Engineering Cryptographic Software

Cryptography on the Arm Cortex-M4

Peter Schwabe

January 2026

Optimizing cryptographic software

- Primary goal: Make software as fast as possible
- Main constraint: don’t leak secrets (more later)

Optimizing cryptographic software

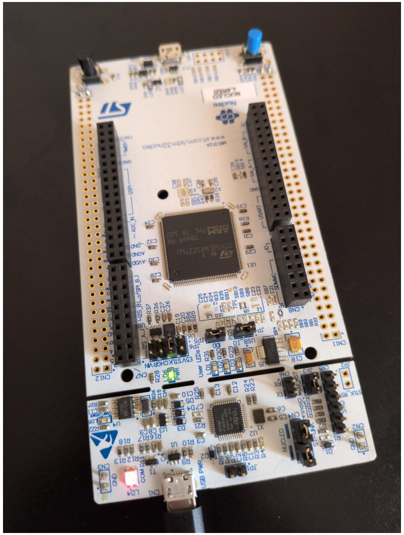
- Primary goal: Make software as fast as possible
- Main constraint: don't leak secrets (more later)
- Optimization involves:
  - High-level algorithmics
  - Data representation and low-level algorithmics
  - Assembly-level optimization for target platform

Optimizing cryptographic software

- Primary goal: Make software as fast as possible
- Main constraint: don't leak secrets (more later)
- Optimization involves:
  - High-level algorithmics
  - Data representation and low-level algorithmics
  - Assembly-level optimization for target platform
- Levels of optimization are typically *not* independent
- For this course we want
  - Predictable and easy target platform
  - Still somewhat "interesting" for low-level optimization
  - Consider limited, but not trivial, attacker/leakage

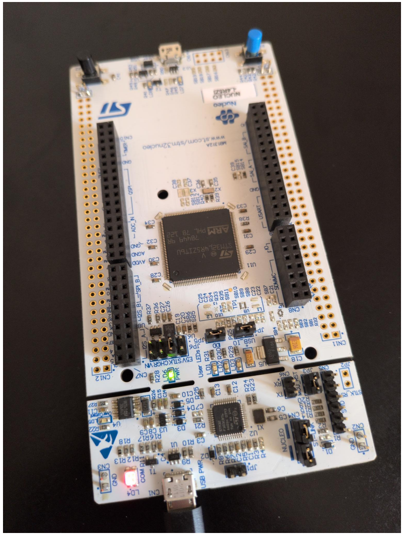
Our target platform

- Arm Cortex-M4 on STM32Nucleo-L4R5ZI board
- Implements the ARMv7E-M architecture
- 640 KB RAM, 2 MB Flash (ROM)
- Maximum CPU frequency of 120 MHz



Our target platform

- Arm Cortex-M4 on STM32Nucleo-L4R5ZI board
- Implements the ARMv7E-M architecture
- 640 KB RAM, 2 MB Flash (ROM)
- Maximum CPU frequency of 120 MHz
- Available for ≈USD 20 (<1000 MUR) from, e.g., Mouser: `https://www2.mouser.com/ProductDetail/STMicroelectronics/NUCLEO-L4R5ZI?qs=j%252B1pi9TdxUYHwRjgL7zLGg%3D%3D`
- Additionally need micro-USB cable



Getting started: Hello world!

```
#!/usr/bin/env python3

print("Hello world!")
```

Getting started: Hello world!

```
#!/usr/bin/env python3

print("Hello world!")
```

- This would need a Python interpreter (we dont' have that)

Getting started: Hello world!

```
#!/usr/bin/env python3

print("Hello world!")
```

- This would need a Python interpreter (we dont' have that)
- Probably would also need an operating system (we don't have that, either)

Getting started: Hello world! (next attempt)

```c
#include <stdio.h>

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

Getting started: Hello world! (next attempt)

```
#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! (next attempt)

```c
#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! (next attempt)

```
#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! (next attempt)

```
#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! (next attempt)

```
#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: `gcc-arm-none-eabi`

Fixing all of those issues: the idea

1. Install a cross compiler: `gcc-arm-none-eabi`
2. Install `openocd` to communicate with the board

Fixing all of those issues: the idea

1. Install a cross compiler: `gcc-arm-none-eabi`
2. Install `openocd` to communicate with the board
3. Extend `hello.c` with some setup boilerplate code
    - Initialize CPU and set clock frequency
    - Set up serial port (USART) through USB

Fixing all of those issues: the idea

1. Install a cross compiler: `gcc-arm-none-eabi`
2. Install `openocd` to communicate with the board
3. Extend `hello.c` with some setup boilerplate code
   - Initialize CPU and set clock frequency
   - Set up serial port (USART) through USB
4. Replace `printf` with code that sends `"Hello World!"` through serial

Fixing all of those issues: the idea

1. Install a cross compiler: `gcc-arm-none-eabi`
2. Install `openocd` to communicate with the board
3. Extend `hello.c` with some setup boilerplate code
   - Initialize CPU and set clock frequency
   - Set up serial port (USART) through USB
4. Replace `printf` with code that sends `"Hello World!"` through serial
5. Compile to ARM **binary** (not ELF) file, say `hello.bin`

Fixing all of those issues: the idea

1. Install a cross compiler: `gcc-arm-none-eabi`
2. Install `openocd` to communicate with the board
3. Extend `hello.c` with some setup boilerplate code
   - Initialize CPU and set clock frequency
   - Set up serial port (USART) through USB
4. Replace `printf` with code that sends `"Hello World!"` through serial
5. Compile to ARM **binary** (not ELF) file, say `hello.bin`
6. Connect board through micro-USB cable
7. On the host side (your laptop), start serial-port listener

Fixing all of those issues: the idea

1. Install a cross compiler: `gcc-arm-none-eabi`
2. Install `openocd` to communicate with the board
3. Extend `hello.c` with some setup boilerplate code
   - Initialize CPU and set clock frequency
   - Set up serial port (USART) through USB
4. Replace `printf` with code that sends `"Hello World!"` through serial
5. Compile to ARM **binary** (not ELF) file, say `hello.bin`
6. Connect board through micro-USB cable
7. On the host side (your laptop), start serial-port listener
8. Flash bin file to the board over mini-USB:

```
openocd -f $(OPENOCD_CFG) \
  -c "init" \
  -c "reset init" \
  -c "flash write_image erase hello.bin 0x08000000" \
  -c "reset run" \
  -c "shutdown"
```

Fixing all of those issues: the idea

1. Install a cross compiler: `gcc-arm-none-eabi`
2. Install `openocd` to communicate with the board
3. Extend `hello.c` with some setup boilerplate code
   - Initialize CPU and set clock frequency
   - Set up serial port (USART) through USB
4. Replace `printf` with code that sends `"Hello World!"` through serial
5. Compile to ARM **binary** (not ELF) file, say `hello.bin`
6. Connect board through micro-USB cable
7. On the host side (your laptop), start serial-port listener
8. Flash bin file to the board over mini-USB:

```
openocd -f $(OPENOCD_CFG) \
  -c "init" \
  -c "reset init" \
  -c "flash write_image erase hello.bin 0x08000000" \
  -c "reset run" \
  -c "shutdown"
```

9. Push “Reset” button to re-run the program

Getting started

Good news! Most of that work is already done.

https://github.com/dop-amin/2026-cryptoeng-assignments-pub

Getting started

Good news! Most of that work is already done.

`https://github.com/dop-amin/2026-cryptoeng-assignments-pub`

- Includes examples for
  - Unidirectional communication (“Hello World!”): **`playground0-print`**
  - Performance benchmarking: **`playground1-bench`**
  - Calling a function written in Jasmin, compiled to assembly: **`playground2-jasmin`**

Getting started

Good news! Most of that work is already done.

`https://github.com/dop-amin/2026-cryptoeng-assignments-pub`

- Includes examples for
  - Unidirectional communication (“Hello World!”): `playground0-print`
  - Performance benchmarking: `playground1-bench`
  - Calling a function written in Jasmin, compiled to assembly: `playground2-jasmin`
- In each of the subdirectories, simply run `make && make test-board`

Getting started

Good news! Most of that work is already done.

https://github.com/dop-amin/2026-cryptoeng-assignments-pub

- Includes examples for
  - Unidirectional communication (“Hello World!”): `playground0-print`
  - Performance benchmarking: `playground1-bench`
  - Calling a function written in Jasmin, compiled to assembly: `playground2-jasmin`
- In each of the subdirectories, simply run `make && make test-board`
- Requires `openocd` and `gcc-arm-none-eabi`

Getting started

Good news! Most of that work is already done.

`https://github.com/dop-amin/2026-cryptoeng-assignments-pub`

- Includes examples for
  - Unidirectional communication (“Hello World!”): `playground0-print`
  - Performance benchmarking: `playground1-bench`
  - Calling a function written in Jasmin, compiled to assembly: `playground2-jasmin`
- In each of the subdirectories, simply run `make && make test-board`
- Requires `openocd` and `gcc-arm-none-eabi`
- All pre-installed in the virtual machine at `https://tinyurl.com/mru2026`

(redirect to `https://nce.mpi-sp.org/index.php/s/y4ddz4cwYgxjKD7`)

Before we optimize: how do we benchmark?

- Want to optimize “down to the last CPU cycle”
- Need high-resolution, cycle-accurate measurements
- All modern CPUs include cycle counters

Before we optimize: how do we benchmark?

- Want to optimize “down to the last CPU cycle”
- Need high-resolution, cycle-accurate measurements
- All modern CPUs include cycle counters
- Getting reliable measurements is hard *in general*

Before we optimize: how do we benchmark?

- Want to optimize “down to the last CPU cycle”
- Need high-resolution, cycle-accurate measurements
- All modern CPUs include cycle counters
- Getting reliable measurements is hard *in general*
- Fairly easy on embedded microcontrollers:
  - Fixed clock frequency (no scaling)
  - No interference from other processes or OS

Before we optimize: how do we benchmark?

- Want to optimize “down to the last CPU cycle”
- Need high-resolution, cycle-accurate measurements
- All modern CPUs include cycle counters
- Getting reliable measurements is hard *in general*
- Fairly easy on embedded microcontrollers:
  - Fixed clock frequency (no scaling)
  - No interference from other processes or OS
- See example in `playground1-bench/src/main.c`
- Caveats:
  - At $>$20 MHz wait cycles introduced by memory controller
  - Cycle counter overflows after $\approx$3 min (20 MHz)

Exercises in this course

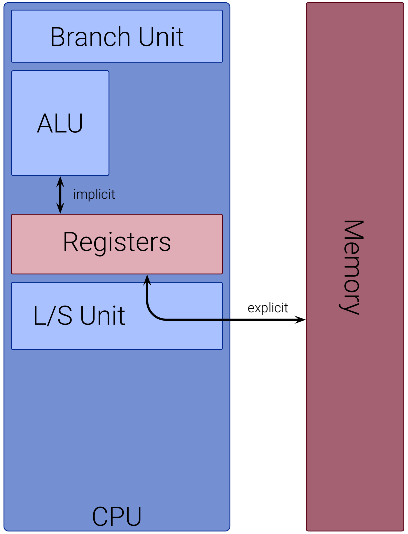
- Extensive build, testing, and benchmarking framework is given
- Tasks in the assignments:
  - Implement (or modify) functions in Jasmin (more this afternoon)
  - Compile from Jasmin to assembly (by invoking `make`)
  - Run given tests to check functionality
  - Run given benchmarks to measure performance

Exercises in this course

- Extensive build, testing, and benchmarking framework is given
- Tasks in the assignments:
  - Implement (or modify) functions in Jasmin (more this afternoon)
  - Compile from Jasmin to assembly (by invoking **`make`**)
  - Run given tests to check functionality
  - Run given benchmarks to measure performance
- **You will not have to write assembly**
- Jasmin is *“assembly in your head”*
- We need to conceptually understand programs on assembly level
- It is useful to be able to read (a bit of) assembly

Computers and computer programs

A highly simplified view



- A program is a sequence of *instructions*
- Load/Store instructions move data between memory and registers (processed by the L/S unit)
- Branch instructions (conditionally) jump to a position in the program
- Arithmetic instructions perform simple operations on values in registers (processed by the ALU)
- Registers are fast (fixed-size) storage units, addressed “by name”

ARMv7-M Registers

- 16 registers: `r0–r15`

ARMv7-M Registers

- 16 registers: `r0`–`r15`
- Some special registers
  - `r13`: `sp` (stack pointer)
  - `r14`: `lr` (link register)
  - `r15`: `pc` (program counter)

ARMv7-M Registers

- 16 registers: `r0`–`r15`
- Some special registers
  - `r13`: `sp` (stack pointer)
  - `r14`: `lr` (link register)
  - `r15`: `pc` (program counter)
- `r13` and `r15` should be used only for their purpose
- There are really only 14 registers freely available
- Jasmin will take care of the correct use of registers

Instructions

- Format: `Instr Rd, Rn(, Rm)`

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory (see later) or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory (see later) or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory (see later) or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - Many instructions have a variant that sets flags by appending `s`
  - Flags record carry, negative, zero, and overflow

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory (see later) or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - Many instructions have a variant that sets flags by appending `s`
  - Flags record carry, negative, zero, and overflow
- Bitwise operations: `eor`, `and`, `orr`, `mvn`

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory (see later) or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - Many instructions have a variant that sets flags by appending `s`
  - Flags record carry, negative, zero, and overflow
- Bitwise operations: `eor`, `and`, `orr`, `mvn`
- Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`

Instructions

- Format: `Instr Rd, Rn(, Rm)`
- `mov r0, r1` (equivalent to `uint32_t r0 = r1;`)
- `mov r0, #18`
  - Sometimes, a constant is too large to fit in an instruction
  - Put constant in memory (see later) or construct it
  - `movw` for bottom 16 bits, `movt` for top 16 bits
- `add`, but also `adds`, `adc`, and `adcs`
  - Many instructions have a variant that sets flags by appending `s`
  - Flags record carry, negative, zero, and overflow
- Bitwise operations: `eor`, `and`, `orr`, `mvn`
- Shifts/rotates: `ror`, `lsl`, `lsr`, `asr`
- All have variants with registers as operands and with a constant (‘immediate’)

Branching and labels

- After every 32-bit instruction, `pc += 4`
- By writing to the `pc`, we can jump to arbitrary locations (and continue execution from there)

Branching and labels

- After every 32-bit instruction, `pc += 4`
- By writing to the `pc`, we can jump to arbitrary locations (and continue execution from there)
- While programming, addresses of instructions are not known

Branching and labels

- After every 32-bit instruction, `pc += 4`
- By writing to the `pc`, we can jump to arbitrary locations (and continue execution from there)
- While programming, addresses of instructions are not known
- Solution: define a *label* and use `b` to branch to labels

Branching and labels

- After every 32-bit instruction, `pc += 4`
- By writing to the `pc`, we can jump to arbitrary locations (and continue execution from there)
- While programming, addresses of instructions are not known
- Solution: define a *label* and use `b` to branch to labels
- Assembler and linker later resolve the address

Branching and labels

- After every 32-bit instruction, `pc += 4`
- By writing to the `pc`, we can jump to arbitrary locations (and continue execution from there)
- While programming, addresses of instructions are not known
- Solution: define a *label* and use `b` to branch to labels
- Assembler and linker later resolve the address

```
mov r0, #42
b somelabel
mov r0, #37
somelabel:
...
```

Conditional branches

- How to do a `while` loop?

Conditional branches

- How to do a `while` loop?
- Need to do a *test* and branch depending on the outcome

Conditional branches

- How to do a `while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`

Conditional branches

- How to do a `while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`
- Really: subtract, set status flags, discard result

Conditional branches

- How to do a `while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`
- Really: subtract, set status flags, discard result
- Instead of `b`, use a conditional branch
  - `beq` *label* (`r0 == r1`)
  - `bne` *label* (`r0 != r1`)

Conditional branches

- How to do a `while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`
- Really: subtract, set status flags, discard result
- Instead of `b`, use a conditional branch

`beq` *label*	(`r0 == r1`)
`bne` *label*	(`r0 != r1`)
`bhi` *label*	(`r0 > r1`, unsigned)
`bls` *label*	(`r0 <= r1`, unsigned)
`bgt` *label*	(`r0 > r1`, signed)
`bge` *label*	(`r0 >= r1`, signed)
`bmi` *label*	(result is negative)

Conditional branches

- How to do a `while` loop?
- Need to do a *test* and branch depending on the outcome
  - `cmp r0, r1` (`r1` can also be shifted/rotated!)
  - `cmp r0, #5`
- Really: subtract, set status flags, discard result
- Instead of `b`, use a conditional branch
  - `beq` *label* (`r0 == r1`)
  - `bne` *label* (`r0 != r1`)
  - `bhi` *label* (`r0 > r1`, unsigned)
  - `bls` *label* (`r0 <= r1`, unsigned)
  - `bgt` *label* (`r0 > r1`, signed)
  - `bge` *label* (`r0 >= r1`, signed)
  - `bmi` *label* (result is negative)
  - And many more

Conditional branches (example)

- In C:

```
uint32_t a, b = 100;

for (a = 0; a <= 50; a++) {
  b += a;
}
```

- In assembly:

```
mov r0, #0     // a
mov r1, #100   // b

loop:
add r1, r0    // b += a

add r0, #1    // a++
cmp r0, #50   // compare a and 50
bls loop      // loop if <=
```

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;
}
```

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 general lessons to learn

- *Some* loop unrolling helps:
  - Less loop-control overhead per computation
  - Can merge operations across iterations

Some general lessons to learn

- *Some* loop unrolling helps:
  - Less loop-control overhead per computation
  - Can merge operations across iterations
- Full unrolling can be problematic:
  - Code size can increase massively
  - Loop length may be known only at runtime

Some general lessons to learn

- *Some* loop unrolling helps:
  - Less loop-control overhead per computation
  - Can merge operations across iterations
- Full unrolling can be problematic:
  - Code size can increase massively
  - Loop length may be known only at runtime
- Make best use of architectural features:
  - Merge `cmp` and `sub`
  - Merge counter increase into `ldr`

Some general lessons to learn

- *Some* loop unrolling helps:
  - Less loop-control overhead per computation
  - Can merge operations across iterations
- Full unrolling can be problematic:
  - Code size can increase massively
  - Loop length may be known only at runtime
- Make best use of architectural features:
  - Merge **`cmp`** and **`sub`**
  - Merge counter increase into **`ldr`**
- Pay attention to microarchitecture:
  - Loads and stores are faster when grouped

Some general lessons to learn

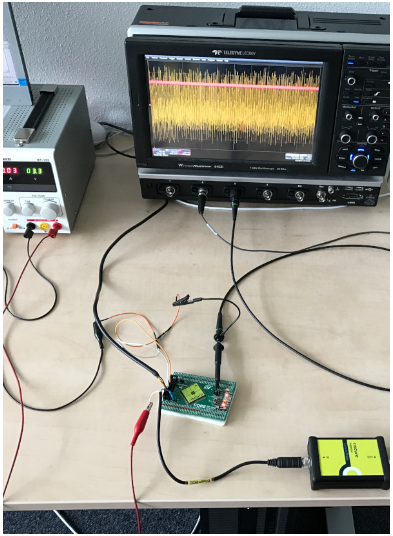
- *Some* loop unrolling helps:
  - Less loop-control overhead per computation
  - Can merge operations across iterations
- Full unrolling can be problematic:
  - Code size can increase massively
  - Loop length may be known only at runtime
- Make best use of architectural features:
  - Merge **`cmp`** and **`sub`**
  - Merge counter increase into **`ldr`**
- Pay attention to microarchitecture:
  - Loads and stores are faster when grouped
- Optimized code may require more registers

Optimizing “something” vs. optimizing crypto

- So far there was nothing crypto-specific in this lecture
- We did not yet talk about “leaking secrets”
- Need to think about our attacker!

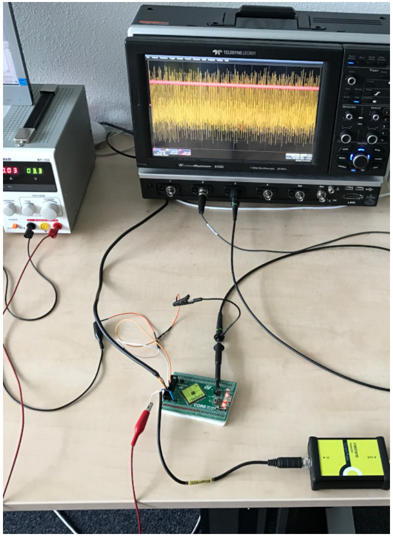
Implementation Security

Implementation Security



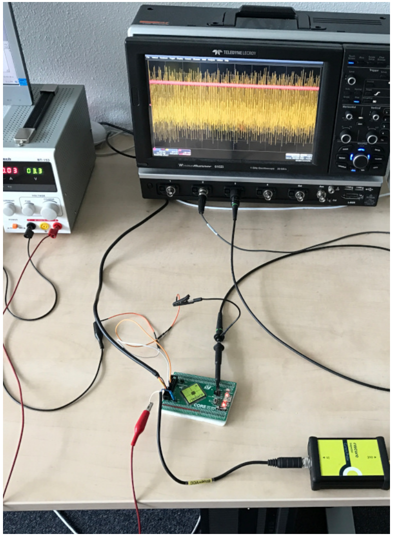
- Attackers see more than input/output:
  - Power consumption
  - Electromagnetic radiation
  - Timing

Implementation Security



- Attackers see more than input/output:
  - Power consumption
  - Electromagnetic radiation
  - Timing
- Side-channel attacks:
  - Measure information
  - Use to obtain secret data

Implementation Security



- Attackers see more than input/output:
  - Power consumption
  - Electromagnetic radiation
  - Timing
- **Side-channel attacks:**
  - Measure information
  - Use to obtain secret data
- **Timing attacks** can be done **remotely**

Timing attacks and “constant-time” programming

Timing attacks

- Attacker obtains fine-granular timing information
- **Not** just overall execution time!
- Essentially time of each individual instruction

Timing attacks and “constant-time” programming

Timing attacks

- Attacker obtains fine-granular timing information
- **Not** just overall execution time!
- Essentially time of each individual instruction
- Timing attacks are practical:
  Osvik, Tromer, Shamir, 2006: 65 ms to steal a 256-bit AES key used for Linux hard-disk encryption

Timing attacks and “constant-time” programming

Timing attacks

- Attacker obtains fine-granular timing information
- **Not** just overall execution time!
- Essentially time of each individual instruction
- 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

Timing attacks and “constant-time” programming

Timing attacks

- Attacker obtains fine-granular timing information
- **Not** just overall execution time!
- Essentially time of each individual instruction
- 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

“Constant-time” programming

- Misnomer: timing is only indendent of secret data
- Idea: No data flow from secrets into variable-time operations

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
    $r \leftarrow A$
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

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
    $r \leftarrow A$
else
    $r \leftarrow B$
end if
```

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

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

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

Eliminating branches

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

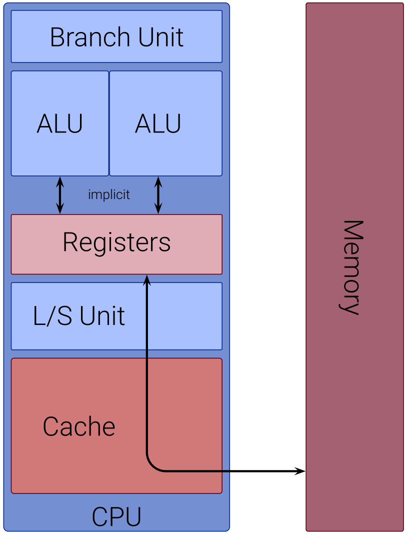
```
if $s$ then
    $r \leftarrow A$
else
    $r \leftarrow B$
end if
```

- Replace by

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

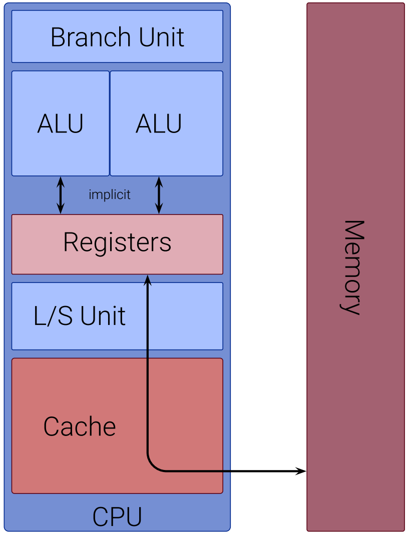
- Can expand s to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication
- For very fast A and B this can even be faster

Cached memory access



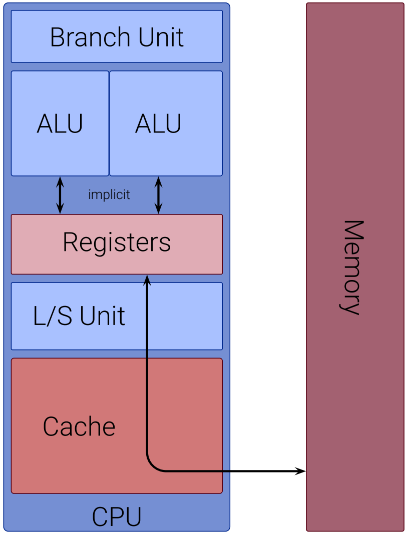
- On most CPUs, memory access goes through a **cache**
- Small but fast transparent memory for frequently used data

Cached memory access



- On most CPUs, memory access goes through a **cache**
- Small but fast transparent memory for frequently used data
- A load from memory places data also in the cache
- Data remains in cache until it's replaced by other data

Cached memory access



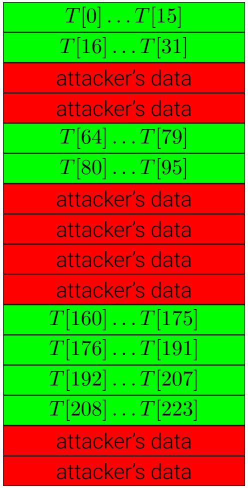
- On most CPUs, memory access goes through a **cache**
- Small but fast transparent memory for frequently used data
- A load from memory places data also in the cache
- Data remains in cache until it's replaced by other data
- Loading data is fast if data is in the cache (**cache hit**)
- Loading data is slow if data is not in the cache (**cache miss**)

Timing leakage part II

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

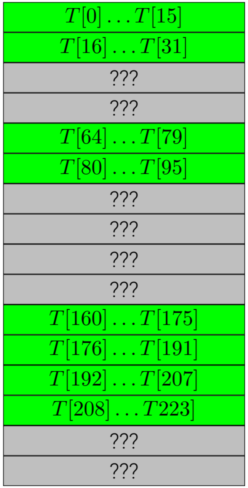
- Consider lookup table of 32-bit integers
- Assume that *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache

Timing leakage part II



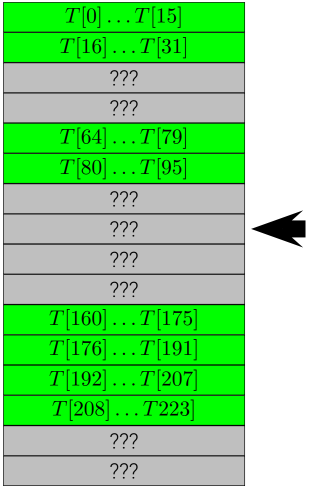
- Consider lookup table of 32-bit integers
- Assume that *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines

Timing leakage part II



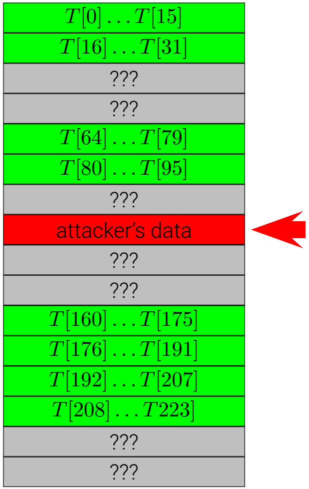
- Consider lookup table of 32-bit integers
- Assume that *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again

Timing leakage part II



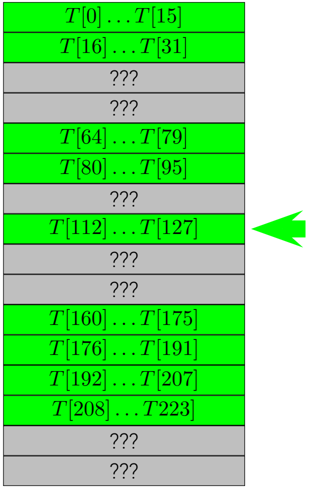
- Consider lookup table of 32-bit integers
- Assume that *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads his data:

Timing leakage part II



- Consider lookup table of 32-bit integers
- Assume that *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 his data:
  - Fast: cache hit (crypto did not just load from this line)

Timing leakage part II



- Consider lookup table of 32-bit integers
- Assume that *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 his data:
  - Fast: cache hit (crypto did not just load from this line)
  - Slow: cache miss (crypto just loaded from this line)

Should we care? Does the Cortex-M4 have 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

Should we care? Does the Cortex-M4 have 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

Should we care? Does the Cortex-M4 have caches?

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

—ARM Cortex-M7 TRM

Some comments on timing attacks

- What I just showed is only the *most basic* cache-timing attack

Some comments on timing attacks

- What I just showed is only the *most basic* cache-timing attack
- Non-secret cache lines are not enough for security
- Generally, load/store addresses influence timing in many different ways
- **Do not access memory at secret-data-dependent addresses**

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
    $d \leftarrow T[i]$
    if $p = i$ then
        $r \leftarrow d$
    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
    $d \leftarrow T[i]$
    if $p = i$ then
        $r \leftarrow d$
    end if
end for
```

- Problem 1: if-statements are not constant time (see before)
- Problem 2: Need to be careful with comparisons (at least in high-level languages)

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
- Jasmin helps with this! (more this afternoon)

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
- Jasmin helps with this! (more this afternoon)

- **Good news:** On Cortex M4, that is pretty much it

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
- Jasmin helps with this! (more this afternoon)

- **Good news:** On Cortex M4, that is pretty much it
- **Bad news:** On other microarchitectures, there may also be *variable-time arithmetic*, e.g.,
  - `DIV`, `IDIV`, `FDIV` on pretty much all Intel/AMD CPUs
  - `UMULL`, `SMULL`, `UMLAL`, and `SMLAL` on ARM Cortex-M3

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
- Jasmin helps with this! (more this afternoon)

- **Good news:** On Cortex M4, that is pretty much it
- **Bad news:** On other microarchitectures, there may also be *variable-time arithmetic*, e.g.,
  - `DIV`, `IDIV`, `FDIV` on pretty much all Intel/AMD CPUs
  - `UMULL`, `SMULL`, `UMLAL`, and `SMLAL` on ARM Cortex-M3
- **More good news:** Jasmin also helps here!