"(&ret_val):"eax","edx","ebx","ecx");
    return ret_val;
}
#define MEASURE(x)                                                                                         
    for(RDTSC_MEASURE_ITERATOR=0; RDTSC_MEASURE_ITERATOR<= WARMUP; RDTSC_MEASURE_ITERATOR++)       
    {                                                                                                      
        {x};                                                                                              
    }                                                                                                      
    RDTSC_start_clk = get_Clks();                                                                         
    for (RDTSC_MEASURE_ITERATOR = 0; RDTSC_MEASURE_ITERATOR < REPEAT; RDTSC_MEASURE_ITERATOR++)       
    {                                                                                                      
        {x};                                                                                              
    }                                                                                                      
    RDTSC_end_clk = get_Clks();                                                                          
    RDTSC_total_clk = (double)(RDTSC_end_clk-RDTSC_start_clk)/REPEAT;
```

Indocrypt 2011. Tutorial. Shay Gueron
Example – measure OpenSSL SHA1

```c
#include <openssl/bn.h>
#include "measurements.h"
#define LEN (1024)

int main()
{
    unsigned char message[LEN];
    unsigned char hash[20];
    int i;

    for(i=0; i<LEN; i++)
    {
        // Init message to 0,1,2,3...
        message[i] = i;
    }

    MEASURE({
        SHA1(message, LEN, hash);
    });

    for(i=0; i<20; i++)
    {
        printf("%02x", hash[i]);
    }
    printf("\n");
    printf("Total CPU cycles: %.2f.\n", RDTSC_total_clk);
    printf("CPU cycles/byte: %.2f.\n", RDTSC_total_clk/LEN);
}
```

Output:
```
>icc measure_sha1.c -DRDTSC -lcrypto -o measure_sha1
>measure_sha1
5b00669c480d5cffbdfa8bdeab956116f02d1b77
Total CPU cycles: 7998.49.
CPU cycles/byte: 7.81.
```

Indocrypt 2011. Tutorial. Shay Gueron
IndoCrypt 2011 - Tutorial

Software Optimizations for Cryptographic Primitives
on General Purpose x86_64 platforms

Shay Gueron
University of Haifa
Department of Mathematics, Faculty of Natural Sciences, University of Haifa, Israel

Intel Corporation
Intel Corporation, Israel Development Center, Haifa, Israel

shay@math.haifa.ac.il

IndoCrypt 2011, December 11-14, 2011 Chennai, India

IndoCrypt 2011 - Tutorial
Software Optimizations for Cryptographic Primitives
on General Purpose x86_64 platforms

Part III:
AES Performance on the 2nd Generation
Intel® Core™ Processor Family

Shay Gueron
University of Haifa
Department of Mathematics, Faculty of Natural Sciences, University of Haifa, Israel

Intel Corporation
Intel Corporation, Israel Development Center, Haifa, Israel

shay@math.haifa.ac.il

IndoCrypt 2011, December 11-14, 2011 Chennai, India
Introduction

• AES: Advanced Encryption Standard
  - The Federal Information Processing Standard for symmetric encryption, and it is defined by FIPS Publication #197 (2001). From the cryptographic perspective, AES is widely believed to be secure and efficient, and is therefore broadly accepted as the standard for both government and industry applications.

• Intel introduced a new set of instructions, beginning with the
  - Previous Generation 2010 Intel® Core™ processor family based on the 32nm Intel® microarchitecture codename Westmere.

• Six instructions:
  - AESENCLAST, AESDEC, AESDELAST facilitate high performance AES encryption and decryption,
  - AESIMC and AESKEYGENASSIST: support the AES key expansion.

• These instructions provide full hardware support for AES
  - high performance; enhanced security; software usage flexibility
Introduction – cont’d

• We discuss here optimizations for using these instructions for the 2nd Generation Intel® Core™ processors family.
• Details can be found in:

Where to look for details

• Details found in:
  • AES-NI:
  • AES-GCM using PCLMULQDQ:
Optimization for AES on the 2nd Generation Intel® Core™

- In the 2nd Generation Intel® Core™ processor family:
  - The four AES round instructions
    - (AESENC, AESECNLAST, AESDEC AESDECLAST)
    - Throughput of 1 cycle and latency 8 cycles.
- Previous generation 2010 Intel® Core™ processors
  - Throughput of 2 cycles and latency of 6 cycles.

- The 2nd Generation Intel® Core™ offers
  - A two fold increase in the throughput
  - Some slowdown in the latency.
Throughput vs. Latency

• Latency dominated performance:
  – Perform 10 rounds serially; next round starts only after previous one ended
  – Performance: \( \approx 10 \times \text{AES-NI latency} + 1 \)

• Throughput dominated performance
  – Perform one round on one block; one round on second block ...
  – Once the round completed on first block, issue the second round for all blocks
  – Performance can be “1 round per throughput cycles”

• Software can be written differently to optimize for an architecture

• Parallelization parameter: how many blocks to operate on in parallel?
  – On the 2nd generation Core: optimal parallelization parameter is 8 blocks.
  – On the Previous generation Core: optimal parallelization parameter is 4 blocks

Parallel vs. Serial modes of operation

• Modes that allow parallelization
  – ECB
  – CTR
  – CBC decryption
  – GCM

• Serial modes
  – CBC encryption
Example: Optimizing the CTR mode

• CTR Pseudo code:
  
  IV is a 64bit value  
  Nonce is a 32bit value  
  ONE is the value 1 as 32bit in a big-endian notation  
  CTRBLK := NONCE || IV || ONE  
  FOR i := 0 to n-1 DO  
    CT[i] := PT[i] XOR AES(CTRBLK)  
    CTRBLK := CTRBLK + 1  
  END  

• Per each block of PT, a counter block is calculated, encrypted and xored with the PT, to produce CT. The values of the counter blocks are known in advance, that’s why it is possible to encrypt several blocks together, in order to improve performance.

Example: Optimizing the CTR mode

• Parallelizing four blocks in CTR:
  
  CTRBLK1 := NONCE || IV || ONE  
  CTRBLK2 := NONCE || IV || TWO  
  CTRBLK3 := NONCE || IV || THREE  
  CTRBLK4 := NONCE || IV || FOUR  
  FOR i := 0 to (n/4–1) DO  
    CT[i*4] := AES(CTRBLK1)  
    CT[i*4+1] := AES(CTRBLK2)  
    CT[i*4+2] := AES(CTRBLK3)  
    CT[i*4+3] := AES(CTRBLK4)  
    CT[i*4+1] := PT[i*4+1] XOR CT[i*4+1]  
    CT[i*4+2] := PT[i*4+2] XOR CT[i*4+2]  
    CT[i*4+3] := PT[i*4+3] XOR CT[i*4+3]  
  CTRBLK1 := CTRBLK1 + 4  
  CTRBLK2 := CTRBLK2 + 4  
  CTRBLK3 := CTRBLK3 + 4  
  CTRBLK4 := CTRBLK4 + 4  
  END  

  –If blocks remain, encrypt them serially

• Parallelizing for eight blocks is similar
Example: Optimizing the ECB mode

- C code:

```c
void AES_ECB_encrypt(const unsigned char *in, unsigned char *out, unsigned long length, const unsigned char *key, int nr)
{
    __m128i tmp1,tmp2,tmp3,tmp4,tmp5,tmp6,tmp7,tmp8;
    __m128i *Key_Schedule = (__m128i*)key;
    int i,j;
    if (length%16)
        length = length/16 + 1;
    else length/=16;
    for(i=0; i < length/8; i++){
        tmp1 = _mm_loadu_si128 (&((__m128i*)in)[i*8+0]);
        tmp2 = _mm_loadu_si128 (&((__m128i*)in)[i*8+1]);
        tmp3 = _mm_loadu_si128 (&((__m128i*)in)[i*8+2]);
        tmp4 = _mm_loadu_si128 (&((__m128i*)in)[i*8+3]);
        tmp5 = _mm_loadu_si128 (&((__m128i*)in)[i*8+4]);
        tmp6 = _mm_loadu_si128 (&((__m128i*)in)[i*8+5]);
        tmp7 = _mm_loadu_si128 (&((__m128i*)in)[i*8+6]);
        tmp8 = _mm_loadu_si128 (&((__m128i*)in)[i*8+7]);
        tmp1 = _mm_xor_si128 (tmp1,Key_Schedule[0]);
        tmp2 = _mm_xor_si128 (tmp2,Key_Schedule[0]);
        tmp3 = _mm_xor_si128 (tmp3,Key_Schedule[0]);
        tmp4 = _mm_xor_si128 (tmp4,Key_Schedule[0]);
        tmp5 = _mm_xor_si128 (tmp5,Key_Schedule[0]);
        tmp6 = _mm_xor_si128 (tmp6,Key_Schedule[0]);
        tmp7 = _mm_xor_si128 (tmp7,Key_Schedule[0]);
        tmp8 = _mm_xor_si128 (tmp8,Key_Schedule[0]);
        for(j=1; j < nr; j++)
            tmp1 = _mm_aesenc_si128 (tmp1,Key_Schedule[j]);
            tmp2 = _mm_aesenc_si128 (tmp2,Key_Schedule[j]);
            tmp3 = _mm_aesenc_si128 (tmp3,Key_Schedule[j]);
            tmp4 = _mm_aesenc_si128 (tmp4,Key_Schedule[j]);
            tmp5 = _mm_aesenc_si128 (tmp5,Key_Schedule[j]);
            tmp6 = _mm_aesenc_si128 (tmp6,Key_Schedule[j]);
            tmp7 = _mm_aesenc_si128 (tmp7,Key_Schedule[j]);
            tmp8 = _mm_aesenc_si128 (tmp8,Key_Schedule[j]);
        tmp1 = _mm_aesenclast_si128 (tmp1,Key_Schedule[j]);
        tmp2 = _mm_aesenclast_si128 (tmp2,Key_Schedule[j]);
        tmp3 = _mm_aesenclast_si128 (tmp3,Key_Schedule[j]);
        tmp4 = _mm_aesenclast_si128 (tmp4,Key_Schedule[j]);
        tmp5 = _mm_aesenclast_si128 (tmp5,Key_Schedule[j]);
        tmp6 = _mm_aesenclast_si128 (tmp6,Key_Schedule[j]);
        tmp7 = _mm_aesenclast_si128 (tmp7,Key_Schedule[j]);
        tmp8 = _mm_aesenclast_si128 (tmp8,Key_Schedule[j]);
        _mm_storeu_si128 (&((__m128i*)out)[i*8+0],tmp1);
        _mm_storeu_si128 (&((__m128i*)out)[i*8+1],tmp2);
        _mm_storeu_si128 (&((__m128i*)out)[i*8+2],tmp3);
        _mm_storeu_si128 (&((__m128i*)out)[i*8+3],tmp4);
        _mm_storeu_si128 (&((__m128i*)out)[i*8+4],tmp5);
        _mm_storeu_si128 (&((__m128i*)out)[i*8+5],tmp6);
        _mm_storeu_si128 (&((__m128i*)out)[i*8+6],tmp7);
        _mm_storeu_si128 (&((__m128i*)out)[i*8+7],tmp8);
    }
    for(j=i*8;j<length;j++)
        tmp1 = _mm_loadu_si128 (&((__m128i*)in)[j]);
        tmp1 = _mm_xor_si128 (tmp1,Key_Schedule[0]);
        for(i=1; i < nr; i++)
            tmp1 = _mm_aesenc_si128 (tmp1,Key_Schedule[i]);
        tmp1 = _mm_aesenclast_si128 (tmp1,Key_Schedule[i]);
        _mm_storeu_si128 (&((__m128i*)out)[j],tmp1);
}
```

More about CBC

- Encryption: $C_i = E_k(C_{i-1})$, $C_0 = IV$
  - Each ciphertext block depends on previous block
  - Can't calculate $C_i$ before $C_{i-1}$ is finished
  - However still can encrypt separate messages in parallel
- Decryption: $P_i = D_k(C_i) @ G_{k-1}$, $P_0 = IV$
  - All ciphertext is known, therefore can decrypt in parallel
CBC performance

The performance of AES CBC Encryption on 1KB buffer, Intel® Core™ i7-2600K.

Code parallelization applied
Different parallelization parameter for each architecture

CBC – concurrent encryption of multiple buffers

The performance of AES CBC Encryption on 8 x 1KB buffer in CPU cycles per Byte, Intel® Core™ i7-2600K vs. 4 x 1KB on Intel® Core™ i7-880 Processor, Lower is better.
Some numbers (ECB)
The performance of AES ECB Encryption/Decryption on 1KB buffer in CPU cycles per Byte, Intel® Core™ i7-2600K vs. Intel® Core™ i7-880 Processor, Lower is better

Some numbers (CTR)
The performance of AES CTR Encryption/Decryption on 1KB buffer in CPU cycles per Byte, Intel® Core™ i7-2600K vs. Intel® Core™ i7-880 Processor, Lower is better
Some numbers (XTS)

The performance of AES XTS Encryption/Decryption on 1KB buffer in CPU cycles per Byte, Intel® Core™ i7-2600 vs. Intel® Core™ i7-880 Processor, Lower is better

<table>
<thead>
<tr>
<th>Key Size</th>
<th>Encryption/Decryption</th>
<th>Performance on Intel® Core™ i7-2600K</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128</td>
<td>1.22</td>
<td>1.59</td>
</tr>
<tr>
<td>AES-192</td>
<td>1.18</td>
<td>1.40</td>
</tr>
<tr>
<td>AES-256</td>
<td>1.18</td>
<td>1.53</td>
</tr>
</tbody>
</table>

The effect of the parallelization parameter

The performance of AES ECB Encryption/Decryption on 1KB buffer in CPU cycles per Byte, Intel® Core™ i7-2600K. Encryption of 8 blocks at a time vs. encryption of 4 blocks at a time.
AES-GCM

- Authenticated encryption: Galois Counter Mode
- Produces a message digest, “Galois Hash” from the encrypted data.
- Used for high performance message authentication.
- In each step: previous Galois Hash value is XOR-ed with the current ciphertext block.
- The result is multiplied in GF(2^128) with a hash key value.
- GCM uses GF(2^128) defined by the (“lowest”) irreducible polynomial
  \[ g = g(x) = x^{128} + x^{7} + x^{2} + x + 1. \]

- The multiplication in GF(2^128) involves carry-less multiplication of 128-bit operands, to generate a 255-bit result (256-bit result with a zero msbit), followed by and reduction modulo the irreducible polynomial g.
- A more optimized software implementations of the GCM mode use a lookup table based algorithm
- Newer implementations uses SSE instructions and “slicing”
  - E. Käsper, P. Schwabe
- We use AES-NI and PCLMULQDQ

AES-GCM — cont’d

- The multiplication in GF(2^128) involves carry-less multiplication of 128-bit operands, to generate a 255-bit result (256-bit result with a zero msbit), followed by and reduction modulo the irreducible polynomial g.
- A more optimized software implementations of the GCM mode use a lookup table based algorithm
- Newer implementations uses SSE instructions and “slicing”
  - E. Käsper, P. Schwabe
- We use AES-NI and PCLMULQDQ
Performing Carry-less Multiplication of 128-bit Operands Using PCLMULQDQ

Algorithm 1

- Step 1: multiply carry-less the following operands: A0 with B0, A1 with B1, A0 with B1, and A1 with B0. Let the results of the above four multiplications be:
  \[ A_0 \cdot B_0 = [C_0 : C_1], \quad A_1 \cdot B_1 = [D_0 : D_1], \quad A_0 \cdot B_1 = [E_0 : E_1], \quad A_1 \cdot B_0 = [F_0 : F_1] \]
- Step 2: construct the 256-bit output of the multiplication \([A_1 : A_0] \cdot [B_1 : B_0]\) as follows:
  \[ [A_1 : A_0] \cdot [B_1 : B_0] = [C_1 : C_0 \oplus C_1 \oplus D_0 \oplus E_1 : D_0 \oplus C_1 \oplus D_0 \oplus F_1 \cdot D_0] \]

Performing Carry-less Multiplication of 128-bit Operands Using PCLMULQDQ

Algorithm 2

- Carry-less Karatsuba
- Algorithm 2

- Step 1: multiply carry-less the following operands: A1 with B1, A0 with B0, and \(A_0 \oplus A_1\) with \(B_0 \oplus B_1\). Let the results of the above three multiplications be:
  \[ [C_1 : C_0], \quad [D_1 : D_0], \quad [E_1 : E_0] \]
- Step 2: construct the 256-bit output of the multiplication \([A_1 : A_0] \cdot [B_1 : B_0]\) as follows:
  \[ [A_1 : A_0] \cdot [B_1 : B_0] = [C_1 : C_0 \oplus C_1 \oplus D_0 \oplus E_1 : D_1 \oplus C_1 \oplus D_0 \oplus E_0 \cdot D_0] \]
Reduction modulo $x^{128}+x^7+x^2+x+1$

— Algorithm 4

Denote the input operand by $[X_3,X_2,X_1,X_0]$ where $X_3$, $X_2$, $X_1$, and $X_0$ are 64-bit long each.

**Step 1:** shift $X_3$ by 63, 62 and 57-bit positions to the right. Compute the following numbers:

- $A = X_3 >> 63$
- $B = X_3 >> 62$
- $C = X_3 >> 57$

**Step 2:** XOR $A$, $B$, and $C$ with $X_2$. Compute a number $D$ as follows:

- $D = X_2 ^ A ^ B ^ C$ 

**Step 3:** shift $[X3:D]$ by 1, 2 and 7 bit positions to the left. Compute the following numbers:

- $[E1:E0] = [X3:D] << 1$
- $[F1:F0] = [X3:D] << 2$
- $[G1:G0] = [X3:D] << 7$

**Step 4:** XOR $[E1:E0]$, $[F1:F0]$, and $[G1:G0]$ with each other and $[X3:D]$. Compute a number $[H1:H0]$ as follows:

- $[H1:H0] = [X3 ^ E1 ^ F1 ^ G1 : D ^ E0 ^ F0 ^ G0]$ 

Return: $[X1 ^ H1 : X ^ H0]$ 

---

Bit Reflection Peculiarity of GCM

- Special peculiarity should be taken into account when implementing the GCM mode:
  - The standard specifies that the bits inside their 128-bit double-quad-words are reflected.
  - This is not merely the difference between Little Endian and Big Endian notations.
  - One approach for way handle the bit reflection peculiarity is to bit-reflect the input to the gfmul function.
Avoiding Bit Reflecting

• An alternative that voids bit reflecting the inputs.
• Fundamental property of carry-less multiplication:
  \[ \text{reflected}(A) \cdot \text{reflected}(B) = \text{reflected}(A \cdot B) \gg 1 \]

• Using this identity, the PCLMULQDQ instruction can be used for performing multiplication in the finite field \( GF(2^{128}) \) seamlessly, regardless on the representation of the input and the output operands.
Indocrypt 2011. Tutorial. Shay Gueron
Aggregated Reduction

• A way to delay the reduction step and apply the reducing to the aggregated result only once every few multiplications.
  – Proposed by Krzysztof Jankowski, Pierre Laurent.
• The standard Ghash formula is the following:
  – $Y_i = [(X_i + Y_{i-1}) \cdot H] \mod P$

Aggregated Reduction

• Instead, one can apply the following recursion:
  – $Y_i = [(X_i + Y_{i-1}) \cdot H] \mod P$
  – $= [(X_i \cdot H) + (Y_{i-1} \cdot H)] \mod P$
  – $= [(X_i \cdot H) + (X_{i-2} + Y_{i-3}) \cdot H^2] \mod P$
  – $= [(X_i \cdot H) + (X_{i-1} \cdot H^2) + (X_{i-2} + Y_{i-3}) \cdot H^3] \mod P$
  – $= [(X_i \cdot H) + (X_{i-1} \cdot H^2) + (X_{i-2} \cdot H^3) + (X_{i-3} + Y_{i-4}) \cdot H^4] \mod P$
  – This can be further expanded to any depth.
• The only overhead in this computation is the need to pre-calculate the powers of H mod P.
• The gain: reduction required only once per few blocks
Optimizing for 2nd Generation Core

- We found the best optimization for the 2nd generation Core to use 4 parallel CTR encryption with quad-aggregated reduction:
  - Encrypt 4 counter-blocks
  - Reduce using the aggregated method
  - Interleave the encryption-reduction

Results

The performance of AES-128 GCM Encryption on 4KB buffer in CPU cycles per Byte, Intel® Core™ i7-2600K vs. Intel® Core™ i7-880 Processor, Lower is better

Results

- AES-GCM:
  - 16KB message: 2.59 C/B.
  - 4KB message: 2.67 C/B
    - Reminder: CTR performance: 0.83 C/B
    - The cost of the GHASH is ~1.84
      - That's 69% of the computations
IndoCrypt 2011 - Tutorial
Software Optimizations for Cryptographic Primitives on General Purpose x86_64 platforms

Shay Gueron
University of Haifa
Department of Mathematics, Faculty of Natural Sciences, University of Haifa, Israel
Intel Corporation
Intel Corporation, Israel Development Center, Haifa, Israel
shay@math.haifa.ac.il

IndoCrypt 2011, December 11-14, 2011 Chennai, India

Part IV: SHA algorithms
Speeding up SHA-1, SHA-256 and SHA-512 on the 2nd Generation Intel® Core™ Processors

Shay Gueron
University of Haifa
Department of Mathematics, Faculty of Natural Sciences, University of Haifa, Israel
Intel Corporation
Intel Corporation, Israel Development Center, Haifa, Israel
shay@math.haifa.ac.il

IndoCrypt 2011, December 11-14, 2011 Chennai, India
Introduction

- SHA1 and SHA2 are widely used set of NIST standard hash functions
- SHA 2 consists of SHA-224/SHA-256/SHA-384/SHA-512
- SHA1 is the fastest (produces 160-bit digests)
- The most popular variants of SHA2: SHA-256 (256-bit digest)
- Surprisingly, SHA-512 (512-bit digest) is faster on modern 64-bit processors
- By truncating the result of SHA-512 to 256bit, it is possible to benefit from the performance of SHA-512 and still save space
  - SHA-512 truncation was recently standardized by NIST
- More details in:
  - S. Gueron, "Speeding up SHA-1, SHA-256, SHA-512 on the 2nd Generation Intel® Core™ Processors" (ITNG 2012)

SHA1 and SHA2 general flow

- SHA1/SHA2 flows can be viewed as:
  - Init - initialize the hash value
  - Update – split the message to (equally sized) blocks, and compress (to digest size)
  - Finalize – add padding to the last block, and compress the block (or two blocks)
- SHA1/SHA256 operate on 512bit (64byte) blocks
- SHA512 operates on 1024bit (128byte) blocks
The SHA-1 algorithm

- The compression function (“cipher”)
  - Input:
    - 512 bit of message
    - h0, ..., h4 – five 32-bit values (from previous compression)
  - Message scheduling:
    - Break input message into 16 32-bit big-endian words w[i]: 0 ≤ i ≤ 15
    - For i = 16 to 79: w[i] = \( \text{rol} (w[i-3] \ast w[i-8] \ast w[i-14] \ast w[i-16]), 1) \)
  - Init:
    - a = h0; b = h1; c = h2; d = h3; e = h4
The SHA-1 algorithm

- **Update:**
  - For i = 0 to 79
    - If 0 ≤ i ≤ 19: \( f = (b \& c) | \neg b \& d \); \( k = 0x5a827999 \)
    - If 20≤ i ≤ 39: \( f = b \oplus c \oplus d \); \( k = 0x6ed9eba1 \)
    - If 40≤ i ≤ 59: \( f = (b \& c) | (b \& d) | (c \& d) \); \( k = 0x8f1bbcdc \)
    - If 60≤ i ≤ 79: \( f = b \oplus c \oplus d \); \( k = 0xc66d9e17 \)
  - \( t = \text{rol}(a, 5) + f + e + k + w[i] \)
  - \( e = d; d = c; c = \text{rol}(b, 30); b = a; a = t; \)

- **Add to previous hash value:**
  - \( h0 = h0 + a \)
  - \( h1 = h1 + b \)
  - \( h2 = h2 + c \)
  - \( h3 = h3 + d \)
  - \( h4 = h4 + e \)

The SHA-1 algorithm - implementation

- OpenSSL implementation: uses the ROL instruction to rotate left by 1/5/30.
- Uses 3-source LEA instruction for the addition: rol(a, 5) + f + e + k + w[i]
  - Adding 3 operands in a single instruction
- Snippet from OpenSSL:
  ```
  rol $5, %edi
  andl %r12d, %ebx
  movi %eax, 12(%rsp)
  addl %edi, %ebp
  xorl %esi, %ebx
  rol $30, %r12d
  addl %ebx, % ebp
  leal 3518500249(%rax, %rsi, 1), %edi
  movi %r12d, %ebx
  movi 16(%rsi), %eax
  movi %ebp, %esi
  xorl %edx, %ebx
  ```
The SHA-256 algorithm

• The compression function:
  — Input:
    • 512 bit of message
    • h0 ... h7 — eight 32-bit values (from previous compression)
  — Message scheduling:
    • Break input message into 16 32-bit big-endian words w[i]; 0 ≤ i ≤ 15
    • For i = 16 to 63:
      – \( s0 = \text{ror}(w[i-15], 7) \oplus \text{ror}(w[i-15], 18) \oplus (w[i-15] >> 3) \)
      – \( s1 = \text{ror}(w[i-2], 17) \oplus \text{ror}(w[i-2], 19) \oplus (w[i-2] >> 10) \)
      – \( w[i] = w[i-16] + s0 + w[i-7] + s1 \)
  — Init:
    • a= h0; b= h1; c= h2; d= h3; e= h4; f= h5; g= h6; h= h7

• Update:
  — For i = 0 to 63:
    – \( s0 = \text{ror}(a, 2) \oplus \text{ror}(a, 13) \oplus \text{ror}(a, 22) \)
    – maj = (a \& b) \oplus (a \& c) \oplus (b \& c)
    – t2 = s0 + maj
    – s1 = \text{ror}(e, 6) \oplus \text{ror}(e, 11) \oplus \text{ror}(e, 25)
    – ch = (e \& f) \oplus ((\sim e) \& g)
    – t1 = h + s1 + ch + k[i] + w[i]
    – h = g; g = f; f = e; e = d + t1; d = c; c = b; b = a; a = t1 + t2
  — Add to previous hash value:
    • h0 = h0 + a; h1 = h1 + b; h2 = h2 + c; h3 = h3 + d
    • h4 = h4 + e; h5 = h5 + f; h6 = h6 + g; h7 = h7 + h
  — SHA-512 is similar, but all values are 64bit; the number of rounds is 80; the shift/rotate “immediates” are different
The SHA-256 algorithm -
implementation

– Snippet from OpenSSL:

```assembly
movl %r8d,%r13d
movl %r8d,%r14d
movl %r9d,%r15d
rorl $6,%r13d
rorl $11,%r14d
xorl %r10d,%r15d
xorl %r14d,%r13d
rorl $14,%r14d
andl %r8d,%r15d
movl %eax,%r11d
addl (%rbp,%rdi,4),%r12d
xorl %r13d,%r11d
movl %eax,%r14d
rorl $2,%r11d
rorl $13,%r13d
rorl %eax,%r15d
orl %ecx,%r14d
xorl %r13d,%r11d
andl %ecx,%r15d
addl %r12d,%edx
andl %ebx,%r14d
addl %r12d,%r11d
orl %r15d,%r14d
leaq 1(%rdi),%rdi
addl %r14d,%r11d
movl 4(%rsi),%r12d
```
Example 1: consider the following recurrence relation:

\[ a_0 = 0, \quad a_1 = 0, \quad a_n = (a_{n-1} + a_{n-2} + k) \mod 2^m, \quad \text{for } n \geq 2 \quad (k \text{ is a constant}) \]

---

### Efficient instructions choices for the 2nd Generation Intel® Core™ processors – cont.

- **Second observation**
- The SHLD instruction: SHLD reg1, reg2, imm8
  - Concatenates the registers reg1 and reg and shifts them to the left by the constant amount specified by imm8.
  - If reg1=reg2 we get rotation
  - Applies equivalently to right or left rotations

- **In the 2nd Generation Intel® Core™ processors family:**
  - SHLD by a constant has throughput 1 (and all shifts affect no flags)
  - Therefore: using SHLD reg1, reg1, imm8 (or SHRD reg1, reg1, imm8) for rotating a double-word (32-bit) or a quad-word (64-bit) by a constant, is faster than using the ROL/ROR instruction for this purpose.
Example 1: The SHA-256 compression:

```
for i = 16 to 79
    s0 = Right-Rotate(w[i-15], 7) \&\& Right-Rotate(w[i-15], 18) \&\& (w[i-15] >> 3)
    s1 = Right-Rotate(w[i-2], 17) \&\& Right-Rotate(w[i-2], 19) \&\& (w[i-2] >> 10)
    w[i] = w[i-16] + s0 + w[i-7] + s1
end
```

Option 1 (using ROL)

```
mov rax, W
mov r8d, [rax]
mov r10d,[rax+4]
mov r9d, [rax+44]
mov r15d,[rax+56]
add r8d, r9d
mov r11d, r10d
mov r12d, r10d
mov r14d, r13d
mov r15d, r13d
rol r10d, 25
rol r13d, 15
shr r12d, 3
shr r15d, 10
xor r10d, r12d
xor r13d, r15d
shr r11d, r14d
shr r14d, r13d
shr r15d, r13d
add r8d, r10d
mov [rax+64], r8d
```

Option 2: (using SHLD)

```
mov rax, W
mov r8d, [rax]
mov r10d,[rax+4]
mov r9d, [rax+44]
mov r15d,[rax+56]
add r8d, r9d
mov r11d, r10d
mov r12d, r10d
mov r14d, r13d
mov r15d, r13d
shld r10d, r10d, 25
shld r13d, r13d, 15
shr r12d, 3
shr r15d, 10
xor r10d, r12d
xor r13d, r15d
shld r11d, r11d, 14
shld r14d, r14d, 13
shr r15d, r13d
add r8d, r10d
mov [rax+64], r8d
```

...Don't try this at home...

NOTE:
For the 2nd Generation Core

Indocrypt 2011. Tutorial. Shay Gueron

### Replacing 3src-LEA and ROL instructions

- When compiling C code: the latest Intel compiler will do the replacement for you
  - (if tuned for the 2nd Generation Core)
- For assembler code: automatic substitution
- Example:
  - asm code (AT&T style); use the following SED scripts
  - `sed 's/roll\(\[ \] \[0-9]*\), \(*[a-z0-9-]*/shld\$2,3,\3/'`
  - `sed 's/\(leal\)\(\[ \] \[-0-9]*\)\(\[%a-z0-9-]\),\(\[%a-z0-9-\]*/leal\$2\(4,5\),\6\n\$1addl\$2,3,\6/'`

Indocrypt 2011. Tutorial. Shay Gueron
Results on the 2Nd Generation Intel® Core™ Processor

<table>
<thead>
<tr>
<th>SHA-1 (Update function; performance in Cycles/Byte)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSL 1.0.0e</td>
<td>7.79</td>
</tr>
<tr>
<td>OpenSSL 1.0.0e optimized by replacing ROL with SHLD</td>
<td>6.72</td>
</tr>
<tr>
<td>OpenSSL 1.0.0e optimized by replacing ROL by SHLD, and avoiding three source LEA</td>
<td>6.55</td>
</tr>
<tr>
<td>Intel optimized code using SSE instructions*</td>
<td>6.58</td>
</tr>
<tr>
<td>Intel optimized code using SSE instructions* With ROL replaced by SHLD (after compilation)</td>
<td>5.75</td>
</tr>
</tbody>
</table>

In addition: Andy Polyakov of the OpenSSL development team included the optimization discussed here, together with optimization in *, together with the use of non-destructive AVX instructions and other optimization achieved performance of 5.18 C/B.

The performance of SHA-256 and SHA-512 update functions on the 2nd Generation Intel® Core™ Processor. The results show the effect of our proposed optimizations, compared to the implementation of OpenSSL 1.0.0e as the baseline. Lower is better.
Part V: SHA-256/SHA-512
Parallelizing message schedules to accelerate hash computations

Shay Gueron
University of Haifa
Department of Mathematics, Faculty of Natural Sciences, University of Haifa, Israel
Intel Corporation
Intel Corporation, Israel Development Center, Haifa, Israel
shay@math.haifa.ac.il
IndoCrypt 2011, December 11-14, 2011 Chennai, India
Introduction

• Observation:
  • SHA-256 and SHA-512 hash functions
  • (Davies-Meyer) compress with a block cipher, using the message block as the key
    • Key expanded (“message scheduling”)
  • Message scheduling is independent of the state

The SHA-2 message scheduling

• The message schedule operates on 32 bit dwords (64-bit in SHA-512)
• Each input block is independent of the previous blocks and the state
• Measurements: SHA-256 message scheduling is ~27% of the total computational cost
• For SHA-512 almost 30% (there are 80 rounds)
SHA-256 message scheduling

- The compression function:
  - Input:
    - 512 bit of message
    - $h_0$ ... $h_7$ – 32 bit values with the hash value from previous compress
  - Message scheduling:
    - Break input message into 16 32-bit big-endian words $w[i]; 0 \leq i \leq 15$
    - For $i = 16$ to $63$:
      - $s_0 = \text{ror}(w[i-15], 7) \oplus \text{ror}(w[i-15], 18) \oplus (w[i-15] >> 3)$
      - $s_1 = \text{ror}(w[i-2], 17) \oplus \text{ror}(w[i-2], 19) \oplus (w[i-2] >> 10)$
      - $w[i] = w[i-16] + s_0 + w[i-7] + s_1$
  - Init:
    - $a = h_0; b = h_1; c = h_2; d = h_3; e = h_4; f = h_5; g = h_6; h = h_7$

SHA-256 – current implementation

```c
for(i=0; i<64; i++)
{
    if(i<16)
    {
        w[i] = bswap_32(msg[i]);
    }
    else
    {
        S0 = rotr(w[i-15],7)\&\&rotr(w[i-15],18)\&\&(w[i-15]>>3);
        S1 = rotr(w[i-2],17)\&\&rotr(w[i-2],19)\&\&(w[i-2]>>10);
        w[i] = w[i-16] + S0 + w[i-7] + S1;
    }
}
```
Quadrupled message scheduling

- QMS: computed 4 message schedules in parallel
  - Use SSE4/AVX instructions.

- Details in:
  - S. Gueron, V. Krasnov. Speeding up software implementation of SHA2 by processing multiple message schedules of a single message (to be published)
// this function gathers blocks from 4 consecutive message blocks
inline __m128i gather(unsigned int *address)
{
    __m128i temp;
    temp = _mm_cvtsi32_si128(address[0]);
    temp = _mm_insert_epi32(temp, address[16], 1);
    temp = _mm_insert_epi32(temp, address[32], 2);
    temp = _mm_insert_epi32(temp, address[48], 3);
    return temp;
}

// this function calculates the small sigma 0 transformation
inline __m128i sigma_0(__m128i W)
{
    return _mm_xor_si128(_mm_xor_si128(_mm_xor_si128(_mm_srli_epi32(W, 7), _mm_srli_epi32(W, 18)), _mm_xor_si128(_mm_srli_epi32(W, 3), _mm_slli_epi32(W, 25))), _mm_slli_epi32(W, 14));
}

// the message scheduling round
#define SCHEDULE_ROUND(w1, w2, w3, w4)
    s0 = sigma_0(w1);
    s1 = sigma_1(w2);
    schedule[i] = _mm_add_epi32(w3, Ki[i]);
    w3 = _mm_add_epi32(_mm_add_epi32(w3, w4), _mm_add_epi32(s0, s1));
    i++;

// this function calculates the small sigma 1 transformation
inline __m128i sigma_1(__m128i W)
{
    return _mm_xor_si128(_mm_xor_si128(_mm_xor_si128(_mm_srli_epi32(W, 17), _mm_srli_epi32(W, 10)), _mm_xor_si128(_mm_srli_epi32(W, 19), _mm_slli_epi32(W, 15))), _mm_slli_epi32(W, 13));
}
## Results

Performance of SHA-256 for four calls to the “Update” function, compressing 256 bytes

<table>
<thead>
<tr>
<th>Processor</th>
<th>OpenSSL 1.0.0.e</th>
<th>QMS method with shld optimization</th>
<th>QMS method with shld optimization and AVX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Previous Generation Intel® Core™</td>
<td>18.01 C/B</td>
<td>15.18 C/B</td>
<td>N/A</td>
</tr>
<tr>
<td>2nd Generation Intel® Core™</td>
<td>18.32 C/B</td>
<td>16.27 C/B</td>
<td>15.11 C/B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>13.64 C/B</strong></td>
</tr>
</tbody>
</table>

- 7.3% speedup on the 2nd Generation processor
- 14.3% speedup on the Previous generation processor

Note for SHA3 candidates

### Results – cont’d

Performance of SHA-256 on Atom, Core™ 2 Duo, “Westmere” and “Sandy Bridge” processors, as a function of the hashed buffer size (performance reported in cycles per byte (C/B); lower count means faster). Comparison between the OpenSSL 1.0.0.e and the QMS method
Future Processors

- AVX2 (in 2013) instructions set will allow manipulations on 256 bit of data
  - Would make QMS possible for SHA-512
  - Would allow 8-MS (EMS) for SHA-256

- Advanced vector extensions programming reference (June 2011).
  [link](http://software.intel.com/file/36945)

AVX2 Code snippet – SHA512 QMS

```c
#define vpbroadcastq(vec, k) vec = _mm256_broadcastq_epi64(*(__m128i*)k)

// this function calculates the small sigma 0 transformation
inline __m256i sigma_0(__m256i W) {
  return _mm256_xor_si256(_mm256_xor_si256(_mm256_srli_epi64(W, 7), _mm256_srli_epi64(W, 8)),
                        _mm256_xor_si256(_mm256_srli_epi64(W, 1), _mm256_slli_epi64(W, 56)));
}

// this function calculates the small sigma 1 transformation
inline __m256i sigma_1(__m256i W) {
  return _mm256_xor_si256(_mm256_xor_si256(_mm256_srli_epi64(W, 6), _mm256_srli_epi64(W, 61)),
                        _mm256_xor_si256(_mm256_srli_epi64(W, 19), _mm256_slli_epi64(W, 3)));
}

// the message scheduling round
#define SCHEDULE_ROUND(w1, w2, w3, w4) \
  vpbroadcastq(Ki, &k[i]); \
  s0 = sigma_0(w1); \
  s1 = sigma_1(w2); \
  schedule[i] = _mm256_add_epi64(w3, Ki); \
  w3 = _mm256_add_epi64(_mm256_add_epi64(w3, w4), _mm256_add_epi64(s0, s1)); \
  i++;
```

Indocrypt 2011. Tutorial. Shay Gueron
### Instruction Count (SHA-256)

Average number of instructions per invocation of update function

<table>
<thead>
<tr>
<th></th>
<th>OpenSSL 1.0.0e</th>
<th>Using QMS with AVX</th>
<th>Using 8-M5 with AVX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>3046</td>
<td>2709</td>
<td>2367</td>
</tr>
</tbody>
</table>

In SHA-256: 28% reduction in the instructions count

### Future Processors

- Currently: SHA-512 at 9.71 C/B
  - Note SHA-512/256 standardized
- How about QMS for SHA-512 using AVX2?

Average number of instructions per invocation of update function

<table>
<thead>
<tr>
<th></th>
<th>OpenSSL 1.0.0e</th>
<th>Using QMS with AVX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>3888</td>
<td>3245</td>
</tr>
</tbody>
</table>

This means tougher life for the SHA3 candidates
IndoCrypt 2011 - Tutorial
Software Optimizations for Cryptographic Primitives on General Purpose x86_64 platforms

Shay Gueron
University of Haifa
Department of Mathematics, Faculty of Natural Sciences, University of Haifa, Israel
Intel Corporation
Intel Corporation, Israel Development Center, Haifa, Israel
shay@math.haifa.ac.il

IndoCrypt 2011, December 11-14, 2011 Chennai, India

Part VI: RSA acceleration
How Fast can RSA go on General Purpose Processors?

Shay Gueron
University of Haifa
Department of Mathematics, Faculty of Natural Sciences, University of Haifa, Israel
Intel Corporation
Intel Corporation, Israel Development Center, Haifa, Israel
shay@math.haifa.ac.il

IndoCrypt 2011, December 11-14, 2011 Chennai, India
RSA preliminaries

Motivation

• RSA computations have a significant effect on the workloads of SSL/TLS servers
  • Handshake + authenticated encryption
  • AES ~0.8-5 C/B, SHA* ~6/11/18 C/B
  • RSA dominates the handshake hundred-thousands cycles
  • Every cycle counts...
• Focus: 512-bit modular exponentiation
  • used for 1024-bit RSA (most popular)
• 1024-bit modular exponentiation
  • used for 2048-bit RSA (NIST recommended)
• Our problem: how to speed up RSA in SW?
  – How fast can it go?
RSA in general 1

- 2n-bit modulus $N = P \times Q$
  - $P, Q$ are n-bit primes.
  - Can assume on RSA keys: $2^{n-1} < P, Q < 2^n$. (in OpenSSL: by construction)
- 2n-bit private exponent (private key) $d$.
- RSA Decryption is the heavy operation: 2n-bit $C^d \mod N$
  - A server workload.
- Using CRT (algorithm to get 4x speedup)
  - Pre-compute: $d_1 = d \mod (P-1), d_2 = d \mod (Q-1), Qinv = Q^{-1} \mod P$.
  - Convert problem to two n-bit modular exponentiations
  - $M_1 = C^{d_1} \mod P$ and $M_2 = C^{d_2} \mod Q$.
  - Recombine $C^d \mod N = M_2+(Qinv \times (M_1-M_2) \mod P) \times Q$. (negligible)
- RSA performance $\sim 2 \times n$-bit mod-exp

RSA in general 2

- For example: RSA2048
- $2n=2048 \rightarrow$ we measure 1024-bit mod-exp
  - Input: a, x, m; output $a^x \mod m$
    - a, x, m are 1024-bit
  - RSA key is re-used $\rightarrow$ Pre-computations allowed
  - Some used for mod-exp with Montgomery Multiplication
    - OpenSSL has: $R = 2^{2048} \mod m$ (1024-bit)
    - $k0 = -m^{-1} \mod 2^{64}$ (64-bit)
    - (m is the 1024-bit modulus)
Modular exponentiation

- Sequence of modular multiplications (or equivalent)
- Mod-exp cost (window size w)
  \[ \approx (k+1) \times w \times \text{Cost(MSQR)} + (k+1+2w) \times \text{Cost(MM)} + \text{Cost(Store/Retrieve)} \]
- For \( w=5 \)
  - Roughly 5 MSQR and 1MMUL
  - 512 bit mod-exp: 515 MSQRs and 135 MMs
  - 1024 bit mod-exp: 1025 MSQRs and 237 MMs
- Cost of “store/retrieve” is non-negligible
  - Required for side channel protected implementation.
  - Not related to mul/add instructions (related to gather/scatter)
  - This topic is not discussed here.

The w-ary Exponentiation

- Input: \( m \) (odd modulus), \( a < m \), \( x \) such that
  \[ x = x_0 + x_1 \times 2^w + ... + x_k \times 2^{kw}, \]
  where \( 0 \leq x_0, x_1, ..., x_k \leq 2^{w-1} \)
  (i.e., \( x \) is written as \( k+1 \) "digits" in radix \( 2^w \))
- Parameters: \( s, w \)
  - \( w \) is the window size; \( s \) is a Montgomery parameter
  - Pre-computed: \( c_2 = 2^{2s} \mod m \)
- Output: \( a^x \mod m \)

1. \( a' = \text{MM}(a, c_2) \)
2. \( m[0] = \text{MM}(c_2, 1) \)
3. \( m[1+] = a' \)
4. For \( i = 2, ..., 2^w-1 \)
   4.1. \( m[i] = \text{MM}(m[i-1+], a') \)
   End For
5. Store \( m[0], ..., m[2^w-1] \)
   in a table (A)
6. Retrieve \( m[0] \) from table A
   in a table (A)
7. \( h = m[0] \)
8. For \( i = k, ..., 0 \) do
   8.1. For \( j = 1, ..., w \)
     8.1.1. \( h = \text{MSQR}(h) \)
   End For
   8.2. Retrieve \( m[x_i] \) from A
   8.3. \( h = \text{MM}(h, m[x_i]) \)
   End For
9. \( h = \text{MM}(h, 1) \)
Return \( h \)
Montgomery Multiplications

• m: odd integer (modulus)
• a, b: integers such that 0 ≤ a, b < m
• t: a positive integer
• MM (a, b) = a×b×2⁻ᵗ mod m.
  – 2ᵗ is the Montgomery parameter.
• Montgomery multiplication is a most efficient method for computing modular exponentiation
  – Does not require division

Montgomery Multiplications basics

• Demonstrate computation of: a*b mod m
• Definition: MM (a, b) = a×b×2⁻ᵗ mod m.
• Pre-compute: R = 2²ᵗ mod m
  1. Convert inputs to “Montgomery base”:
     1. a' = a×2²ᵗ mod m;
        Do this by: a' = MM (a, R)
     2. b' = b×2²ᵗ mod m;
        Do this by: b' = MM (b, R)
  2. Do the MM: T = MM (a', b') = a'×b'×2⁻ᵗ mod m = a×b×2⁻ᵗ mod m
  3. Convert result to residues base: MM (T, 1) = T × 1×2⁻ᵗ mod m = a×b mod m
• Involves an overhead for convert to/from
  – Amortized for a long sequence… such as Mod-exp (see backup foils)
  – MM is “stable” (output can be used as an input to another MM)
Word-by-Word Montgomery Multiplication (WW-MM)

Input: \( m < 2^n \) (odd modulus), \( 0 \leq a, b < m, n=s \times k \)
Output: \( a \times b \times 2^{-n} \mod m \)
Pre-computed: \( k0 = -m^{-1} \mod 2^s \)

1. \( T = a \times b \)
   
   For \( i = 1 \) to \( k \) do
   
   2. \( T1 = T \mod 2^s \)
   
   3. \( Y = T1 \times k0 \mod 2^s \)
   
   4. \( T2 = Y \times m \)
   
   5. \( T3 = (T + T2) \)
   
   6. \( T = T3 \div 2^s \)

   End For

7. If \( T \geq m \) then \( X = T - m \);
   else \( X = T \)

Return \( X \)

Almost Montgomery Multiplication

- A variant of MM [1] (see also [2], [3])
- \( m \) odd integer; \( 0 \leq a, b < B \) for some \( B (= 2^n) \)
- AMM is an integer \( U \) satisfying the conditions
  - \( U \mod m = a \times b \times 2^{-t} \mod m \) and \( U < B \).
  - Almost Montgomery Square (AMSQR) is AMM where \( a = b \).
- AMM compared to MM: Easier end reduction
- AMM is “stable” (output can be used as input to next AMM)
- We DO the reduction, but...

3. Before: C. Walter (Montgomery multiplication with no end reduction)
Word-by-Word Almost Montgomery Multiplication (WW-AMM)

Input: \( m < 2^n \) (odd modulus), \( 0 \leq a, b < 2^n \), \( n=s \times k \)

Output: \( ab \times 2^{-n} \mod m \)

Pre-computed: \( k0 = -m^{-1} \mod 2^s \)

1. \( T = ab \)
2. For \( i = 1 \) to \( k \) do
3. \( T1 = T \mod 2^s \)
4. \( Y = T1 \times k0 \mod 2^s \)
5. \( T2 = Y \times m \)
6. \( T = T3 / 2^s \)
7. If \( T \geq 2^n \) then \( X = T - m \);
8. else \( X = T \)
9. Return \( X \)

Post-condition: \( X \mod m = ab \times 2^{-n} \mod m \), and \( X < 2^n \)

---

w-ary exponentiation using Almost MM

- Input: \( m \) (odd modulus), \( a < B \), \( x \) such that
  \( x = x0 + x1 \times 2^w + \ldots + xk \times 2^{kw} \)
  \( 0 \leq x0, x1, \ldots, xk \leq 2^{w-1} \)
  (i.e., \( x \) is written as \( k+1 \) "digits" in radix \( 2^w \))
- Parameters: \( s, w \)
- Pre-computed: \( c2 = 2^{2s} \mod m \)
- Output: \( a^x \mod m \)

1. \( a' = \text{AMM}(a, c2) \)
2. \( m[0] = \text{AMM}(c2, 1) \)
3. \( m[1] = a' \)
4. For \( i = 2, \ldots, 2^{w-1} \) do
5. \( m[i] = \text{AMM}(m[i-1], a') \)
6. Store \( m[0], \ldots, m[2^{w-1}] \) in a table \( A \)
7. \( h = m[0] \)
8. For \( i = k, \ldots, 0 \) do
9. For \( j = 1, \ldots, w \) do
10. \( h = \text{AMSQR}(h) \)
11. Retrieve \( m[xi] \) from \( A \)
12. \( h = \text{AMM}(h, m[xi]) \)
13. End For
14. \( h = \text{REDUCE h modulo m} \)
15. Return \( h \)
RSA algorithms

RSA-Zariz (RSAZ)

- Offering Improvements
  - At the primitives’ level:
    - Improved MM/MSQR and AMM/AMSQR
    - Big-mul, big-sqr, interleaving and fusing
  - At the exponentiation level:
    - Improved optimized Store/Retrieve: reduce the cost of protecting the w-ary exponentiation algorithm against cache/timing side channel attacks (not in scope here)
    - Other shortcuts
  - Fully described in [1].
  - Implemented in: S. Gueron and V. Krasnov. (OpenSSL patch);
Two-Step Folding AMM (RSAX) 2009

  - Stated only for a 512-bit modulus (can be generalized)
  - Mistakenly claimed to be an algorithm for \(a \times b \times 2^{-128} \mod m\) (MM), but actually computes 512-bit AMM.
  - Assume \(2^{511} < m < 2^{512}\) (can be assumed for RSA in OpenSSL)
  - Store/Retrieve optimization
  - Speedup on OpenSSL ver. 0.9.8h (current in 2009; was slow implementation)
  - A different interface compared to WW-MM (more difficult to integrate)
  - Suggested as the fastest modular multiplication (but it is not the case)

(*) corrected, generalized, and proven in [1]

Indocrypt 2011. Tutorial. Shay Gueron

---

Two-Step Folding Almost Montgomery Multiplication

- Input: \(m < 2^n\) (odd modulus), \(0 \leq a, b < 2^n\); \(n = 4s\)
- Output: \(X\) satisfying the Post-condition
- Pre-compute:
  - \(k_1 = -m^{-1} \mod 2^s\)
  - \(M_1 = 2^{6s} \mod m; M_2 = 2^{5s} \mod m\)
- Tab\[0\] = \(0\)
- Tab\[1\] = \(2^{4s} \mod m\)
- Tab\[2\] = \(2^{4s} \mod m\)
- Tab\[3\] = \(2^{4s+1} \mod m\)
- Tab\[4\] = \(2^{5s} \mod m\)
- Tab\[5\] = \((2^{5s} + 2^{4s}) \mod m\)
- Tab\[6\] = \((2^{5s} + 2^{4s}) \mod m\)
- Tab\[7\] = \((2^{5s} + 2^{4s+1}) \mod m\)
- Flow:
  1. \(X = a \times b\)
  2. \(cf_1 = 0; cf_2 = 0; cf_3 = 0\)
  3. \(X_h = \lfloor X / 2^{6s}\rfloor; \ X_l = X \mod 2^{6s}\)
  4. \(X = X_h \times M_1 + X_l\)
  5. If \(X \geq 2^{6s}\)
    5.1. \(cf_1 = 1\)
    5.2. \(X = X \mod 2^{6s}\)
  6. \(X_h = \lfloor X / 2^{5s}\rfloor; \ X_l = X \mod 2^{5s}\)
  7. \(X = X_h \times M_1 + X_l\)
  8. If \(X \geq 2^{5s}\)
    8.1. \(cf_2 = 1\)
    8.2. \(X = X \mod 2^{5s}\)
  9. \(X_l = X \mod 2^s\)
  10. \(Q = (X_l \times k_1) \mod 2^s\)
  11. \(X = X + m \times Q\)
  12. If \(X \geq 2^{4s}\)
    12.1. \(cf_3 = 1\)
    12.2. \(X = X \mod 2^{4s}\)
  13. \(X = X / 2^s\)
  14. \(X = X + \text{Tab}[cf_1 \times 4 + cf_2 \times 2 + cf_3]\)
  15. If \(X \geq 2^{4s}\)
    15.1. \(X = X - m\)
  16. If \(X \geq 2^{4s}\)
    16.1. \(X = X - m\)
- Return \(X\)
- Post-condition:
  \(X \mod m = a \times b \times 2^{-s} \mod m\), and \(X < 2^n\)

Indocrypt 2011. Tutorial. Shay Gueron

---

1/2/2012
RSAZ - WW Reduction

960 bit \rightarrow \times \rightarrow 960 bit

\langle \times \rangle \times m \mod 2^64

\rightarrow m

Montgomery trick, low 64-bits are indeed 0

Repeat 8 times

If result $\geq 2^{512}$ subtract m (once)

TSF-AMM - First stage of reduction

768 bit \rightarrow \times \rightarrow 512 bit

2^{128} \mod m

\rightarrow m

512 bit \rightarrow \times \rightarrow 32 bit

2^{32} \mod m

\rightarrow m
TSF-AMM - Second stage of reduction

\[(\pm 1)^{m} \mod 2^{128}\]

128 bit \( \rightarrow \) 512 bit

TSF-AMM - Final correction

\[
\begin{array}{c}
000: 0 \\
001: 2^{512} \mod m \\
010: 2^{1024} \mod m \\
011: 2^{1024} \mod m \\
100: 2^{2048} \mod m \\
101: 2^{2048} \mod m \\
110: 2^{2048} + 2^{512} \mod m \\
111: 2^{2048} + 2^{512} \mod m
\end{array}
\]

512 bit \( \rightarrow \) 512 bit

If result still \( \geq 2^{512} \) subtract m
Comparing WW-AMM and TSF-AMM

- Different number of (pre-computed) constants
  - (RSAZ) WW-AMM:
    - One 512-bit value
  - (RSAX) TSF-AMM:
    - Eight 512-bit pre-computed values
    - Two 512-bit constants $M_1, M_2$
    - One 128-bit (k1)

- (RSAZ) WW-AMM:
  - Easily reverted to WW-MM (no need to require $2^{n-1} < m < 2^n$)
  - OpenSSL compatible: Integrates naturally into OpenSSL interface
  - Already absorbed into main tree of the coming OpenSSL version

- (RSAX) TSF-AMM
  - Requires tailored (nonstandard) interface for OpenSSL
  - Integrated an "engine" (thus outside FIPS boundary)

<table>
<thead>
<tr>
<th>Processor</th>
<th>OpenSSL 1.0.0e (WW-MM)</th>
<th>TSF-AMM (from RSAX patch)</th>
<th>WW-AMM (from RSAX patch)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU Cycles</td>
<td></td>
<td>CPU Cycles</td>
</tr>
<tr>
<td>Westmere</td>
<td>924</td>
<td>710</td>
<td>637</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>594</td>
<td>453</td>
<td>428</td>
</tr>
<tr>
<td></td>
<td>512-bit AMSQR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Westmere</td>
<td>N/A</td>
<td>588</td>
<td>550</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>N/A</td>
<td>401</td>
<td>342</td>
</tr>
</tbody>
</table>

Previous generation 2010 Intel® Core™ processor: WW-AMM is 33% faster than the OpenSSL implementation, and 12% faster than the TSF-AMM of [11].

2nd Generation Intel® Core™ processor: all three algorithms run significantly faster (by more than 30%); WW-AMM remains the fastest one.
The 2\textsuperscript{nd} Generation Intel® Core™ is faster

- Improved unsigned 64-bit multiplication (MUL) and add-with-carry (ADC) instructions
  - In the 2\textsuperscript{nd} Generation Intel® Core™ Processor
    - MUL: latency 4 cycles
      - Was 9 cycles
    - ADC (with immediate=0): latency 1 cycle
      - Was 2 cycles
  - “Decoded Instruction Cache”
    - Cache decoded instructions
    - Execute them faster when they are re-invoked.
    - Optimize algorithms by making its code stay resident in cache.

\textbf{RSAZ – savings at exponent level}

\textbf{OpenSSL pseudo flow}

1. \(a' = \text{MM}(a, c^2)\)
2. \(m[0] = \text{MM}(c^2, 1)\)
3. \(m[1] = a'\)
4. \(\text{For } i = 2, \ldots, 2^n:\)
   4.1. \(m[i] = \text{MM}(m[i-1], a')\)
5. Store \(m[0], \ldots, m[2^n-1]\) in a table \(A\)
6. Retrieve \(m[0]\) from table \(A\)
7. \(h = m[0]\)
8. \(\text{For } i = k, \ldots, 0 \text{ do}\)
   8.1. \(\text{For } j = 1, \ldots, w\)
      8.1.1. \(h = \text{MM}(h, h)\)
   End For
8.2. Retrieve \(m[i]\) from table \(A\)
8.3. \(h = \text{MM}(h, m[i])\)
End For
9. \(h = \text{MM}(h, 1)\)

\textbf{RSAZ pseudo flow}

1. \(a' = \text{AMM}(a, c^2)\)
2. \(m[0] = 2^{125} - m\) // Save a multiplication
3. \(m[1] = a'\)
4. \(\text{For } i = 1, \ldots, 2^n-1:\)
   4.1. \(m[i] = \text{AMM}(m[i], a')\) // Use square
   4.2. \(m[i+2] = \text{AMM}(m[i+1], a')\)
End For
5. Store \(m[0], \ldots, m[2^n-1]\) in a table \(A\)
6. Retrieve \(m[0]\) from table \(A\) // Optimized table
7. \(h = m[0]\)
8. \(\text{For } i = k-1, \ldots, 0 \text{ do}\) // Save an iteration
   8.1. \(\text{For } j = 1, \ldots, w\)
      8.1.1. \(h = \text{AMSQR}(h)\) // Use square
   End For
8.2. Retrieve \(m[i]\) from \(A\)
8.3. \(h = \text{AMM}(h, m[i])\)
End For
9. \(h = \text{AMM}(h, 1)\)
Optimizing the w-ary exponentiation’s Store/Retrieve

- Cache based side channel attacks are a recent threat to software implementations of cryptographic algorithms.
  - Due to such vulnerabilities, modular exponentiation code need to be written in a way that its memory access patterns (at the granularity of a cache line) do not leak secret information. This requires a special method for storing (in memory) and retrieving values from table A.

For $n=512$ and $w=5$, table holds $2^w=32$ values, each of 512-bit.

OpenSSL tackles the problem by storing the bytes of each such value at addresses spaced by $2^w$ bytes.

- Reading a 512-bit value from the scattered table involves 64 move operations to/from memory (all cache lines are accessed, thus dependency on the exponent bits is avoided).
- For platforms where the cache lines consist of 64 bytes (the more common case), this implementation supports window sizes of up to $w=6$.
- (If the cache lines consist of 32 bytes, the implementation supports window size of up to $w=5$).
Optimizing the $w$-ary exponentiation’s Store/Retrieve

- Reference [1] proposes a useful optimization
  - Tailored to platforms with cache lines of 64 bytes and the choice $w=5$.
  - The 32 values of the table are split into 16-bit “words”, which are stored at addresses spaced by $2^{w+1}$ words (i.e., $2\times2^w$ bytes).
  - This way, each 512-bit value of the table has one word in each of the cache lines spanned by the table.
  - Retrieving a value from the table involves only 32 move operations - half the number required by the OpenSSL implementation
  - (albeit with (acceptable) loss of generality).

---

Optimizing the $w$-ary exponentiation’s Store/Retrieve

- Our optimization:
  - Side channel store/retrieve protection has high cost
  - We choose a window size of $w=4$
  - Table has only $2^w=16$ 512-bit values.
  - This allows for scattering these 16 values in 32-bit “dwords” with spacing of $2^w$ dwords (i.e., $4\times2^w$ bytes).
  - The choice $w=4$ requires 144 AMM’s (9 more than with $w=5$).
  - On the other hand, retrieving a value from the table requires only 16 move operations, which is half the number of moves involved with the method of [1] and a quarter of the number of moves use by OpenSSL implementation.
  - In addition, the reduced table size with $w=4$ saves 1024 bytes (sixteen cache lines) in the first level cache, compared to $w=5$. 
Why is current OpenSSL slower?

• Until version 1.0.0e, OpenSSL is relatively slow. Why?
• Some insights:
  – Interleaved MM with no optimization for SQR
  – Slow side channel protection
  – Less efficient implementation of big-num multiplications
  – Exponentiation flow can be further optimized
  – Code Generality
  – Coding style
The coming OpenSSL 1.1.0

- A (near) future version of OpenSSL
  - Significantly faster than current OpenSSL 1.0.0e
    - Available as a “development branch” from OpenSSL site
    - Still unofficial ver. but will become official (soon, the new baseline)
- Improvements in RSA:
  - Borrows the ideas of RSAZ [1] (with due reference), from personal communications. Integrating a into OpenSSL main tree
    - Improved mod-exp implementation (any key size)
    - Dedicated modular square (MM) function
    - Assembly implementation with optimizations for sizes of 512/1024/2048 bit (although not with dedicated functions)
    - Dedicated fused multiply-gather function (for Store/Retrieve)

512-bit Modular Exponentiation

<table>
<thead>
<tr>
<th></th>
<th>512-bit modular exponentiation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenSSL 1.0.0e constant time</td>
</tr>
<tr>
<td>Previous Generation</td>
<td>CPU Cycles</td>
</tr>
<tr>
<td>Intel® Core™</td>
<td>675,000</td>
</tr>
<tr>
<td>2nd Generation</td>
<td>435,400</td>
</tr>
<tr>
<td>Intel® Core™</td>
<td></td>
</tr>
</tbody>
</table>

- RSAZ is ~47% faster than OpenSSL and ~10% faster than RSAX
**RSA1024**

<table>
<thead>
<tr>
<th></th>
<th>OpenSSL 1.0.0e constant time</th>
<th>RSAX (posted patch)</th>
<th>RSAZ (posted patch)</th>
<th>OpenSSL development branch (integrated RSAZ into main code)</th>
</tr>
</thead>
<tbody>
<tr>
<td>openssl speed sign/s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Previous Generation Intel® Core™</td>
<td>2,297</td>
<td>3,670</td>
<td>3,957</td>
<td>3,544</td>
</tr>
<tr>
<td>2nd Generation Intel® Core™</td>
<td>3,646</td>
<td>5,462</td>
<td>5,908</td>
<td>4,681</td>
</tr>
</tbody>
</table>

• RSAZ has X1.62-X1.72 the performance of OpenSSL and ~X1.08 that of RSAX

**1024-bit Modular Exponentiation**

<table>
<thead>
<tr>
<th>1024-bit modular exponentiation</th>
<th>OpenSSL 1.0.0e constant time</th>
<th>RSAZ (posted patch)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Cycles</td>
<td>4,413,906</td>
<td>2,624,316</td>
</tr>
<tr>
<td>Previous Generation Intel® Core™</td>
<td>2,833,200</td>
<td>1,823,342</td>
</tr>
</tbody>
</table>

• RSAZ is 35%-40% faster than OpenSSL (1.0.0e)
RSA2048

<table>
<thead>
<tr>
<th></th>
<th>OpenSSL 1.0.0e constant time</th>
<th>RSAZ</th>
<th>OpenSSL development branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>openssl speed sign/s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Previous Generation Intel® Core™</td>
<td>368</td>
<td>592</td>
<td>549</td>
</tr>
<tr>
<td>2nd Generation Intel® Core™</td>
<td>519</td>
<td>853</td>
<td>762</td>
</tr>
</tbody>
</table>

• RSAZ gives 1.61x-1.64x the performance of OpenSSL (1.0.0e)

Conclusion

• RSAZ is the fastest on WSM/SNB
  – Mod-mul, mod-exp, full RSA
• Scales easily for general key lengths
  – RSAZ patch available for RSA1024/RSA2048
• Has a standard interface
  – Fits smoothly into OpenSSL
• RSAZ integrated into OpenSSL coming version
  – Gaining most of the performance
  • With OpenSSL 1.0.1 ➔ will become the comparison baseline
RSAZ references

- S. Gueron and V. Krasnov (OpenSSL patch), "Efficient and side channel analysis resistant 512-bit and 1024-bit modular exponentiation for optimizing RSA1024 and RSA2048 on x86_64 platforms" [http://marc.info/?l=openssl-dev&m=131365563615525&w=2](http://marc.info/?l=openssl-dev&m=131365563615525&w=2)

Soon to come
- S. Gueron, "Efficient Software Implementations of Modular Exponentiation" (extended version).
- S. Gueron and V. Krasnov, "Speeding up Big-Number Squaring"