Implementing post-quantum cryptography on embedded microcontrollers

Peter Schwabe
peter@cryptojedi.org
https://cryptojedi.org

September 17, 2019

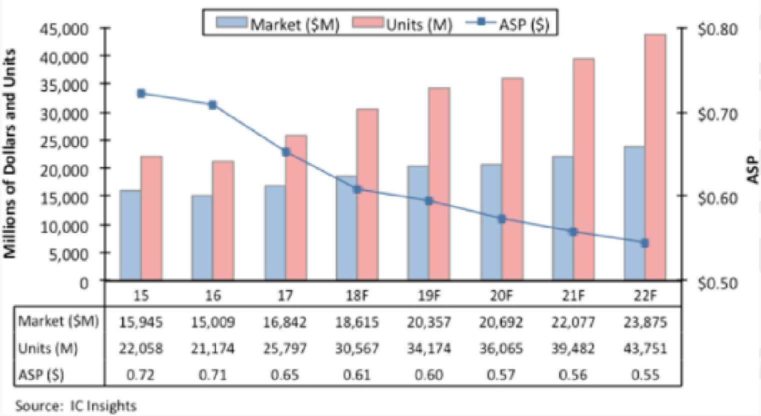
Embedded microcontrollers

"A microcontroller (or MCU for microcontroller unit) is a small computer on a single integrated circuit. In modern terminology, it is a system on a chip or SoC." —Wikipedia

Embedded microcontrollers

"A microcontroller (or MCU for microcontroller unit) is a small computer on a single integrated circuit. In modern terminology, it is a system on a chip or SoC."

—Wikipedia



	15	16	17	18F	19F	20F	21F	22F
Market ($M)	15,945	15,009	16,842	18,615	20,357	20,692	22,077	23,875
Units (M)	22,058	21,174	25,797	30,567	34,174	36,065	39,482	43,751
ASP ($)	0.72	0.71	0.65	0.61	0.60	0.57	0.56	0.55

- AVR ATmega and ATtiny 8-bit microcontrollers (e.g., Arduino)

...so many to choose from!

- AVR ATmega and ATtiny 8-bit microcontrollers (e.g., Arduino)
- MSP430 16-bit microcontrollers

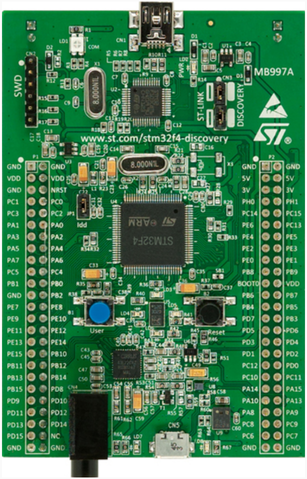
. . . so many to choose from!

- AVR ATmega and ATtiny 8-bit microcontrollers (e.g., Arduino)
- MSP430 16-bit microcontrollers
- ARM Cortex-M 32-bit MCUs (e.g., in NXP, ST, Infineon chips)
  - Low-end M0 and M0+
  - Mid-range Cortex-M3
  - High-end Cortex-M4 and M7

. . . so many to choose from!

- AVR ATmega and ATtiny 8-bit microcontrollers (e.g., Arduino)
- MSP430 16-bit microcontrollers
- ARM Cortex-M 32-bit MCUs (e.g., in NXP, ST, Infineon chips)
  - Low-end M0 and M0+
  - Mid-range Cortex-M3
  - High-end Cortex-M4 and M7
- RISC-V 32-bit MCUs (e.g., SiFive boards)

Our Target platform



- ARM Cortex-M4 on STM32F4-Discovery board
- 192KB RAM, 1MB Flash (ROM)
- Available for <25 EUR from various vendors (e.g., ebay, RS Components, Digi-Key): https://www.digikey.at/product-detail/de/stmicro/STM32F407G-DISC1/497-16287-ND/5824404
- Additionally need USB-TTL converter and mini-USB cable

Getting started: Hello world!

```
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

Getting started: Hello world!

```
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

- `gcc hello.c` is going to produce an x86 ELF file

Getting started: Hello world!

```
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

- `gcc hello.c` is going to produce an x86 ELF file
- Given an ARM ELF file, how do we get it to the board?

Getting started: Hello world!

```
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

- `gcc hello.c` is going to produce an x86 ELF file
- Given an ARM ELF file, how do we get it to the board?
- How would the ELF file get run?

Getting started: Hello world!

```
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

- `gcc hello.c` is going to produce an x86 ELF file
- Given an ARM ELF file, how do we get it to the board?
- How would the ELF file get run?
- What is `printf` supposed to do?

Getting started: Hello world!

```
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

- `gcc hello.c` is going to produce an x86 ELF file
- Given an ARM ELF file, how do we get it to the board?
- How would the ELF file get run?
- What is `printf` supposed to do?
- Should we even expect `printf` to work?

Fixing all of those issues: the idea

1. Install a cross compiler: `apt install gcc-arm-none-eabi`

Fixing all of those issues: the idea

1. Install a cross compiler: `apt install gcc-arm-none-eabi`
2. Install `stlink`:

```
apt install build-essential libusb-1.0-0-dev cmake
git clone https://github.com/texane/stlink.git
cd stlink && make release
cd build/Release && sudo make install
```

Fixing all of those issues: the idea

1. Install a cross compiler: `apt install gcc-arm-none-eabi`
2. Install `stlink`:

```
apt install build-essential libusb-1.0-0-dev cmake
git clone https://github.com/texane/stlink.git
cd stlink && make release
cd build/Release && sudo make install
```

3. Extend `hello.c` with some setup boilerplate code
   - Initialize CPU and set clock frequency
   - Set up serial port (USART) using USB-TTL

Fixing all of those issues: the idea

1. Install a cross compiler: `apt install gcc-arm-none-eabi`
2. Install `stlink`:

```
apt install build-essential libusb-1.0-0-dev cmake
git clone https://github.com/texane/stlink.git
cd stlink && make release
cd build/Release && sudo make install
```

3. Extend `hello.c` with some setup boilerplate code
   - Initialize CPU and set clock frequency
   - Set up serial port (USART) using USB-TTL
4. Replace `printf` with `send_USART_str`

Fixing all of those issues: the idea

1. Install a cross compiler: `apt install gcc-arm-none-eabi`
2. Install `stlink`:

```
apt install build-essential libusb-1.0-0-dev cmake
git clone https://github.com/texane/stlink.git
cd stlink && make release
cd build/Release && sudo make install
```

3. Extend `hello.c` with some setup boilerplate code
   - Initialize CPU and set clock frequency
   - Set up serial port (USART) using USB-TTL
4. Replace `printf` with `send_USART_str`
5. Compile to ARM **binary** (not ELF) file, say `usart.bin`

Fixing all of those issues: the idea

1. Install a cross compiler: `apt install gcc-arm-none-eabi`
2. Install `stlink`:

```
apt install build-essential libusb-1.0-0-dev cmake
git clone https://github.com/texane/stlink.git
cd stlink && make release
cd build/Release && sudo make install
```

3. Extend `hello.c` with some setup boilerplate code
   - Initialize CPU and set clock frequency
   - Set up serial port (USART) using USB-TTL
4. Replace `printf` with `send_USART_str`
5. Compile to ARM **binary** (not ELF) file, say `usart.bin`
6. Connect USB-TTL converter with board
7. Set up listener on serial port hostside

Fixing all of those issues: the idea

1. Install a cross compiler: `apt install gcc-arm-none-eabi`
2. Install `stlink`:

```
apt install build-essential libusb-1.0-0-dev cmake
git clone https://github.com/texane/stlink.git
cd stlink && make release
cd build/Release && sudo make install
```

3. Extend `hello.c` with some setup boilerplate code
   - Initialize CPU and set clock frequency
   - Set up serial port (USART) using USB-TTL
4. Replace `printf` with `send_USART_str`
5. Compile to ARM **binary** (not ELF) file, say `usart.bin`
6. Connect USB-TTL converter with board
7. Set up listener on serial port hostside
8. `st-flash write usart.bin 0x8000000` (flash over mini-USB)

Fixing all of those issues: the idea

1. Install a cross compiler: `apt install gcc-arm-none-eabi`
2. Install `stlink`:

```
apt install build-essential libusb-1.0-0-dev cmake
git clone https://github.com/texane/stlink.git
cd stlink && make release
cd build/Release && sudo make install
```

3. Extend `hello.c` with some setup boilerplate code
   - Initialize CPU and set clock frequency
   - Set up serial port (USART) using USB-TTL
4. Replace `printf` with `send_USART_str`
5. Compile to ARM **binary** (not ELF) file, say `usart.bin`
6. Connect USB-TTL converter with board
7. Set up listener on serial port hostside
8. `st-flash write usart.bin 0x8000000` (flash over mini-USB)
9. Push "Reset" button to re-run the program

Good news! Most of that work is already done.

https://github.com/joostrijneveld/STM32-getting-started

Good news! Most of that work is already done.

https://github.com/joostrijneveld/STM32-getting-started

- Includes examples for
  - Unidirectional communication ("Hello World!")
  - Bidirectional communication (echo)
  - Direct Memory Access
  - performance benchmarking
  - calling a function written in assembly

Good news! Most of that work is already done.

`https://github.com/joostrijneveld/STM32-getting-started`

- Includes examples for
  - Unidirectional communication ("Hello World!")
  - Bidirectional communication (echo)
  - Direct Memory Access
  - performance benchmarking
  - calling a function written in assembly
- Requires `python` and `python-serial` packages

Before we optimize: how do we benchmark?

```
SCS_DEMCR |= SCS_DEMCR_TRCENA;
DWT_CYCCNT = 0;
DWT_CTRL |= DWT_CTRL_CYCCNTENA;

int i;
unsigned int oldcount = DWT_CYCCNT;

  /* Your code goes here */

unsigned int newcount = DWT_CYCCNT;

unsigned int cycles = newcount - oldcount;
```

- See `cyclecount.c` example in STM32-Getting-Started

Before we optimize: how do we benchmark?

```
SCS_DEMCR |= SCS_DEMCR_TRCENA;
DWT_CYCCNT = 0;
DWT_CTRL |= DWT_CTRL_CYCCNTENA;

int i;
unsigned int oldcount = DWT_CYCCNT;

  /* Your code goes here */

unsigned int newcount = DWT_CYCCNT;

unsigned int cycles = newcount - oldcount;
```

- See `cyclecount.c` example in STM32-Getting-Started
- Caveats:
  - At >24 MHz wait cycles introduced by memory controller

Before we optimize: how do we benchmark?

```
SCS_DEMCR |= SCS_DEMCR_TRCENA;
DWT_CYCCNT = 0;
DWT_CTRL |= DWT_CTRL_CYCCNTENA;

int i;
unsigned int oldcount = DWT_CYCCNT;

  /* Your code goes here */

unsigned int newcount = DWT_CYCCNT;

unsigned int cycles = newcount - oldcount;
```

- See `cyclecount.c` example in STM32-Getting-Started
- Caveats:
  - At >24 MHz wait cycles introduced by memory controller
  - Cycle counter overflows after ≈3 min (20 MHz)

Optimizing

- Optimize software on the assembly level
  - Crypto is worth the effort for better performance
  - Also, no compiler to introduce, e.g. side-channel leaks
  - It's fun

Optimizing

- Optimize software on the assembly level
  - Crypto is worth the effort for better performance
  - Also, no compiler to introduce, e.g. side-channel leaks
  - It's fun
- Different from optimizing on "large" processors:
  - Size matters! (RAM and ROM)
  - Less parallelism (no vector units, not superscalar)
  - Often critical: reduce number of loads/stores

Cortex-M4 assembly basics

- 16 registers, `r0` to `r15`
- 32 bits wide
- Not all can be used freely
  - `r13` is `sp`, stack pointer (don't misuse!)
  - `r14` is `lr`, link register (can be used)
  - `r15` is `pc`, program counter
- Some status registers for, e.g., flags (carry, zero, ...)

Cortex-M4 assembly basics

- 16 registers, `r0` to `r15`
- 32 bits wide
- Not all can be used freely
  - `r13` is `sp`, stack pointer (don't misuse!)
  - `r14` is `lr`, link register (can be used)
  - `r15` is `pc`, program counter
- Some status registers for, e.g., flags (carry, zero, . . .)
- `Instr Rd, Rn, Rn`, e.g.:
  - `add r2, r0, r1` (three operands)
  - `mov r1, r0` (two operands)

Cortex-M4 assembly basics

- 16 registers, `r0` to `r15`
- 32 bits wide
- Not all can be used freely
  - `r13` is sp, stack pointer (don't misuse!)
  - `r14` is `lr`, link register (can be used)
  - `r15` is pc, program counter
- Some status registers for, e.g., flags (carry, zero, ...)
- `Instr Rd, Rn, Rn`, e.g.:
  - `add r2, r0, r1` (three operands)
  - `mov r1, r0` (two operands)

Details on instructions: ARMv7-M Architecture Reference Manual https://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARMv7-M_ARM.pdf

Instruction summary and timings: Cortex-M4 Technical Reference Manual http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439b/DDI0439B_cortex_m4_r0p0_trm.pdf

A simple example

```
uint32_t accumulate(uint32_t *array, size_t arraylen) {
  size_t i;
  uint32_t r=0;
  for(i=0;i<arraylen;i++) {
    r += array[i];
  }
  return r;
}

int main(void) {
  uint32_t array[1000], sum;

  init(array, 1000);
  sum = accumulate(array, 1000);

  printf("sum: %d\n", sum);
  return sum;
}
```

accumulate in assembly

```
.syntax unified
.cpu cortex-m4

.global accumulate
.type accumulate, %function
accumulate:
    mov r2, #0

    loop:
        cmp r1, #0
        beq done
        ldr r3,[r0]
        add r2,r3
        add r0,#4
        sub r1,#1
        b loop
    done:

    mov r0,r2
    bx lr
```

How fast is it?

- Arithmetic instructions cost 1 cycle
- (Single) loads cost 2 cycles
- Branches cost 1 instruction if branch is not taken
- Branches cost at least 2 cycles if branch is taken

How fast is it?

- Arithmetic instructions cost 1 cycle
- (Single) loads cost 2 cycles
- Branches cost 1 instruction if branch is not taken
- Branches cost at least 2 cycles if branch is taken
- The loop body should cost at least 9 cycles

Speeding it up, part I

```
.syntax unified
.cpu cortex-m4

.global accumulate
.type accumulate, %function
accumulate:
    mov r2, #0

    loop:
        subs r1,#1
        bmi done
        ldr r3,[r0],#4
        add r2,r3
        b loop
    done:

    mov r0,r2
    bx lr
```

What did we do?

- Merge `cmp` and `sub`
- Need `subs` to set flags
- Have `ldr` auto-increase `r0`
- Total saving should be 2 cycles
- Also, code is (marginally) smaller

Speeding it up, part II

```
accumulate:
    push {r4-r12}

    mov r2, #0

    loop1:
        subs r1,#8
        bmi done1
        ldm r0!,{r3-r10}

        add r2,r3
        ...
        add r2,r10

        b loop1
```

```
done1:
add r1,#8

loop2:
    subs r1,#1
    bmi done2
    ldr r3,[r0],#4
    add r2,r3
    b loop2
done2:

pop {r4-r12}
mov r0,r2
bx lr
```

What did we do?

- Use `ldm` ("load multiple") instruction
- Loading N items costs only $N + 1$ cycles
- Need more registers; need to push "caller registers" to the stack (`push`)
- Restore caller registers at the end of the function (`pop`)

What did we do?

- Use `ldm` ("load multiple") instruction
- Loading N items costs only $N + 1$ cycles
- Need more registers; need to push "caller registers" to the stack (`push`)
- Restore caller registers at the end of the function (`pop`)
- Partially unroll to reduce loop-control overhead
- Makes code somewhat larger, various tradeoffs possible
- Lower limit is slightly above 2000 cycles

What did we do?

- Use `ldm` ("load multiple") instruction
- Loading N items costs only $N + 1$ cycles
- Need more registers; need to push "caller registers" to the stack (`push`)
- Restore caller registers at the end of the function (`pop`)
- Partially unroll to reduce loop-control overhead
- Makes code somewhat larger, various tradeoffs possible
- Lower limit is slightly above 2000 cycles
- Ideas for further speedups?

Some useful features of the M4

- We have already seen `ldm`/`stm` instructions

Some useful features of the M4

- We have already seen `ldm`/`stm` instructions
- Large 32×32-bit multiplier with 64-bit result

Some useful features of the M4

- We have already seen `ldm`/`stm` instructions
- Large 32×32-bit multiplier with 64-bit result
- Second input of arithmetic instructions goes through barrel shifter
- Can shift/rotate one input **for free**, e.g.:
  - `eor r0, r1, r2, lsl #2`: left-shift `r2` by 2, xor to `r1`, place result in `r0`
  - `add r2, r0, r1, ror #5`: right-rotate `r1` by 5, add to `r0`, place result in `r2`

Some useful features of the M4

- We have already seen `ldm`/`stm` instructions
- Large 32×32-bit multiplier with 64-bit result
- Second input of arithmetic instructions goes through barrel shifter
- Can shift/rotate one input **for free**, e.g.:
  - `eor r0, r1, r2, lsl #2`: left-shift `r2` by 2, xor to `r1`, place result in `r0`
  - `add r2, r0, r1, ror #5`: right-rotate `r1` by 5, add to `r0`, place result in `r2`
- DSP vector instructions, e.g.:
  - `smuad r0, r1, r2`: $\texttt{r0} \leftarrow \texttt{r1}_L \cdot \texttt{r2}_L + \texttt{r1}_H \cdot \texttt{r2}_H$
  - `smuadx r0, r1, r2`: $\texttt{r0} \leftarrow \texttt{r1}_L \cdot \texttt{r2}_H + \texttt{r1}_H \cdot \texttt{r2}_L$
  - `smlad r0, r1, r2, r3`: $\texttt{r0} \leftarrow \texttt{r1}_L \cdot \texttt{r2}_L + \texttt{r1}_H \cdot \texttt{r2}_H + \texttt{r3}$
  - `smladx r0, r1, r2, r3`: $\texttt{r0} \leftarrow \texttt{r1}_L \cdot \texttt{r2}_H + \texttt{r1}_H \cdot \texttt{r2}_L + \texttt{r3}$

Definition
Post-quantum crypto is (asymmetric) crypto that resists attacks using classical *and quantum* computers.

Post-quantum crypto

Definition
Post-quantum crypto is (asymmetric) crypto that resists attacks using classical *and quantum* computers.

5 main directions

- Lattice-based crypto (PKE and Sigs)
- Code-based crypto (mainly PKE)
- Multivariate-based crypto (mainly Sigs)
- Hash-based signatures (only Sigs)
- Isogeny-based crypto (so far, mainly PKE)

The NIST competition, initial overview

Count of Problem Category	Column Labels		
Row Labels	Key Exchange	Signature	Grand Total
?	1		1
Braids	1	1	2
Chebychev	1		1
Codes	19	5	24
Finite Automata	1	1	2
Hash		4	4
Hypercomplex Numbers	1		1
Isogeny	1		1
Lattice	24	4	28
Mult. Var	6	7	13
Rand. walk	1		1
RSA	1	1	2
Grand Total	**57**	**23**	**80**

Overview tweeted by Jacob Alperin-Sheriff on Dec 4, 2017.

The NIST competition (ctd.)

"Key exchange"

- What is meant is **key encapsulation mechanisms** (KEMs)
  - $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$
  - $(c, k) \leftarrow \mathsf{Encaps}(\mathsf{vk})$
  - $k \leftarrow \mathsf{Decaps}(c, \mathsf{sk})$

The NIST competition (ctd.)

"Key exchange"

- What is meant is **key encapsulation mechanisms** (KEMs)
  - $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}()$
  - $(c, k) \leftarrow \mathsf{Encaps}(\mathsf{vk})$
  - $k \leftarrow \mathsf{Decaps}(c, \mathsf{sk})$

Status of the NIST competition

- In total 69 submissions accepted as "complete and proper"
- Several broken, 5 withdrawn
- Jan 2019: NIST announces 26 round-2 candidates
  - 17 KEMs and PKEs
  - 9 signature schemes

- Joint work with
  Matthias Kannwischer, Joost Rijneveld, and Ko Stoffelen.
- Started as part of PQCRYPTO H2020 project
- Continued within EPOQUE ERC StG
- Library and testing/benchmarking framework
  - PQ-crypto on ARM Cortex-M4
  - Uses STM32F4 Discovery board
  - 192 KB of RAM, benchmarks at 24 MHz
- Easy to add schemes using NIST API
- Optimized SHA3 and AES shared across primitives

pqm4 usage

- Run functional tests of all primitives and implementations:

```
python3 test.py
```

pqm4 usage

- Run functional tests of all primitives and implementations:

```
python3 test.py
```

- Generate testvectors, compare for consistency (also with host):

```
python3 testvectors.py
```

pqm4 usage

- Run functional tests of all primitives and implementations:

```
python3 test.py
```

- Generate testvectors, compare for consistency (also with host):

```
python3 testvectors.py
```

- Run speed and stack benchmarks:

```
python3 benchmarks.py
```

pqm4 usage

- Run functional tests of all primitives and implementations:

```
python3 test.py
```

- Generate testvectors, compare for consistency (also with host):

```
python3 testvectors.py
```

- Run speed and stack benchmarks:

```
python3 benchmarks.py
```

- Easy to evaluate only subset of schemes, e.g.:

```
python3 test.py newhope1024cca sphincs-shake256-128s
```

Signatures (not) in pqm4

CRYSTALS-Dilithium	✓
FALCON	✓
GeMSS	✗
LUOV	✓
MQDSS	✓
Picnic	✗
qTESLA	✓
Rainbow	✗
SPHINCS+	✓

KEMs (not) in pqm4

	ref/clean	opt
BIKE	—	—
Classic McEliece	✗	✗
CRYSTALS-Kyber	✓	✓
Frodo-KEM	✓	(✓)
HQC	—	—
LAC	✓	—
LEDAcrypt	WIP	WIP
NewHope	✓	✓
NTRU	✓	✓
NTRU Prime	✓	—
NTS-KEM	✗	✗
ROLLO	—	—
Round5	WIP	✓
RQC	—	—
SABER	✓	✓
SIKE	✓	—
ThreeBears	✓	(✓)

KEMs (not) in pqm4

	ref/clean	opt
BIKE	—	—
Classic McEliece	✗	✗
CRYSTALS-Kyber	✓	✓
Frodo-KEM	✓	(✓)
HQC	—	—
LAC	✓	—
LEDAcrypt	WIP	WIP
NewHope	✓	✓
NTRU	✓	✓
NTRU Prime	✓	—
NTS-KEM	✗	✗
ROLLO	—	—
Round5	WIP	✓
RQC	—	—
SABER	✓	✓
SIKE	✓	—
ThreeBears	✓	(✓)

Learning with errors (LWE)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given "noise distribution" χ
- Given samples $\mathbf{As} + \mathbf{e}$, with $\mathbf{e} \leftarrow \chi$

Learning with errors (LWE)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given "noise distribution" χ
- Given samples $\mathbf{As} + \mathbf{e}$, with $\mathbf{e} \leftarrow \chi$
- Search version: find $\mathbf{s}$
- Decision version: distinguish from uniform random

Learning with errors (LWE)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given "noise distribution" χ
- Given samples $\mathbf{A}\mathbf{s} + \mathbf{e}$, with $\mathbf{e} \leftarrow \chi$
- Search version: find $\mathbf{s}$
- Decision version: distinguish from uniform random
- Structured lattices: work in $\mathbb{Z}_q[x]/f$

Learning with rounding (LWR)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given samples $\lceil \mathbf{As} \rfloor_p$, with $p < q$

Learning with rounding (LWR)

- Given uniform $\mathbf{A} \in \mathbb{Z}_q^{k \times \ell}$
- Given samples $\lceil \mathbf{As} \rfloor_p$, with $p < q$
- Search version: find $\mathbf{s}$
- Decision version: distinguish from uniform random
- Structured lattices: work in $\mathbb{Z}_q[x]/f$

Lattice-based KEMs – the basic idea

Alice (server)		Bob (client)
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi$		$\mathbf{s}', \mathbf{e}' \xleftarrow{\$} \chi$
$\mathbf{b} \leftarrow \mathbf{a}\mathbf{s} + \mathbf{e}$	$\xrightarrow{\mathbf{b}}$	$\mathbf{u} \leftarrow \mathbf{a}\mathbf{s}' + \mathbf{e}'$
	$\xleftarrow{\mathbf{u}}$	

Alice has $\mathbf{v} = \mathbf{u}\mathbf{s} = \mathbf{a}\mathbf{s}\mathbf{s}' + \mathbf{e}'\mathbf{s}$

Bob has $\mathbf{v}' = \mathbf{b}\mathbf{s}' = \mathbf{a}\mathbf{s}\mathbf{s}' + \mathbf{e}\mathbf{s}'$

- Secret and noise $\mathbf{s}, \mathbf{s}', \mathbf{e}, \mathbf{e}'$ are small
- $\mathbf{v}$ and $\mathbf{v}'$ are *approximately* the same

Core operation: multiplication in $\mathcal{R}_q = \mathbb{Z}_q[X]/f$

Power-of-two q

- Several schemes use $q = 2^m$, for small m
- Examples: Round5, NTRU, Saber
- More round-1 examples: Kindi, RLizard

Core operation: multiplication in $\mathcal{R}_q = \mathbb{Z}_q[X]/f$

Power-of-two q

- Several schemes use $q = 2^m$, for small m
- Examples: Round5, NTRU, Saber
- More round-1 examples: Kindi, RLizard

Prime "NTT-friendly" q

- Kyber and NewHope use prime q supporting fast NTT
- For $A, B \in \mathcal{R}_q$, $A \cdot B = \mathrm{NTT}^{-1}(\mathrm{NTT}(A) \circ \mathrm{NTT}(B))$
- NTT is Fourier Transform over finite field
- Use $f = X^n + 1$ for power-of-two n

Multiplication in $\mathbb{Z}_{2^m}[X]$

- Joint work with **Matthias Kannwischer** and **Joost Rijneveld**
- Represent coefficients as 16-bit integers
- No modular reductions required, 2^{16} is a multiple of $q = 2^m$

Multiplication in $\mathbb{Z}_{2^m}[X]$

- Joint work with **Matthias Kannwischer** and **Joost Rijneveld**
- Represent coefficients as 16-bit integers
- No modular reductions required, 2^{16} is a multiple of $q = 2^m$
- Schoolbook multiplication takes n^2 integer muls, $(n-1)^2$ adds

Multiplication in $\mathbb{Z}_{2^m}[X]$

- Joint work with **Matthias Kannwischer** and **Joost Rijneveld**
- Represent coefficients as 16-bit integers
- No modular reductions required, 2^{16} is a multiple of $q = 2^m$
- Schoolbook multiplication takes n^2 integer muls, $(n-1)^2$ adds
- Can do better using Karatsuba:

$$\begin{aligned}
&(a_\ell + X^k a_h) \cdot (b_\ell + X^k b_h) \\
= \; & a_\ell b_\ell + X^k(a_\ell b_h + a_h b_\ell) + X^n a_h b_h \\
= \; & a_\ell b_\ell + X^k((a_\ell + a_h)(b_\ell + b_h) - a_\ell b_\ell - a_h b_h) + X^n a_h b_h
\end{aligned}$$

- Recursive application yields complexity $\Theta(n^{\log_2 3})$

Multiplication in $\mathbb{Z}_{2^m}[X]$

- Joint work with **Matthias Kannwischer** and **Joost Rijneveld**
- Represent coefficients as 16-bit integers
- No modular reductions required, 2^{16} is a multiple of $q = 2^m$
- Schoolbook multiplication takes n^2 integer muls, $(n-1)^2$ adds
- Can do better using Karatsuba:

$$\begin{aligned} & (a_\ell + X^k a_h) \cdot (b_\ell + X^k b_h) \\ = \; & a_\ell b_\ell + X^k(a_\ell b_h + a_h b_\ell) + X^n a_h b_h \\ = \; & a_\ell b_\ell + X^k((a_\ell + a_h)(b_\ell + b_h) - a_\ell b_\ell - a_h b_h) + X^n a_h b_h \end{aligned}$$

- Recursive application yields complexity $\Theta(n^{\log_2 3})$
- Generalization: Toom-Cook
  - Toom-3: split into 5 multiplications of 1/3 size
  - Toom-4: split into 7 multiplications of 1/4 size
- Approach: Evaluate, multiply, interpolate

Initial observations

- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials

Initial observations

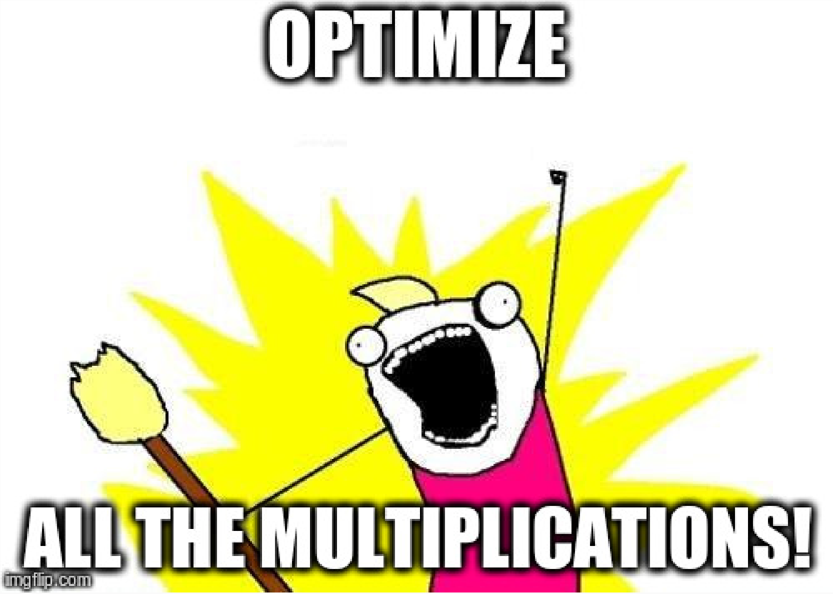
- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials
- Toom-3 needs division by 2, loses 1 bit of precision
- Toom-4 needs division by 8, loses 3 bits of precision
- This limits recursive application when using 16-bit integers
- Can use Toom-4 only for $q \leq 2^{13}$

Initial observations

- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials
- Toom-3 needs division by 2, loses 1 bit of precision
- Toom-4 needs division by 8, loses 3 bits of precision
- This limits recursive application when using 16-bit integers
- Can use Toom-4 only for $q \leq 2^{13}$
- Karmakar, Bermudo Mera, Sinha Roy, Verbauwhede (CHES 2018):
  - Optimize Saber, $q = 2^{13}$, $n = 256$
  - Use Toom-4 + two levels of Karatsuba
  - Optimized 16-coefficient schoolbook multiplication

Initial observations

- Karatsuba/Toom is asymptotically faster, but isn't for "small" polynomials
- Toom-3 needs division by 2, loses 1 bit of precision
- Toom-4 needs division by 8, loses 3 bits of precision
- This limits recursive application when using 16-bit integers
- Can use Toom-4 only for $q \leq 2^{13}$
- Karmakar, Bermudo Mera, Sinha Roy, Verbauwhede (CHES 2018):
    - Optimize Saber, $q = 2^{13}, n = 256$
    - Use Toom-4 + two levels of Karatsuba
    - Optimized 16-coefficient schoolbook multiplication
- **Is this the best approach? How about other values of q and n?**


OPTIMIZE
ALL THE MULTIPLICATIONS!
imgflip.com

Our approach

- Generate optimized assembly for Karatsuba/Toom
- Use Python scripts, receive as input n and q
- Hand-optimize “small” schoolbook multiplications
    - Make heavy use of DSP “vector instructions”
    - Perform two 16×16-bit multiply-accumulate in one cycle
    - Carefully schedule instructions to minimize loads/stores
- Benchmark different options, pick fastest

Multiplication results

	approach	"small"	cycles	stack
Saber ($n = 256,\ q = 2^{13}$)	Karatsuba only	16	41 121	2 020
	Toom-3	11	41 225	3 480
	Toom-4	**16**	**39 124**	**3 800**
	Toom-4 + Toom-3	-	-	-
Kindi-256-3-4-2 ($n = 256,\ q = 2^{14}$)	**Karatsuba only**	**16**	**41 121**	**2 020**
	Toom-3	11	41 225	3 480
	Toom-4	-	-	-
	Toom-4 + Toom-3	-	-	-
NTRU-HRSS ($n = 701,\ q = 2^{13}$)	Karatsuba only	11	230 132	5 676
	Toom-3	15	217 436	9 384
	Toom-4	**11**	**182 129**	**10 596**
	Toom-4 + Toom-3	-	-	-
NTRU-KEM-743 ($n = 743,\ q = 2^{11}$)	Karatsuba only	12	247 489	6 012
	Toom-3	16	219 061	9 920
	Toom-4	**12**	**196 940**	**11 208**
	Toom-4 + Toom-3	16	197 227	12 152
RLizard-1024 ($n = 1024,\ q = 2^{11}$)	Karatsuba only	16	400 810	8 188
	Toom-3	11	360 589	13 756
	Toom-4	**16**	**313 744**	**15 344**
	Toom-4 + Toom-3	11	315 788	16 816

NTT-based multiplication

- Joint work with **Leon Botros** and **Matthias Kannwischer**
- Primary goal: optimize Kyber
- Secondary effect: optimize NewHope (improved by Gérard)

NTT-based multiplication

- Joint work with **Leon Botros** and **Matthias Kannwischer**
- Primary goal: optimize Kyber
- Secondary effect: optimize NewHope (improved by Gérard)
- NTT is an FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(X^2) + X f_1(X^2)$

NTT-based multiplication

- Joint work with **Leon Botros** and **Matthias Kannwischer**
- Primary goal: optimize Kyber
- Secondary effect: optimize NewHope (improved by Gérard)
- NTT is an FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(X^2) + X f_1(X^2)$
  - Huge overlap between evaluating

$$f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \text{ and}$$
$$f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)$$

NTT-based multiplication

- Joint work with **Leon Botros** and **Matthias Kannwischer**
- Primary goal: optimize Kyber
- Secondary effect: optimize NewHope (improved by Gérard)
- NTT is an FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(X^2) + X f_1(X^2)$
  - Huge overlap between evaluating

$$f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \text{ and}$$
$$f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)$$

  - f_0 has $n/2$ coefficients
  - Evaluate f_0 at all $(n/2)$-th roots of unity by recursive application
  - Same for f_1

NTT-based multiplication

- First thing to do: replace recursion by iteration
- Loop over $\log n$ levels with $n/2$ “butterflies” each

NTT-based multiplication

- First thing to do: replace recursion by iteration
- Loop over $\log n$ levels with $n/2$ "butterflies" each
- Butterfly on level k:
  - Pick up f_i and f_{i+2^k}
  - Multiply f_{i+2^k} by a power of ω to obtain t
  - Compute $f_{i+2^k} \leftarrow a_i - t$
  - Compute $f_i \leftarrow a_i + t$

NTT-based multiplication

- First thing to do: replace recursion by iteration
- Loop over $\log n$ levels with $n/2$ "butterflies" each
- Butterfly on level k:
  - Pick up f_i and f_{i+2^k}
  - Multiply f_{i+2^k} by a power of ω to obtain t
  - Compute $f_{i+2^k} \leftarrow a_i - t$
  - Compute $f_i \leftarrow a_i + t$
- Main optimizations on Cortex-M4:
  - "Merge" levels: fewer loads/stores
  - Optimize modular arithmetic (precompute powers of ω in Montgomery domain)
  - Lazy reductions
  - Carefully optimize using DSP instructions

Selected optimized lattice KEM cycles

Scheme	Key Generation	Encapsulation	Decapsulation
ntruhps2048509	77 698 713	645 329	542 439
ntruhps2048677	144 383 491	955 902	836 959
ntruhps4096821	211 758 452	1 205 662	1 066 879
ntruhrss701	154 676 705	402 784	890 231
lightsaber	459 965	651 273	678 810
saber	896 035	1 161 849	1 204 633
firesaber	1 448 776	1 786 930	1 853 339
kyber512	514 291	652 769	621 245
kyber768	976 757	1 146 556	1 094 849
kyber1024	1 575 052	1 779 848	1 709 348
newhope1024cpa	975 736	975 452	162 660
newhope1024cca	1 161 112	1 777 918	1 760 470

Comparison: Curve25519 scalarmult: 625 358 cycles

Selected optimized lattice KEM stack bytes

Scheme	Key Generation	Encapsulation	Decapsulation
ntruhps2048509	21 412	15 452	14 828
ntruhps2048677	28 524	20 604	19 756
ntruhps4096821	34 532	24 924	23 980
ntruhrss701	27 580	19 372	20 580
lightsaber	9 656	11 392	12 136
saber	13 256	15 544	16 640
firesaber	20 144	23 008	24 592
kyber512	2 952	2 552	2 560
kyber768	3 848	3 128	3 072
kyber1024	4 360	3 584	3 592
newhope1024cpa	11 096	17 288	8 308
newhope1024cca	11 080	17 360	19 576

Resources online

- Cortex-M4 examples (including `accumulate`): https://cryptojedi.org/peter/data/stm32f4examples.tar.bz2
- `pqm4` library and benchmarking suite: https://github.com/mupq/pqm4
- `pqriscv` library and benchmarking suite: https://github.com/mupq/pqriscv
- Code of $\mathbb{Z}_{2^m}[x]$ multiplication paper, including scripts: https://github.com/mupq/polymul-z2mx-m4
- $\mathbb{Z}_{2^m}[x]$ multiplication paper: https://cryptojedi.org/papers/#latticem4
- Kyber optimization paper: https://cryptojedi.org/papers/#nttm4