Assignment: Chacha20, Kyber, and Dilithium

Matthias J. Kannwischer and Bo-Yin Yang
Academia Sinica, Taipei, Taiwan
matthias@kannwischer.eu

08 June 2023, Summer School on real-world crypto and privacy, Vodice, Croatia

Assignments

- `https://github.com/mkannwischer/m4-tutorial-croatia2023`
- Part 1: Chacha20
- Part 2: Dilithium NTT (32-bit Montgomery multiplication)
- Part 3: Kyber NTT (16-bit Plantard multiplication)
- Solutions available on request in a few weeks
- You can easily spend weeks (or years in my case) on each of them
  $\Longrightarrow$ Up to you what to work on
  $\Longrightarrow$ Suggestion: Start with the Chacha20 quarterround; then Dilithium; then Kyber

Testing

- We target the Cortex-M4
- Usually: Use a cheap discovery board (e.g., STM32F407-DISCOVERY)
- For getting started: Emulate hardware using qemu
  - Very easy to setup; no fiddling with USB cables
  - Good enough for functional testing and debugging (can attach gdb)
  - Cannot perform benchmarks :(
- Once everything is working in qemu

$\Longrightarrow$ Let me know! We have 26 STM32F407 boards for benchmarking

$\Longrightarrow$ Need to be returned at the end of the tutorial!

Getting started

- Clone the repository

  ```
  git clone --recurse-submodules
  https://github.com/mkannwischer/m4-tutorial-croatia2023
  ```

- Follow setup instructions to setup qemu, arm-none-eabi-gcc, pyserial, and stlink
- Run hello world

  ```
  cd helloworld
  make
  make run-qemu
  ```

- Run one of the assignments (e.g., Chacha20)

  ```
  cd chacha20
  make
  make run-qemu
  ```

Getting started: Real hardware

- Two cables
  - Mini-USB for flashing software and power supply
  - UART adapter for receiving output from the board
- Connect USB cable to your laptop and board
- Connect white cable of the UART adapter to pin PA2

Getting started: Real hardware (2)

- Build `libopencm3` (lib supporting communication with a large number of M3/M4 microcontrollers)

  ```
  cd libopencm3
  make
  ```

- Build binary for target hardware

  ```
  cd helloworld
  make PLATFORM=stm32
  ```

  $\Longrightarrow$ Results in a binary (`bin/stm32f407-test.bin`)

- Write binary into the flash memory of the board

  ```
  st-flash write bin/stm32f407-test.bin 0x8000000
  ```

Getting started: Real hardware (3)

- Receive serial output from the board
  Linux: `pyserial-miniterm /dev/ttyUSB0 38400`
  MacOS: `pyserial-miniterm /dev/tty.usbserial-0001 38400`
  (If you have more USB devices connected, change to USB1/USB2 etc)
- Push the black reset button on the board ⇒ should see the same output as on qemu (now with real cycle counts!)

Getting started: Chacha20

- You are given the reference implementation of Chacha20
- Test is checking outputs of a single call to chacha20
- Task: Replace (parts of) reference implementation to make it fast
- Steps
  - Today: Write quarterround function in assembly
  - Later: Merge 4 quarterround functions into a full round
  - Later: Implement loop over 20 rounds in assembly
- Hints
  - Carefully study the slides on the barrel shifter
  - Note that if you just replace a single function, there is a lot of calling overhead and lots of loads and stores from/to memory. You won't see good performance until you complete all 3 steps.

Chacha20

- Stream-cipher
  - Input: key + nonce
  - Output: Random-looking byte string of certain size
- 256-bit key, 96-bit nonce, 32-bit counter
- State of 16 32-bit integers
- 20 rounds each consisting of 4 quarterrounds

Chacha20: quarterround

quarterround(a, b, c, d):

```
a += b; d ^= a; d <<<= 16;
c += d; b ^= c; b <<<= 12;
a += b; d ^= a; d <<<= 8;
c += d; b ^= c; b <<<= 7;
```

Chacha20: Double round

```
uint32_t x0, ..., x15;
```

Repeat 10 times

```
    quarterround(&x0, &x4, &x8, &x12);
    quarterround(&x1, &x5, &x9, &x13);
    quarterround(&x2, &x6, &x10,&x14);
    quarterround(&x3, &x7, &x11,&x15);
    quarterround(&x0, &x5, &x10,&x15);
    quarterround(&x1, &x6, &x11,&x12);
    quarterround(&x2, &x7, &x8, &x13);
    quarterround(&x3, &x4, &x9, &x14);
```

Chacha20: Single-block

- Initialization of the state
    - x0, ..., x3: constants 0x61707865, 0x3320646e, 0x79622d32, 0x6b206574 (ASCII: expand 32-byte k)
    - x4, ..., x11: key
    - x12: counter (0 for the first block, 1 for the second, etc.)
    - x13, .., x15: nonce
- Perform 20 rounds
- Add initial state to the state (addition modulo 2^{32})
- Produces 64 bytes of output

NTT Assignment

- Task: Write your your Dilithium and Kyber NTT (Folders: `dilithium`, `kyber`)
- Start with Dilithium as it is a little easier
- Follow the steps in the README to run the test
- Implement consecutively
    1. Modular multiplication (`test_mulmod` and `mulmod_asm`)
    2. Butterfly (`test_butterfly` and `butterfly_asm`)
    3. 1st layer (`test_nttlayer1` and `nttlayer1_asm`)
    4. NTT (`test_ntt` and `ntt_asm`)

Tests

- The assignment comes with extensive unit tests!
- When you start it should look at
    - `test.c`
    - `ref.c`
    - `m4.S`

Hints

- Dilithium prime: 8380417; Kyber prime: 3329
  - Dilithium: Represent coefficients as `int32_t`
  - Kyber: Represent coefficients as `int16_t`
- Dilithium: 8 layer NTT; Kyber: 7 layer NTT
- Dilithium: Use 32-bit signed Montgomery multiplication
  - Careful: Need to pre-compute twiddles in Montgomery domain $\implies tR \bmod q$ with $R = 2^{32}$
  - I have done that for you already (`twiddles_asm` in `test.c`); but try to understand it
- Kyber: Use Plantard multiplication
  - Need to pre-compute $tq^{-1} \bmod^{\pm} 2^{2\ell}$ with $\ell = 16$
  - Precomputation is already done; see `twiddles_asm` in `test.c`

More hints

- Note that the twiddle factors for the reference implementation are different from the ones on the assembly implementation
- In general, we would need to be very careful that the additions/subtractions do not overflowing
    - Need to add extra reductions before that happens
    - Here: We are lucky and there can be no overflows
        - Dilithium: Input at most $q \rightarrow$ Output at most $9q < 2^{31}$
        - Kyber: Input at most $q \rightarrow$ Output at most $4.5q < 2^{16}$

More hints (2)

- 32-bit Signed Montgomery multiplication ($R = 2^{32}$)
  ```
  smull tl, th, a, b
  mul t2, tl, q'
  smlal tl, th, t2, q
  ```
  (Result in `th`)
- 16-bit Plantard Multiplication by a constant
  ```
  smulwb r, bq', a
  smlabb r, r, q, q2^α
  ```
  (Result in upper half of `r`; $\alpha = 3$ for Kyber)

Next steps

- Benchmark your code on actual hardware
- Perform micro-architectural optimizations
  - Pipeline loads
  - Minimize memory access by performing multiple NTT layers at once (layer merging)
  - Use floating point registers to cache values
  - Ensure instruction alignment
- Add final reduction (e.g., Barrett reduction) after the NTT
- Implement inverse NTT
- Implement base multiplication in assembly
- Plug code into a Kyber/Dilithium implementation

Pointers

- STM32-getting-started: Simple example for getting started on the STM32F407 discovery board (and others)
  `https://github.com/mkannwischer/stm32-getting-started`
- PQM4: Collection of state-of-the-art implementations and unified benchmarking framework
  `https://github.com/mupq/pqm4`
- M4 Cryptographic Engineering Assignment: Unoptimized code + tests that you can try to speed-up (or give to students)
  Including SHA2, SHA-3/SHAKE, Poly1305, Gimli, ECDH25519, Chacha20, AES
  `https://github.com/mkannwischer/m4-crypto-eng-assignments`