MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

Optimizing crypto on embedded microcontrollers

Peter Schwabe

October 4, 2020

Outline

1. embedded microcontrollers
2. optimizing
3. crypto

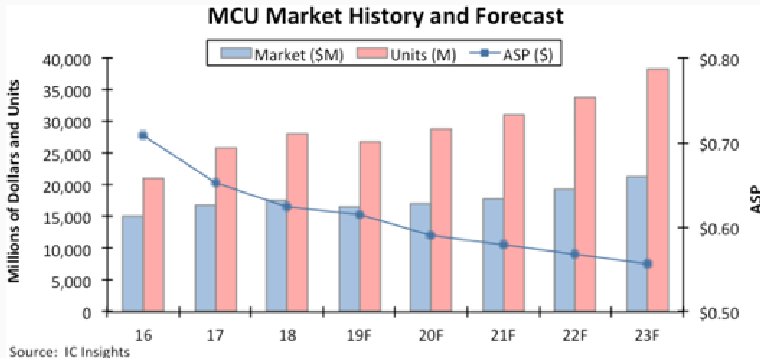
Embedded microcontrollers

"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



. . . so many to choose from!

- 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

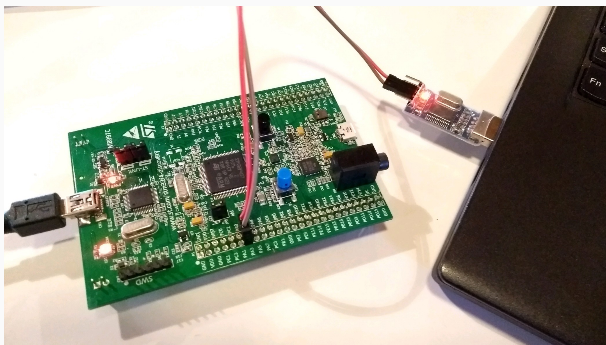
. . . so many to choose from!

- 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 <30 EUR from various vendors (e.g., ebay, Mouser, myMCU):
  https://shop.mymcu.de/index.php?sp=article.sp.php&artID=200167
- Additionally need USB-TTL converter and mini-USB cable

Getting started: Hello world!

```c
#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

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 $\approx$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://foobt.net/spring2020/csci10_7446/files/arm_cortexm4_processor_trm_100166_0001_00_en.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?

Optimizing “something” vs. optimizing crypto

- So far there was nothing crypto-specific in this tutorial
- Is optimizing crypto the same as optimizing any other software?

Optimizing “something” vs. optimizing crypto

- So far there was nothing crypto-specific in this tutorial
- Is optimizing crypto the same as optimizing any other software?
- No.

Optimizing "something" vs. optimizing crypto

- So far there was nothing crypto-specific in this tutorial
- Is optimizing crypto the same as optimizing any other software?
- No. Cryptographic software deals with secret data (e.g., keys)
- Information about secret data must not leak through side channels

Optimizing “something” vs. optimizing crypto

- So far there was nothing crypto-specific in this tutorial
- Is optimizing crypto the same as optimizing any other software?
- No. Cryptographic software deals with secret data (e.g., keys)
- Information about secret data must not leak through side channels
- For today, only consider timing side-channel:
  - Can be exploited **remotely**
  - Can eliminate systematically through “constant-time” code

Optimizing “something” vs. optimizing crypto

- So far there was nothing crypto-specific in this tutorial
- Is optimizing crypto the same as optimizing any other software?
- No. Cryptographic software deals with secret data (e.g., keys)
- Information about secret data must not leak through side channels
- For today, only consider timing side-channel:
  - Can be exploited **remotely**
  - Can eliminate systematically through “constant-time” code
  - Generic techniques to write constant-time code
  - Performance penalty highly algorithm-dependent

Timing leakage part I

- Consider the following piece of code:

  if s **then**

  $r \leftarrow A$

  else

  $r \leftarrow B$

  end if

Timing leakage part I

- Consider the following piece of code:

  if s **then**
      $r \leftarrow A$
  else
      $r \leftarrow B$
  end if

- General structure of any conditional branch
- A and B can be large computations, r can be a large state

Timing leakage part I

- Consider the following piece of code:

  if s **then**

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if

- General structure of any conditional branch
- A and B can be large computations, r can be a large state
- This code takes different amount of time, depending on s
- Obvious timing leak if s is secret

Timing leakage part I

- Consider the following piece of code:

  if s **then**
      $r \leftarrow A$
  else
      $r \leftarrow B$
  end if

- General structure of any conditional branch
- A and B can be large computations, r can be a large state
- This code takes different amount of time, depending on s
- Obvious timing leak if s is secret
- Even if A and B take the same amount of cycles this is *generally not* constant time!
- Reasons: Branch prediction, instruction-caches
- **Never use secret-data-dependent branch conditions**

Eliminating branches

- So, what do we do with this piece of code?

  if s **then**

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if

Eliminating branches

- So, what do we do with this piece of code?

  if s **then**

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if

- Replace by

$$r \leftarrow sA + (1 - s)B$$

Eliminating branches

- So, what do we do with this piece of code?

  if s **then**

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if

- Replace by

$$r \leftarrow sA + (1 - s)B$$

- Can expand s to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication

Eliminating branches

- So, what do we do with this piece of code?

  if s **then**
      $r \leftarrow A$
  else
      $r \leftarrow B$
  end if

- Replace by

$$r \leftarrow sA + (1 - s)B$$

- Can expand s to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication
- For very fast A and B this can even be faster

How about caches?

"Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, and Cortex-M4 processors do not have any internal cache memory.

—ARM Cortex-M Programming Guide to Memory Barrier Instructions

How about caches?

"Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, and Cortex-M4 processors do not have any internal cache memory. However, it is possible for a SoC design to integrate a system level cache."

—ARM Cortex-M Programming Guide to Memory Barrier Instructions

How about caches?

"The memory system is configured during implementation and can include instruction and data caches of varying sizes."

—ARM Cortex-M7 TRM

Timing leakage part II

$T[0] \dots T[15]$
$T[16] \dots T[31]$
$T[32] \dots T[47]$
$T[48] \dots T[63]$
$T[64] \dots T[79]$
$T[80] \dots T[95]$
$T[96] \dots T[111]$
$T[112] \dots T[127]$
$T[128] \dots T[143]$
$T[144] \dots T[159]$
$T[160] \dots T[175]$
$T[176] \dots T[191]$
$T[192] \dots T[207]$
$T[208] \dots T[223]$
$T[224] \dots T[239]$
$T[240] \dots T[255]$

- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache

Timing leakage part II

$T[0] \dots T[15]$
$T[16] \dots T[31]$
attacker's data
attacker's data
$T[64] \dots T[79]$
$T[80] \dots T[95]$
attacker's data
attacker's data
attacker's data
attacker's data
$T[160] \dots T[175]$
$T[176] \dots T[191]$
$T[192] \dots T[207]$
$T[208] \dots T[223]$
attacker's data
attacker's data

- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines

Timing leakage part II

$T[0]\dots T[15]$
$T[16]\dots T[31]$
???
???
$T[64]\dots T[79]$
$T[80]\dots T[95]$
???
???
???
???
$T[160]\dots T[175]$
$T[176]\dots T[191]$
$T[192]\dots T[207]$
$T[208]\dots T223]$
???
???

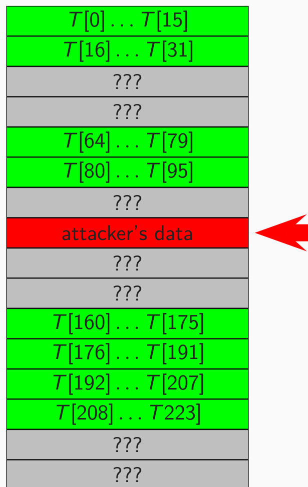
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again

Timing leakage part II

Cache lines
$T[0] \dots T[15]$
$T[16] \dots T[31]$
???
???
$T[64] \dots T[79]$
$T[80] \dots T[95]$
???
???
???
???
$T[160] \dots T[175]$
$T[176] \dots T[191]$
$T[192] \dots T[207]$
$T[208] \dots T223]$
???
???

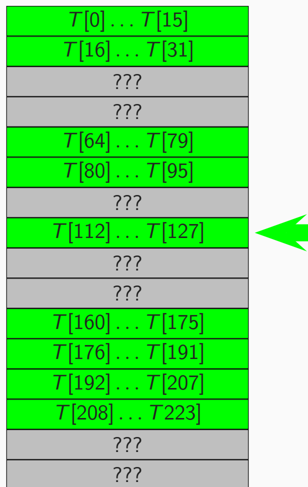
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads data:

Timing leakage part II



- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads data:
  - Fast: cache hit (crypto did not just load from this line)

Timing leakage part II



- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads data:
  - Fast: cache hit (crypto did not just load from this line)
  - Slow: cache miss (crypto just loaded from this line)

Some comments on cache-timing

- This is only the *most basic* cache-timing attack

Some comments on cache-timing

- This is only the *most basic* cache-timing attack
- Non-secret cache lines are not enough for security
- In general, load/store addresses influence timing in many ways
- **Do not access memory at secret-data-dependent addresses** (maybe with the exception of very low-end MCUs?)

Some comments on cache-timing

- This is only the *most basic* cache-timing attack
- Non-secret cache lines are not enough for security
- In general, load/store addresses influence timing in many ways
- **Do not access memory at secret-data-dependent addresses**
  (maybe with the exception of very low-end MCUs?)
- Timing attacks are practical:
  Osvik, Tromer, Shamir, 2006: 65 ms to steal a 256-bit AES key used for Linux hard-disk encryption

Some comments on cache-timing

- This is only the *most basic* cache-timing attack
- Non-secret cache lines are not enough for security
- In general, load/store addresses influence timing in many ways
- **Do not access memory at secret-data-dependent addresses** (maybe with the exception of very low-end MCUs?)
- Timing attacks are practical:
  Osvik, Tromer, Shamir, 2006: 65 ms to steal a 256-bit AES key used for Linux hard-disk encryption
- *Remote* timing attacks are practical:
  Brumley, Tuveri, 2011: A few minutes to steal ECDSA signing key from OpenSSL implementation

Eliminating lookups

- Want to load item at (secret) position p from table of size n

Eliminating lookups

- Want to load item at (secret) position p from table of size n
- Load all items, use arithmetic to pick the right one:

  for i from 0 to $n-1$ **do**
  $\quad d \leftarrow T[i]$
  $\quad$**if** $p = i$ **then**
  $\quad\quad r \leftarrow d$
  $\quad$**end if**
  end for

Eliminating lookups

- Want to load item at (secret) position p from table of size n
- Load all items, use arithmetic to pick the right one:

  for i from 0 to $n-1$ **do**
      $d \leftarrow T[i]$
      if $p = i$ **then**
          $r \leftarrow d$
      end if
  end for
- Problem 1: if-statements are not constant time (see before)

Eliminating lookups

- Want to load item at (secret) position p from table of size n
- Load all items, use arithmetic to pick the right one:

  for i from 0 to $n-1$ **do**
  $\quad d \leftarrow T[i]$
  $\quad$ **if** $p = i$ **then**
  $\quad\quad r \leftarrow d$
  $\quad$ **end if**
  end for

- Problem 1: if-statements are not constant time (see before)
- Problem 2: Comparisons in C may be variable time, replace by, e.g.:

```
static unsigned long long eq(uint32_t a, uint32_t b)
{
  unsigned long long t = a ^ b;
  t = (-t) >> 63;
  return 1-t;
}
```

Eliminating lookups

- Want to load item at (secret) position p from table of size n
- Load all items, use arithmetic to pick the right one:

  for i from 0 to $n-1$ **do**
      $d \leftarrow T[i]$
      if $p = i$ **then**
          $r \leftarrow d$
      end if
  end for

- Problem 1: if-statements are not constant time (see before)
- Problem 2: Comparisons in C may be variable time, replace by, e.g.:

```
static unsigned long long eq(uint32_t a, uint32_t b)
{
  unsigned long long t = a ^ b;
  t = (-t) >> 63;
  return 1-t;
}
```

- Of course much easier: do it in assembly ;-)

Is that all? (Timing leakage part III)

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can *always* be done; cost highly depends on the algorithm

Is that all? (Timing leakage part III)

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can *always* be done; cost highly depends on the algorithm
- On supported platforms, test this with https://www.post-apocalyptic-crypto.org/timecop/

Is that all? (Timing leakage part III)

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can *always* be done; cost highly depends on the algorithm
- On supported platforms, test this with `https://www.post-apocalyptic-crypto.org/timecop/`

"In order for a function to be constant time, the branches taken and memory addresses accessed must be independent of any secret inputs. (That's assuming that the fundamental processor instructions are constant time, but that's true for all sane CPUs.)"

—Langley, Apr. 2010

Is that all? (Timing leakage part III)

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can *always* be done; cost highly depends on the algorithm
- On supported platforms, test this with https://www.post-apocalyptic-crypto.org/timecop/

"In order for a function to be constant time, the branches taken and memory addresses accessed must be independent of any secret inputs. (That's assuming that the fundamental processor instructions are constant time, but that's true for all sane CPUs.)"

—Langley, Apr. 2010

"So the argument to the `DIV` instruction was smaller and `DIV`, on Intel, takes a variable amount of time depending on its arguments!"

—Langley, Feb. 2013

Dangerous arithmetic (examples)

- `DIV`, `IDIV`, `FDIV` on pretty much all Intel/AMD CPUs
- Various math instructions on Intel/AMD CPUs (`FSIN`, `FCOS`...)

Dangerous arithmetic (examples)

- `DIV`, `IDIV`, `FDIV` on pretty much all Intel/AMD CPUs
- Various math instructions on Intel/AMD CPUs (`FSIN`, `FCOS`...)
- `MUL`, `MULHW`, `MULHWU` on many PowerPC CPUs
- `UMULL`, `SMULL`, `UMLAL`, and `SMLAL` on ARM Cortex-M3.

Dangerous arithmetic (examples)

- `DIV`, `IDIV`, `FDIV` on pretty much all Intel/AMD CPUs
- Various math instructions on Intel/AMD CPUs (`FSIN`, `FCOS`...)
- `MUL`, `MULHW`, `MULHWU` on many PowerPC CPUs
- `UMULL`, `SMULL`, `UMLAL`, and `SMLAL` on ARM Cortex-M3.

Solution

- Avoid these instructions
- Make sure that inputs to the instructions don't leak timing information (very tricky!)

“Homework”: Optimize ChaCha20

- Stream cipher proposed by Bernstein in 2008
- Variant of Salsa20 from the eSTREAM software portfolio
- Has a state of 64 bytes, 4×4 matrix of 32-bit words
- Generates random stream in 64-byte blocks, works on 32-bit integers
- Per block: 20 rounds; each round doing 16 add-xor-rotate sequences, such as

```
a += b;
d = (d ^ a) <<< 16;
```

“Homework”: Optimize ChaCha20

- Stream cipher proposed by Bernstein in 2008
- Variant of Salsa20 from the eSTREAM software portfolio
- Has a state of 64 bytes, 4×4 matrix of 32-bit words
- Generates random stream in 64-byte blocks, works on 32-bit integers
- Per block: 20 rounds; each round doing 16 add-xor-rotate sequences, such as

```
a += b;
d = (d ^ a) <<< 16;
```

- Strategy for optimizing on the M4
  - Write `quarterround` function in assembly
  - Merge 4 `quarterround` functions into a full round
  - Implement loop over 20 rounds in assembly
  - (Implement loop over message length in assembly)

Useful features of the M4

- 16 state words won't fit into registers, you need the stack
  - Use `push` and `pop`
  - Can also use `ldr` and `str`, `ldm`, `stm`
  - For example: `push {r0,r1}` is the same as `stmdb sp!, {r0,r1}`

Useful features of the M4

- 16 state words won't fit into registers, you need the stack
    - Use `push` and `pop`
    - Can also use `ldr` and `str`, `ldm`, `stm`
    - For example: `push {r0,r1}` is the same as `stmdb sp!, {r0,r1}`
- Second input of arithmetic instructions goes through barrel shifter
- Can shift/rotate one input **for free**
- Examples:
    - `eor r0, r1, r2, lsl #2`: left-shift `r2` by 2, xor to `r1`, store result in `r0`
    - `add r2, r0, r1, ror #5`: right-rotate `r1` by 5, add to `r0`, store result in `r2`

Getting started

- Download
  https://cryptojedi.org/peter/data/stm32f4examples.tar.bz2
- Unpack: `tar xjvf stm32f4examples.tar.bz2`
- Connect STM32F4 Discovery board with Mini-USB cable
- Connect USB-TTL: RX to PA2, TX to PA3
- Open terminal, run `host_unidirectional.py`
- Build some project, e.g., accumulate using make
- Flash accumulate1.bin to the board:

```
st-flash write accumulate1.bin 0x8000000
```

- Push "reset" button to start/restart program
- Now go for ChaCha20

pqm4: post-quantum crypto on the M4

- Joint work with
  Matthias Kannwischer, Joost Rijneveld, and Ko Stoffelen.
- Library and testing/benchmarking framework
- Easy to add schemes using NIST API
- Optimized SHA3 shared across primitives

pqm4: post-quantum crypto on the M4

- Joint work with
  Matthias Kannwischer, Joost Rijneveld, and Ko Stoffelen.
- Library and testing/benchmarking framework
- Easy to add schemes using NIST API
- Optimized SHA3 shared across primitives
- 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
```

pqm4 status for KEMs

NIST PQC finalist

	reference	optimized
Classic McEliece	✗$_{Key}$	—
CRYSTALS-Kyber	✓	✓
NTRU	✓	✓
SABER	✓	✓

NIST PQC alternate candidates

	reference	optimized
BIKE	✗$_{Lib}$	—
Frodo-KEM	✓	✓
HQC	WIP (?)	—
NTRU Prime	✓	✓
SIKE	✓	✓

pqm4 status for signatures

NIST PQC finalist

	reference	optimized
CRYSTALS-Dilithium	✓	(✓)
FALCON	✓	✓
Rainbow	WIP	WIP

NIST PQC alternate candidates

	reference	optimized
GeMSS	✗$_{Key}$	—
Picnic	✗$_{RAM}$	—
SPHINCS+	✓	—

https://github.com/mupq/pqm4