Implementing Optimized Cryptography for Embedded Systems

Oct 21, 2019 • Sam Markelon

At the start of my summer internship with NCC Group's Cryptography Services (CS), I was told to find a research project that interested me. I have a background and interest in post-quantum cryptography, so when I came across FALCON, a lattice-based cryptographic signature algorithm, I got excited. My background is more on the theoretical side (as theoretical as an undergraduate can get), and I would not consider my self an especially experienced software engineer. The work being done on FALCON at NCC Group, specifically by Thomas Pornin, was all implementation based. One such implementation was the *cxm4* version of FALCON. This version, as the name implies, targeted systems with an ARM Cortex-M4 microprocessor, a popular choice for embedded system. I jumped on the opportunity to gain more experience with cryptographic implementation, but also learn something completely new in embedded systems.

At the start of the project, Thomas told me that the implementation of FALCOM-cxm4 needed to be kick-ass; he then defined this for me.

> …a technical term meaning 'safe, secure [constant-time] and fast' ~ **Thomas Pornin**

The following blogpost goes through some quick background, and then presents notes for complete newbies on developing for ARM. More accurately, I should say, notes on how *not* to develop for ARM, as the real lessons come from things I did wrong. Lastly, although, this was a cryptographic software project, most everything I say related to ARM development broadly applies to software projects of all sorts.

FALCON

As previously mentioned FALCON is a post-quantum signature algorithm. Thomas and a number of other cryptographers have been working on the project for a number of years now, and in January FALCON was announced to be one of the round 2 candidates for the NIST Post-Quantum Cryptography Project. The name is an acronym for "Fast-Fourier Lattice-based Compact Signatures over NTRU"…quite a mouthful, so we will just stick to FALCON. As the word post-quantum implies, FALCON aims to preserve its security even against quantum capable adversaries. Adversaries would be able to break many of the asymmetric encryption and signature schemes (e.g., RSA, DSA, and ECC) with ease when a general purpose quantum computer is realized. FALCON aims for strong security as it comes to leaking the secret key through simply observing any number of signatures, short keys and signatures, and fast signing and verification. Moreover, FALCON avoids key recovery attacks that plagued previous lattice-based signature schemes, such as GGH and NTRUSign. Above all, FALCON uses very little memory making it a good choice for embedded systems. The FALCON website has more information, including the specification document and a browsable source code tree.

ARM Cortex-M4

The Cortex-M4 is a popular choice of micro-controller for the embedded market, especially when digital signal processing is needed. The MCPU is based on the ARMv7E-M Harvard architecture and supports the Thumb and Thumb2 instruction sets. Notable for us was the constant time 32-bit hardware integer multiply and constant-time 32-bit hardware bit shifts. The Cortex-M4 also has optional floating point support, but only supports IEEE-754 single precision (binary32) format.

Motivation and Goals

As glanced over in the introductions the implementation had a number of necessary goals to accomplish. Security (constant time code) and speed were on the forefront. Since, this implementation targets embedded systems it is also pertinent that the code be highly efficient as to not bog down these low power machines.

Specifically of interest to me was that key generation and signing in FALCON involve the use of complex numbers, which can be approximated with IEEE-754 double precision (binary64) floating point numbers. This presents a problem when targeting the Cortex-M4. As mentioned above, the processor *optionally* supports single precision floating point numbers only. How do we solve this dilemma?

The solution is to emulate binary64 floating point numbers using a 64-bit unsigned integer type, the `uint64_t` type in C. Recalling introductory computer architecture class, the first bit is the sign bit, the next 11 are the exponent (-1022 to 1023 range), and then a 52 bit mantissa. Of course it's necessary that all of the floating point operations we need be crafted ourselves for this emulated type. Moreover, since the Cortex-M4 is a 32-bit processor this emulated type, denoted *fpr* from now on, be stored over two registers in the processor.

To heavily speed up performance of FALCON for the Cortex-M4, the cxm4 version was created. The cxm4 version features handcrafted ARM assembly version of fpr functions inlined into the code using GCC naked functions. These fpr functions and all others functions that dealt with sensitive data needed to be constant time to avoid side channel attacks. For a primer on constant time cryptography refer to Why Constant-Time Crypto?.

The specific focus of my work was the so called `fpr_expm_p63` function, which computes a polynomial approximation of `exp(-x)` where `x` is an fpr type. It is extensively used during signing.

Lessons Learned

I learned many things over the course of the project; unfortunately many of those things came the hard way. First off, it must be said that I really didn't know anything about the ARM architecture before starting this project, so I spent a good amount of time reading (sometimes skimming) the technical manual from ARM on the Cortex-M4, the ARMv7E-M architecture, and the Thumb2 instruction set. I don’t think any blogpost can substitute for that, so if you really want to learn this stuff open up your favorite search engine, acquire the appropriate manuals, and get reading. However, keep reading this post to avoid the mistakes I made, and additionaly for some entertainment.

Programming and Debugging Environment

DO NOT DO THE THING I AM ABOUT TO DESCRIBE (PROBABLY)

The following describes the debugging setup I used. I would write my handcrafted asm in `.s` files with my editor of choice (emacs) and then use the GCC ARM cross-compiler tool chain to assemble and link the file, qemu-arm to run the binary, and gdb to debug. qemu-arm provides user-space emulation, allowing you to run a binary with ease on Linux. I will describe why this is actually an issue and what you should do instead.

Install Packages

It is assumed you are running some version of Linux.

1. Install the necessary packages. The snippet assumes you are using Ubuntu, package names may be different on other distributions.

```
sudo apt install qemu qemu-user gdb-multiarch gcc-arm-none-eabi
```

1. To Test Code, Insert your function into the following code snippet.

```
        .text
        .global _start
_start:
        @ LOAD PARAMETERS HERE

        @ ADD MORE AS NECESSARY
        @ REMOVE THOSE THAT ARE UNNECESSARY
        MOVW    R0, #0
        MOVT  R0, #0
        MOVW    R1, #0
        MOVT    R1, #0

        @ START FUNCTION





        @ KEEP NOP SO LAYOUT REGS WORK
        NOP
        @ DEFINE OTHER FUNCTIONS HERE AS NECESSARY
```

1. Compile the source code with debugging symbols on.

```
arm-none-eabi-as template.s -g -o template.o
arm-none-eabi-ld template.o -o template
```

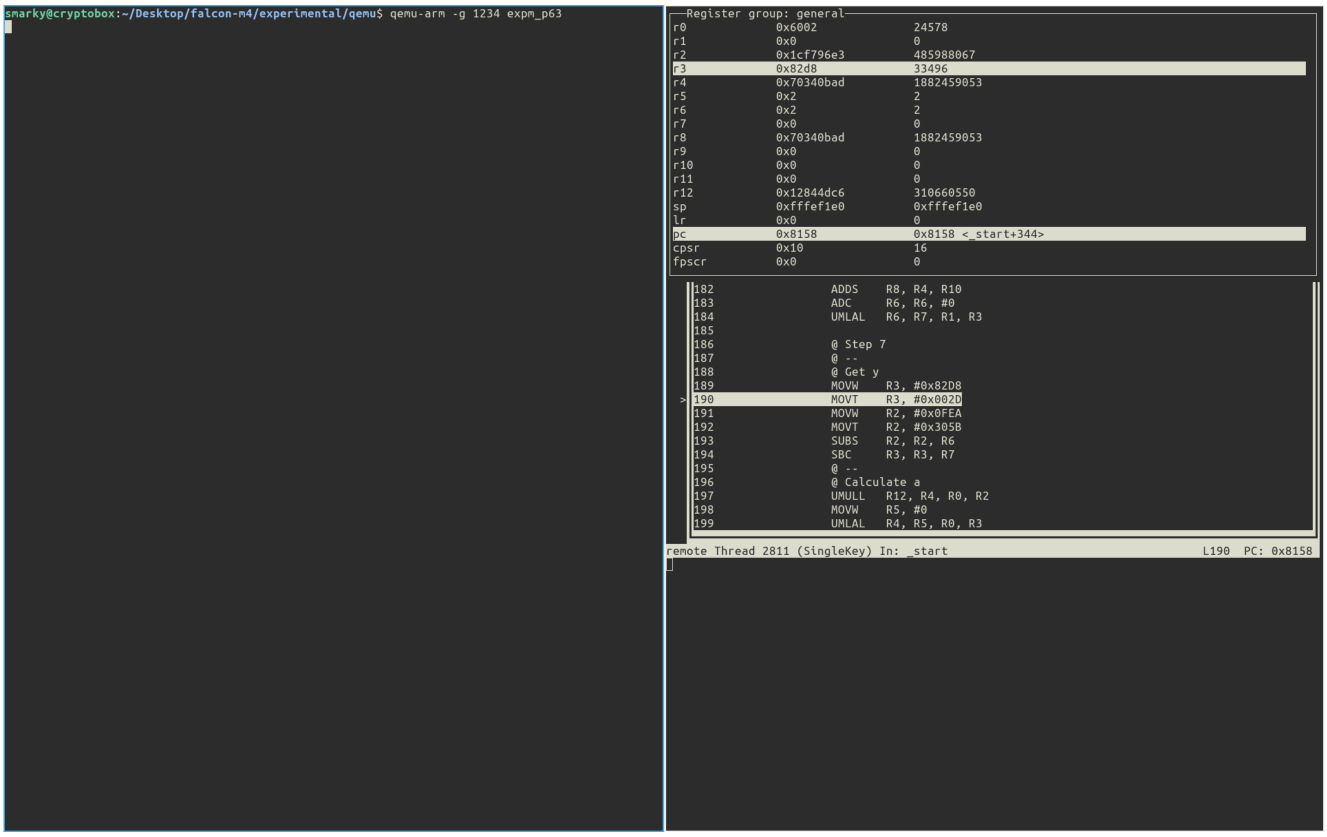
1. Now open two terminals, one for qemu and one for gdb. I do this side by side. Run qemu on our executable.

```
qemu-arm -g 1234 template
```

1. Attach gdb.

```
gdb-multiarch
...
(gdb) file template
...
(gdb) target remote localhost:1234
```

1. Access the gdb TUI (`ctrl+x a`). To view registers run command `layout regs` . Then step through the program.



```
smarky@cryptobox:~/Desktop/falcon-m4/experimental/qemu$ qemu-arm -g 1234 expm_p63
```

```
Register group: general
r0             0x6002              24578
r1             0x0                 0
r2             0x1cf796e3          485988067
r3             0x82d8              33496
r4             0x70340bad          1882459053
r5             0x2                 2
r6             0x2                 2
r7             0x0                 0
r8             0x70340bad          1882459053
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x12844dc6          310660550
sp             0xfffef1e0          0xfffef1e0
lr             0x0                 0
pc             0x8158              0x8158 <_start+344>
cpsr           0x10                16
fpscr          0x0                 0

                    ADDS    R8, R4, R10
                    ADC     R6, R6, #0
                    UMLAL   R6, R7, R1, R3

                    @ Step 7
                    @ --
                    @ Get y
                    MOVW    R3, #0x82D8
>                   MOVT    R3, #0x002D
                    MOVW    R2, #0x0FEA
                    MOVT    R2, #0x305B
                    SUBS    R2, R2, R6
                    SBC     R3, R3, R7
                    @ --
                    @ Calculate a
                    UMULL   R12, R4, R0, R2
                    MOVW    R5, #0
                    UMLAL   R4, R5, R0, R3
remote Thread 2811 (SingleKey) In: _start                L190  PC: 0x8158
```

The issue with this is that qemu-arm is not designed to simulate an ARM core; it is simply designed to run a Linux binary. Linux expects A-core ARM processors, so some things that work with this setup break on an actual M-Core ARM processor and some things that throw errors with qemu-arm may actually work on a Cortex-M4.

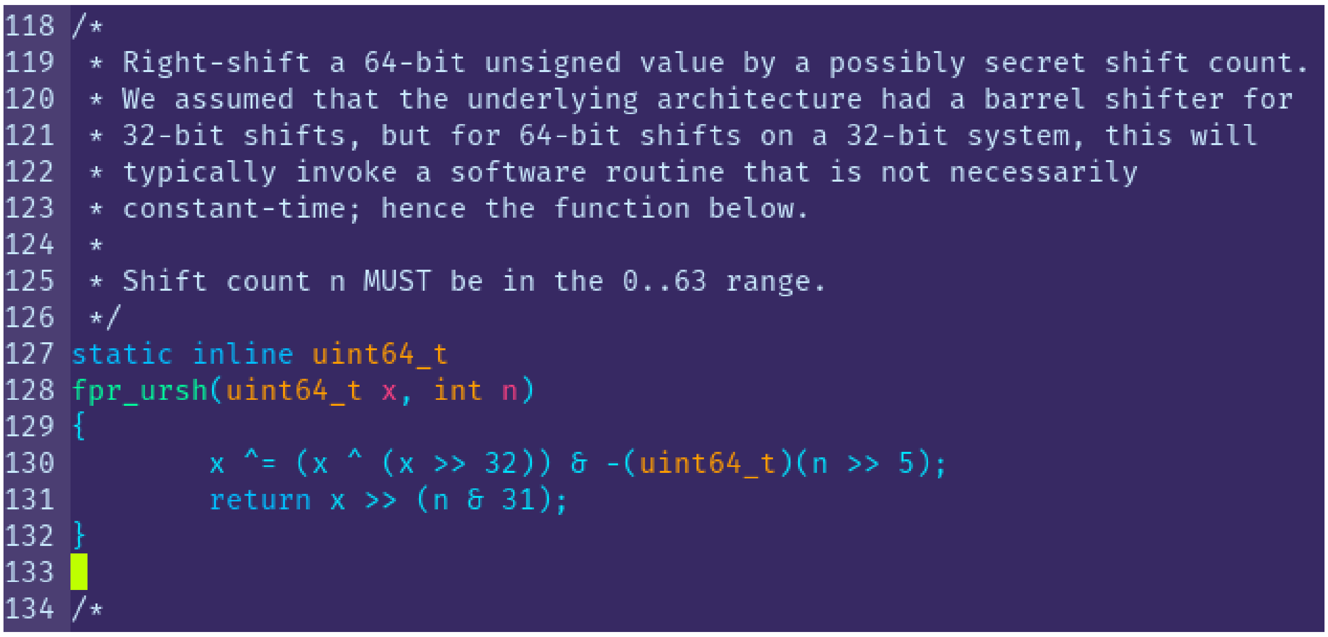
Another issue is that this setup is not very scalable. It does not work when trying to test the entire signature scheme. This setup may be fine for those just wanting to play around with snippets of ARM assembly, but for serious projects you are definitely going to need something else. The something else I suggest is the GNU MCU Eclipse project, which provides plugins for Eclipse and a series of other tools (specifically geared with the Cortex-M series in mind) to test projects of FALCON's size. I highly recommend checking out the linked website and reading the wonderful documentation to get a project rolling.

When it comes to testing on hardware the STM32F4 seems to be a good choice. Testing with hardware is beyond the scope of this post, however.

Compiler Knows Best (Sometimes)

There are unsigned 64-bit left and right shifts called in `fpr_expm63()`. Note these shifts occur on the underlying integer type, not the emulated type. Thomas implemented a generalized constant time version of these functions in C in the FALCON source. The functions seemed simple enough so I decided to start here for my first ARM assembly programming experience.

1. The C code of the generalized constant time version:



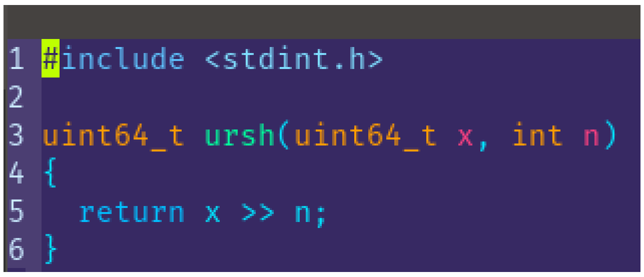
```
/*
 * Right-shift a 64-bit unsigned value by a possibly secret shift count.
 * We assumed that the underlying architecture had a barrel shifter for
 * 32-bit shifts, but for 64-bit shifts on a 32-bit system, this will
 * typically invoke a software routine that is not necessarily
 * constant-time; hence the function below.
 *
 * Shift count n MUST be in the 0..63 range.
 */
static inline uint64_t
fpr_ursh(uint64_t x, int n)
{
        x ^= (x ^ (x >> 32)) & -(uint64_t)(n >> 5);
        return x >> (n & 31);
}

/*
```

1. My handcrafted assembly for `fpr_ursh`

```
fpr_ursh:
        @ R0 holds the most significant 32 bits of x
        @ R1 holds the least significant 32 bits of x
        @ R2 holds n
        @       Assume n is in range 0 to 63

        MOVW R8, #0x0000
        MOVT R8, #0x0000

        @ R4|R5 is tmp x
        XOR R4, R0, R8
        XOR R5, R1, R0
        LRS R10, R2, #5
        SUB R10, R8, R10
        AND R4, R4, R10
        AND R5, R5, R8
        XOR R0, R0, R4
        XOR R1, R1, R5

        @ tmp = hi << 32 - n&31;
        @ hi >>= n&31;
        @ lo >>= n&31;
        @ lo |= tmp;
        AND R2, R2, #31
        MOVW R8, #32
        SUB R3, R8, R2
        LSL R6, R0, R3
        LSR R0, R0, R2
        LSR R1, R1, R2
        ORR R1, R1, R6
```

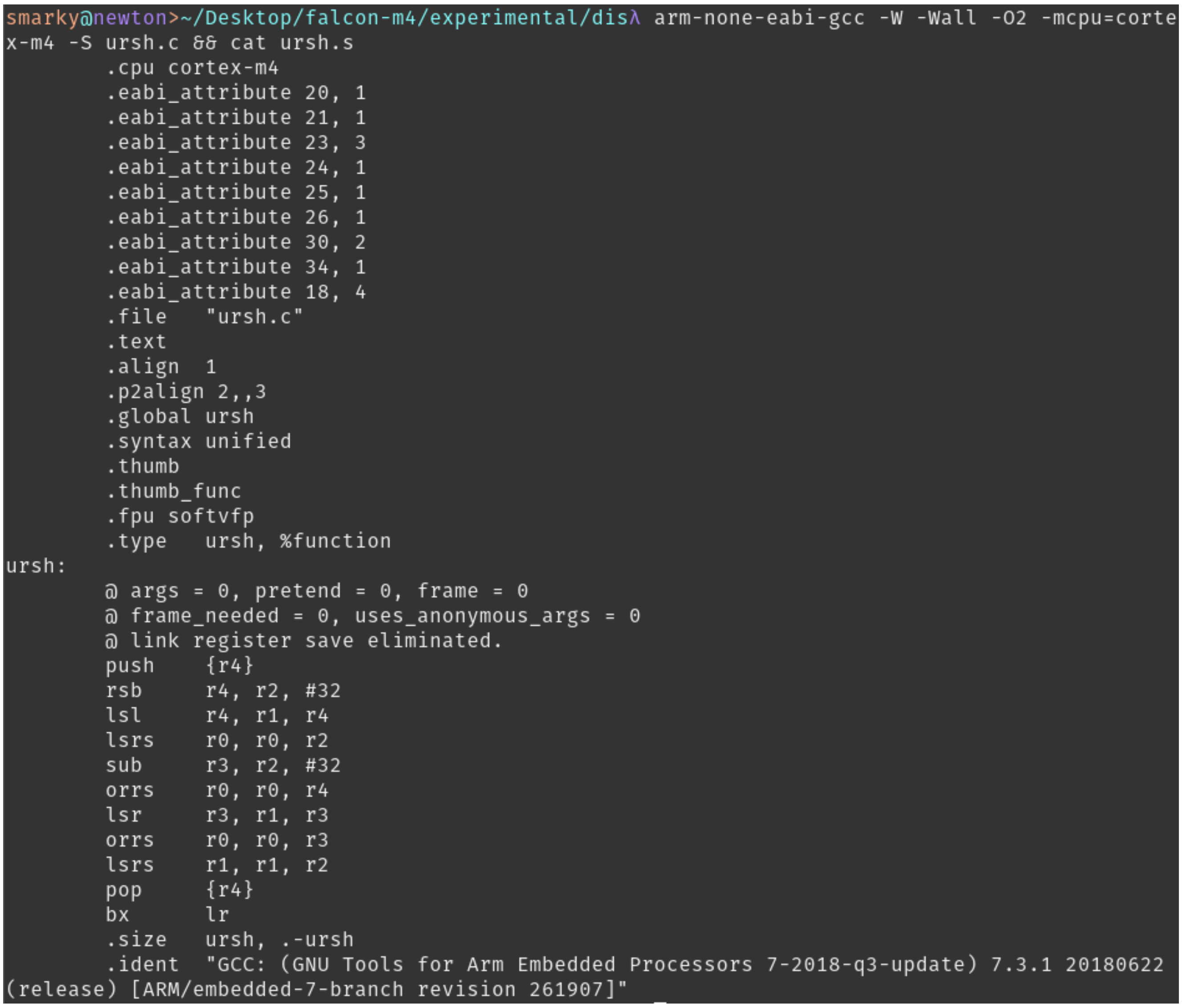
I was quickly told by Thomas that one could do better than what I did (in fact, that one could write more than half the number of opcodes that I wrote). He told me to compile and disassembles a simple function that takes a `uint64_t` as input, an `int` as a shift value, and returns a `uint64_t` shifted right by the shift value - one line of C with one bitwise operation. An astute observer may also note that the endianness in the above assembly I wrote is also wrong. This was also an issue that I realized unfortunately late and took some time to fix throughout my code.

1. The simple C code



```c
#include <stdint.h>

uint64_t ursh(uint64_t x, int n)
{
  return x >> n;
}
```

1. Disassembly output



```
smarky@newton>~/Desktop/falcon-m4/experimental/disλ arm-none-eabi-gcc -W -Wall -O2 -mcpu=corte
x-m4 -S ursh.c && cat ursh.s
        .cpu cortex-m4
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 1
        .eabi_attribute 30, 2
        .eabi_attribute 34, 1
        .eabi_attribute 18, 4
        .file   "ursh.c"
        .text
        .align  1
        .p2align 2,,3
        .global ursh
        .syntax unified
        .thumb
        .thumb_func
        .fpu softvfp
        .type   ursh, %function
ursh:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        @ link register save eliminated.
        push    {r4}
        rsb     r4, r2, #32
        lsl     r4, r1, r4
        lsrs    r0, r0, r2
        sub     r3, r2, #32
        orrs    r0, r0, r4
        lsr     r3, r1, r3
        orrs    r0, r0, r3
        lsrs    r1, r1, r2
        pop     {r4}
        bx      lr
        .size   ursh, .-ursh
        .ident  "GCC: (GNU Tools for Arm Embedded Processors 7-2018-q3-update) 7.3.1 20180622
(release) [ARM/embedded-7-branch revision 261907]"
```

As you can see, ignoring the function entry and exit there are eight simple constant time operations to right shift a 64-bit unsigned value created by the compiler. The lesson is to not assume things and use the compiler as a tool. I assumed that the output of the above simple C snippet would not be constant time, something I should have checked. Moreover, throughout the course of the project, I used the compiler as a tool to get ideas when it came to particularly *scary* looking bits of c code.

> **A Scary Bit of C Code**
>
> ```
> c = (a >> 32) + (b >> 32);
> c += (((uint64_t)(uint32_t)a
>  + (uint64_t)(uint32_t)b) >> 32);
> c += (uint64_t)z1 * (uint64_t)y1;
> ```

Closing remarks

There were a number of other little bugs along the way. There always is, it seems. One that was particularly frustrating took me two days to discover and fix - I was loading test parameters in the wrong radix. `3002 != 0x3002`. Ultimately, this very frustrating bug may have been a net good, as I improved my code and shaved off some more cycles trying to identify this bug.

So How Did I Do?

The table below contains the results of my efforts versus the assembly output from the compiler. I saved a few cycles handcrafting the smaller functions called within `fpr_expm_p63`, but the big speedup came from handcrafting the inner logic of the `fpr_expm_p63` function. It was to quantify the exact speed up because I saved a lot of overhead from not using branching, loading, and storing instructions. Moreover, the "inner loop" (the meat of the function) I wrote uses less cycles than the compiler's output. Note: I estimate a cycle equals an opcode; in the asm I wrote this is more or less true.

	Me		Compiler	
Function	# of cycles (ops)	Constant time (Y/N)	# of cycles (ops)	Constant time (Y/N)
`fpr_ulsh`	`-`	`-`	`8`	`Y`
`fpr_ursh`	`-`	`-`	`8`	`Y`
`fpr_ldexp`	`13`	`Y`	`16`	`Y`
`fpr_trunc`	`32`	`Y`	`34`	`Y`
`fpr_expm_p63`	`~180`	`Y`	`?`	`Y`

I encourage you to play with some ARM assembly, constant time cryptography, and perhaps if you are feeling snazzy, some constant time cryptography written in ARM assembly.

Sam Markelon, Cryptography Services intern, NCC Group

Cryptography Services

Cryptography Services is a dedicated team of consultants from NCC Group focused on cryptographic security assessments, protocol and design reviews, and tracking impactful developments in the space of academia and industry.