RSA Cryptography Optimization for Embedded Systems

Final Project – SENG440, University of Victoria

Waltvin Lee
waltvin@uvic.ca

10th August 2021

Presentation Outline

- Project Specifications
- RSA Public-key Cryptosystem
- Optimizations Performed & Specific Details
- Optimizations that Reduced Performance
- Cycle-count results
- Hardware-assisted solution analysis
- Conclusion
- Invitation for Questions

Project Specifications

Objectives

- Implement RSA Encryption and Decryption targeted at 95-bit keys
- Optimize Montgomery Modular Multiplication
- Optimize 96-bit arithmetic operations
- Analyse hardware-assisted solutions

Platform

- QEMU emulating Fedora VM with ARMv7 Processor
- gcc compiler in the VM

(Rivest–Shamir–Adleman) Public-key Cryptosystem

- Public key pair (E, PQ) published publicly
- Private key D kept secret
- Keys typically are most typically 1024 to 4096 bits

To encrypt

$C = T^E \ mod \ PQ$, where C = ciphertext, T = plaintext

To decrypt

$T = C^D \ mod \ PQ$, where C = ciphertext, T = plaintext.

(Rivest–Shamir–Adleman) Public-key Cryptosystem

- Techniques to perform Modular Exponential efficiently
  - Multiply and Square Algorithm
  - Montgomery Modular Multiplication (MMM)
- Modular Exponential performs Multiply and Square once
- Multiply and Square performs mostly MMM heavily in a loop
- Focus on optimizing MMM

Optimizations Performed

- v0 – MMM Operator Strength Reduction
  - Multiplication is not required in MMM
  - One of the operands is always 0 or 1
- v1 – Further MMM Optimization
  - Replacing right shifts by ***i***-bit to only by 1 or 32
  - Storing only relevant 32-bit chunks of ***uint96_t***
- v2 – Optimizing ***uint96_t*** addition and subtraction
  - Loop Unrolling
- v3 - Rewriting ***rshift_uint96*** by ***i***-bit function
  - To two functions handling only right shifts by 1 or 32
  - Remove if branches required to handle shifts ranging 0-96
- v4 - Adding Register Keywords

Original MMM

Optimizations performed

- Operator Strength Reduction

```
uint96_t mmm(uint96_t X, uint96_t Y, uint96_t M, uint32_t numBits)
{
    uint96_t T_low = {0};
    uint32_t T_high = 0;

    int m = numBits;

    for (int i = 0; i < m; i++)
    {
        uint32_t n = (T_low.value[2] & 1) ^ ((rshift_uint96(X, i).value[2] & 1) & (Y.value[2]
& 1));

        if (rshift_uint96(X, i).value[2] & 1){
            T_low = add_uint96(T_low, Y);

            if (cmp_uint96(T_low, Y)){
                T_high += 1;
            }
        }

        if (n){
            T_low = add_uint96(T_low, M);

            if (cmp_uint96(T_low, M)){
                T_high += 1;
            }
        }
        T_low = rshift_uint96(T_low, 1);
        T_low.value[0] += T_high << 31;
        T_high >>= 1;

    }

    if (T_high & 1 || !cmp_uint96(T_low, M)){
        T_low = sub_uint96(T_low, M);
    }

    return T_low;
}
```

Final MMM

Optimizations performed

- Operator Strength Reduction
- Storing relevant 32-bit sizes
- Replacing shifts by ***i***-bits to 1 or 32
- Register Keyword

```
uint96_t mmm(uint96_t X, uint96_t Y, uint96_t M, int numBits)
{
    register uint96_t T_low = {0};
    register uint32_t T_high = 0;
    register uint32_t n;
    register uint32_t X_32local;
    register uint32_t Y_0 = Y.low & 1;

    for (int i = 0; i < numBits; i++)
    {
        if(!(i % 32)){
            X_32local = X.low;
            X = rshift32_uint96(X);
        }
        n = (T_low.low & 1) ^ ((X_32local & 1) & Y_0);

        if (X_32local & 1){
            T_low = add_uint96(T_low, Y);

            if (cmp_uint96(T_low, Y)){
                T_high += 1;
            }
        }

        if (n){
            T_low = add_uint96(T_low, M);

            if (cmp_uint96(T_low, M)){
                T_high += 1;
            }
        }

        T_low = rshift1_uint96(T_low);
        T_low.high += T_high << 31;
        T_high >>= 1;

        X_32local >>= 1;
    }
    if (T_high & 1 || !cmp_uint96(T_low, M)){
        T_low = sub_uint96(T_low, M);
    }
    return T_low;
}
```

MMM Optimization – Replace shifts by ***i*** to by 1 or 32

- Compiler performs in-lining for ***rshift_uint96***
- Before: ***rshift_uint96*** by loop counter ***i*** ranging from 0 to 96
  - 271653 calls/branches to ***rshift_uint96*** in assembly
- After: ***rshift_uint96*** by constants 1 and 32
  - no calls/branches to ***rshift_uint96*** in assembly

MMM Optimization – Assembly changes

```
283         mov     r0, r7
284         bl      rshift_uint96
285         ldm     r7, {r0, r1, r2}
286         stm     r4, {r0, r1, r2}
287         mov     r9, r2
288         add     r6, r0, fp, lsl #31
289         lsr     fp, fp, #1
290         add     r10, r10, #1
291         ldr     r3, [sp, #20]
292         cmp     r3, r10
293         beq     .L45
294 .L34:
295         str     r10, [sp]
296         ldm     r8, {r1, r2, r3}
297         add     r0, sp, #56
298         bl      rshift_uint96
299         ldr     r5, [sp, #64]
300         ldr     r3, [sp, #16]
301         and     r5, r5, r3
302         eor     r5, r5, r9
303         and     r5, r5, #1
304         str     r10, [sp]
305         ldm     r8, {r1, r2, r3}
306         add     r0, sp, #68
307         bl      rshift_uint96
308         ldr     r3, [sp, #76]
309         tst     r3, #1
310         beq     .L28
311         str     r6, [r4]
```

```
282 .L40:
283         add     r7, r7, #1
284 .L32:
285         lsl     r0, r2, #31
286         add     r1, r0, r1, lsr #1
287         lsl     r0, r3, #31
288         add     r2, r0, r2, lsr #1
289         lsr     r3, r3, #1
290         add     r3, r3, r7, lsl #31
291         lsr     r7, r7, #1
292         lsr     r6, r6, #1
293         add     r8, r8, #1
294         cmp     r9, r8
295         beq     .L47
296 .L35:
297         tst     r8, #31
298         beq     .L48
299 .L28:
300         and     r4, r6, r10
301         eor     r4, r4, r1
302         and     r4, r4, #1
303         tst     r6, #1
304         beq     .L29
305         str     r3, [sp, #68]
306         str     r2, [sp, #72]
307         str     r1, [sp, #76]
308         str     fp, [sp, #128]
309         add     r3, sp, #120
310         ldm     r3, {r0, r1, r2}
```

uint96_t Optimization – Loop unrolling

Before loop unrolling to 96-bit addition and subtraction

- Internal array indexing in loop prevents register usage
- No In-lining to ***add_uint96*** and ***sub_uint96***
  - 89433 calls to ***add_uint96***
  - 191 calls to ***sub_uint96***

After loop unrolling to 96-bit addition and subtraction

- Register keyword can be used
- In-lining by compiler to ***add_uint96*** and ***sub_uint96***
  - 0 calls to both operations

Optimizations that Reduced Performance

- v2neon - Neon Addition
  - 96-bit addition body was replaced with Neon addition
  - Costly overhead of storing and loading into Neon data types/registers
- v5 - Loop-unrolling for MMM
  - Unrolling of five times to target 95-bit keys
  - Reduced performance by 33.75%
  - Tight dependencies = No parallelization
  - Larger code = More instruction cache misses

Cycle Count Averages (Rough Estimation)

Version	Optimizations Performed	Cycle Count Averages (Rough Est.)		
		-O0	-O1	-O3
v0	Operator Strength Reduction • Avoid 96-bit multiplication as one operand is always 0 or 1	43017.8	27207.2	6116.6
v1	+ MMM Optimization • Storing only relevant 32-bit chunks of uint96_t • rshift_uint96 in-lining by replacing shifts by ***i***-bit to 1 or 32	31139.2	12256.2	4931.2
v2	+ uint96_t Addition and Subtraction Optimization • Loop Unrolling • Avoided internal array indexing	29255.0	4022.2	5023.0
v2neon	+ Neon Addition (Not pursued / Removed for next version)	33277.4	13961.6	8141.6
v3	+ Rewriting rshift_uint96 to two functions handling only right shifts by 1 or 32	27711.8	3925.6	5251.8
v4	+ Register Keyword	21813.4	4254.2	4830.0

Hardware-assisted Solution Analysis

- A large portion of time is spent on ***rshift_uint96*** and ***add_uint96***
- ***rshift_uint96*** costs only approximately 5-8 instructions
- ***add_uint96*** costs around 18 instructions
  - Operands however are two 96-bit integers
  - Would require multiple hardware
- Larger chunks of operations that uses these 96-bit operations considered
  - Tight dependencies
  - Difficult put larger chunk of code into functions (especially of 32-bit operands)

Conclusion

- Important to understand software optimization techniques
  - Benefited strongest level of safe optimization
- Test runtime after each optimization
  - Beneficial in theory, but difficult to predict cache misses or how compiler interprets code
- -O1 flag performed better than the –O3 flag after manual optimizations
  - More optimization flags does not always mean faster program
  - Manual software optimizations can result in better performance

Any Questions?