Optimizing crypto on embedded microcontrollers

Peter Schwabe

Radboud University, Nijmegen, The Netherlands

August 30, 2017

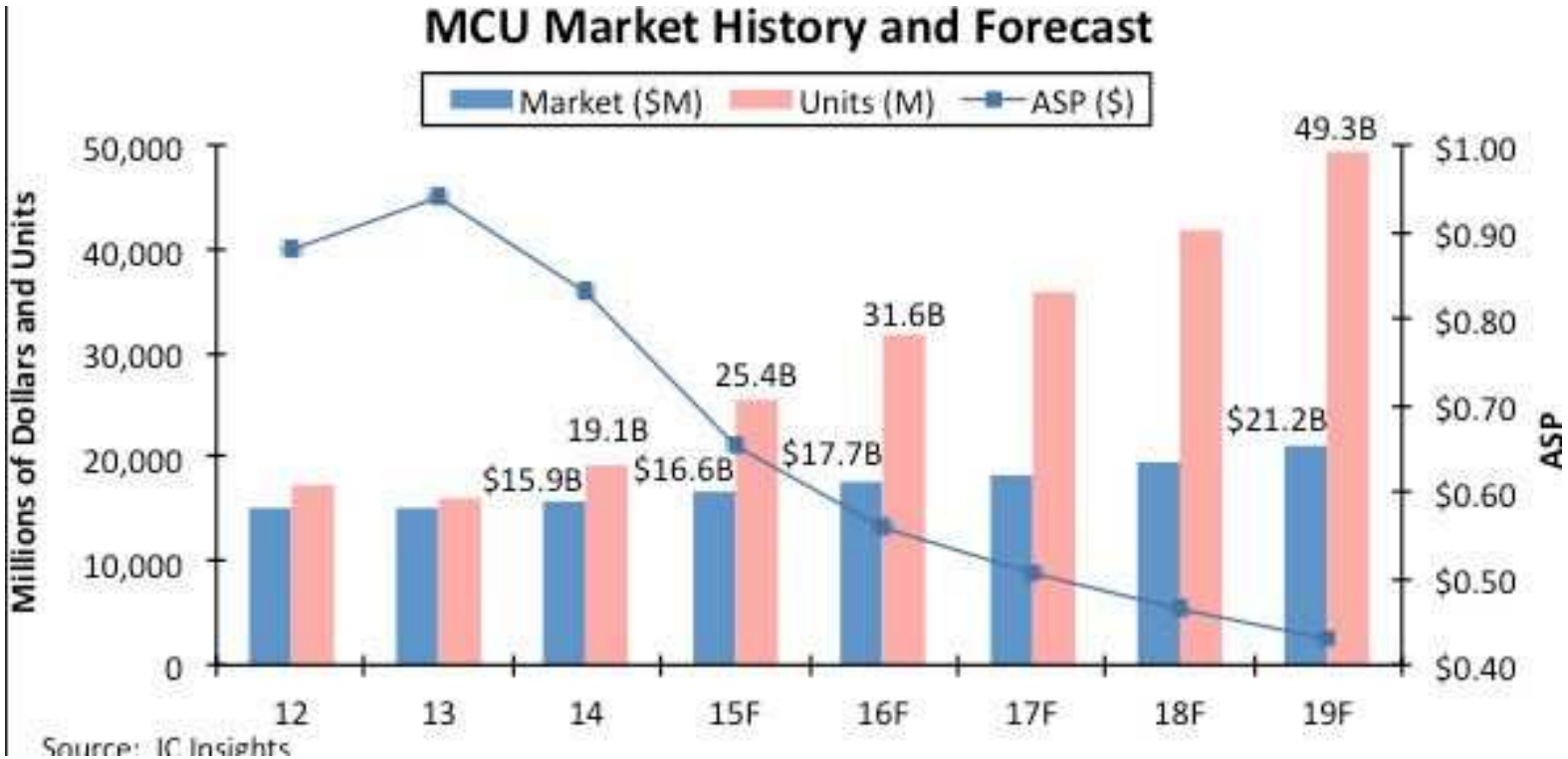
COINS Summerschool 2017

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



Cryptography

Symmetric crypto

- Block ciphers (AES, DES, Present, . . .)
- Stream ciphers (Salsa20, ChaCha20, . . .)
- Hash functions (SHA2, SHA3, . . .)
- Authenticators (HMAC, GHASH, Poly1305, . . .)

Cryptography

Symmetric crypto

- Block ciphers (AES, DES, Present, ...)
- Stream ciphers (Salsa20, ChaCha20, ...)
- Hash functions (SHA2, SHA3, ...)
- Authenticators (HMAC, GHASH, Poly1305, ...)

Asymmetric crypto

- RSA, DSA, Diffie-Hellman, ElGamal
- ECDH, ECDSA, EdDSA
- Post-quantum crypto:
  - Lattice-based crypto
  - Code-based crypto
  - Hash-based signatures
  - Multivariate crypto
  - Supersingular-isogeny-based crypto

Cryptography

Symmetric crypto

- Block ciphers (AES, DES, Present, . . .)
- Stream ciphers (Salsa20, ChaCha20, . . .)
- Hash functions (SHA2, SHA3, . . .)
- Authenticators (HMAC, GHASH, Poly1305, . . .)

Asymmetric crypto

- RSA, DSA, Diffie-Hellman, ElGamal
- ECDH, ECDSA, EdDSA
- Post-quantum crypto:
  - Lattice-based crypto
  - Code-based crypto
  - Hash-based signatures
  - Multivariate crypto
  - Supersingular-isogeny-based crypto

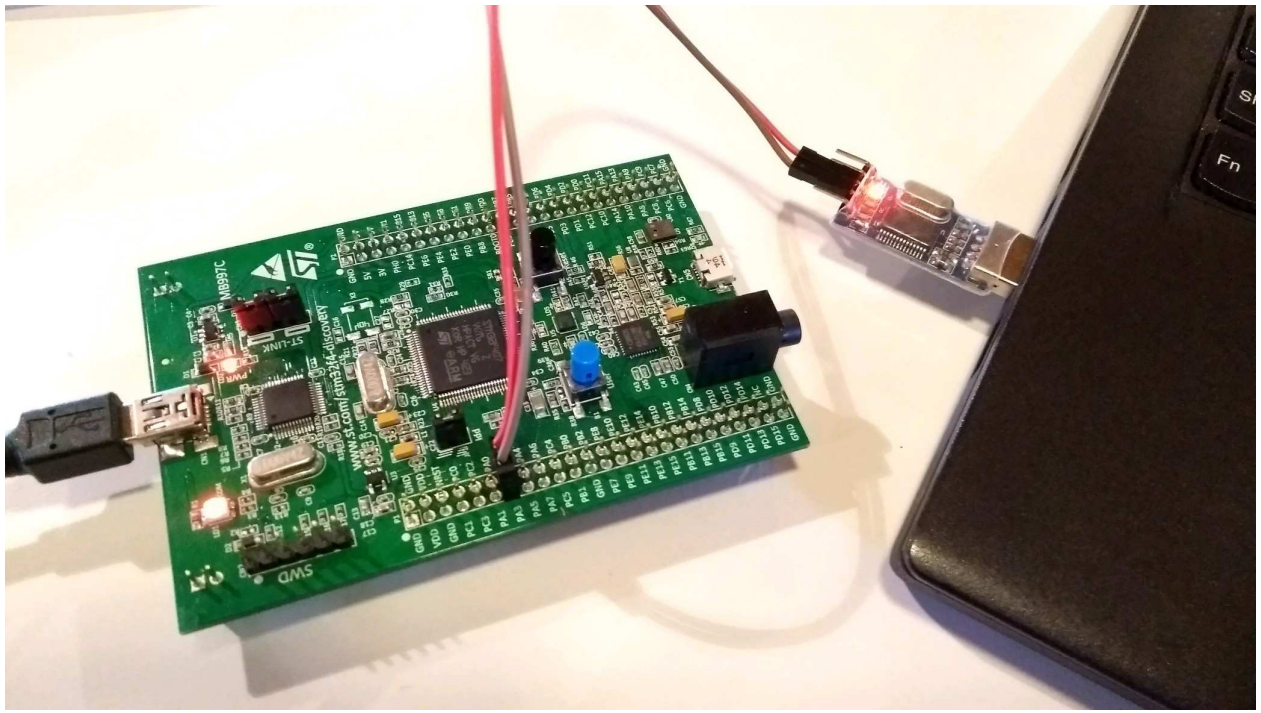
Optimizing

- 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

Our Target platform



- ARM Cortex-M4 on STM32F4-Discovery board
- 192KB RAM, 1MB Flash (ROM)
- Available for <20 Euros from various vendors (e.g., Amazon, RS Components, Conrad)
- Additionally need USB-TTL converter and mini-USB cable

Cortex-M4 basics

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

Cortex-M4 basics

- 16 registers, `r0` to `r15`
- 32 bits wide
- Not all can be used freely
  - `r13` is `sp`, stack pointer
  - `r14` is `lr`, link register
  - `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 basics

- 16 registers, `r0` to `r15`
- 32 bits wide
- Not all can be used freely
  - `r13` is `sp`, stack pointer
  - `r14` is `lr`, link register
  - `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 first 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 at least 2 cycles

How fast is it?

- Arithmetic instructions cost 1 cycle
- (Single) loads cost 2 cycles
- Branches cost at least 2 cycles
- The loop body should cost 10 cycles

Benchmarking

- Read from `DWT_CYCCNT`
- Execute function
- Read from `DWT_CYCCNT`, compare

How fast is it?

- Arithmetic instructions cost 1 cycle
- (Single) loads cost 2 cycles
- Branches cost at least 2 cycles
- The loop body should cost 10 cycles

Benchmarking

- Read from `DWT_CYCCNT`
- Execute function
- Read from `DWT_CYCCNT`, compare
- Needs some setup; see example code (later)

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 lecture
- Is optimizing crypto the same as optimizing any other software?

Optimizing "something" vs. optimizing crypto

- So far there was nothing crypto-specific in this lecture
- 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 lecture
- 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 lecture
- 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:
  - Only side-channel that 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 lecture
- 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:
  - Only side-channel that 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**
  $r \leftarrow A$
  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?

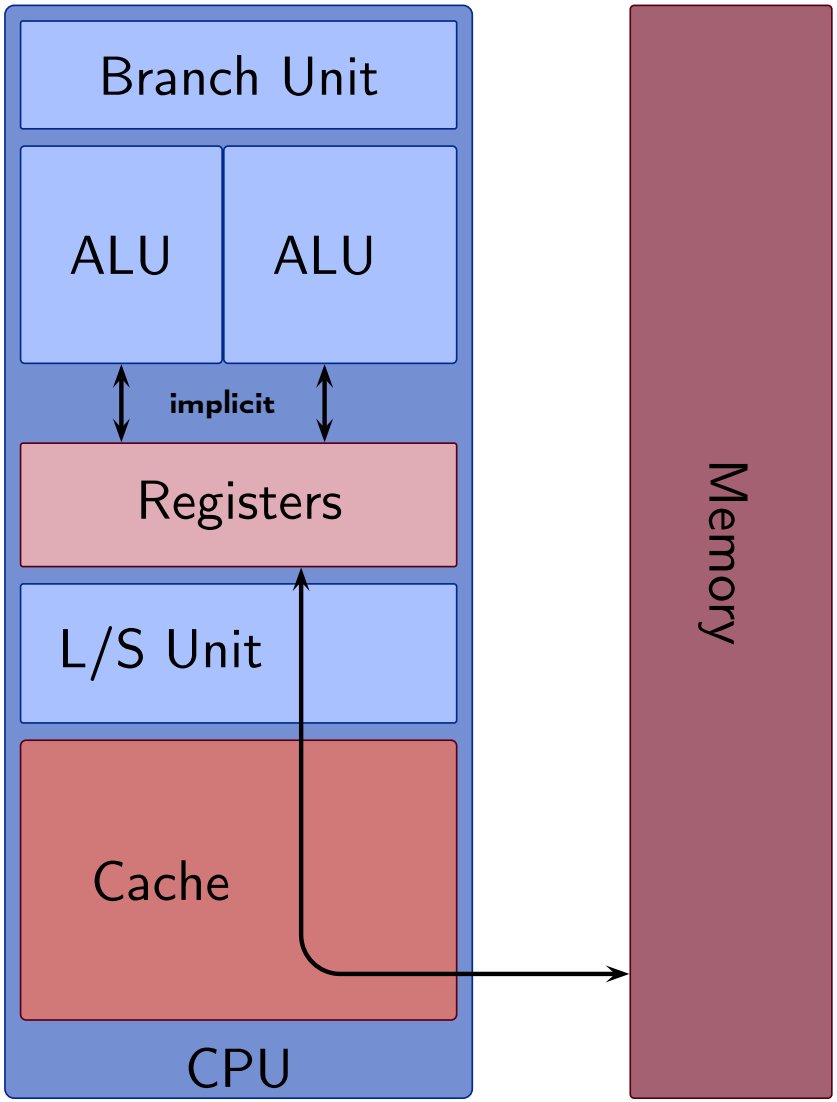
  if s **then**
  $r \leftarrow A$
  else
  $r \leftarrow B$
  end if

- Replace by

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

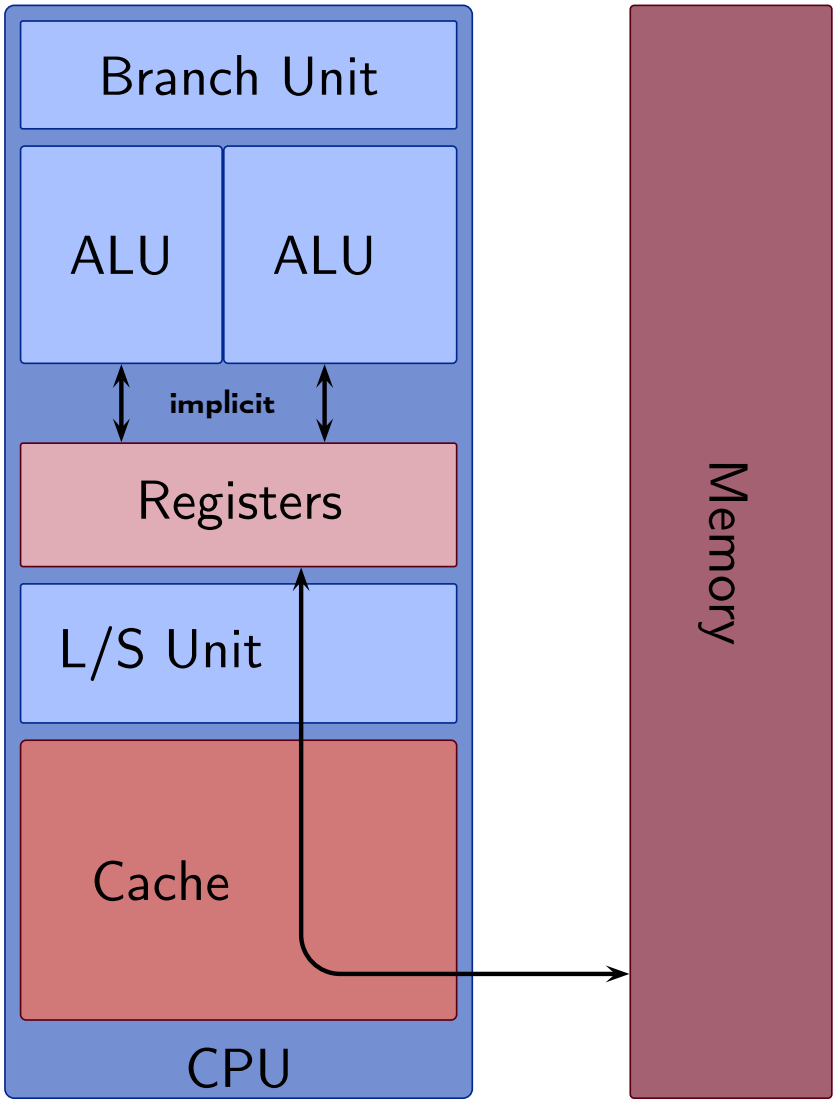
- Can expand s to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication
- For very fast A and B this can even be faster

Cached memory access



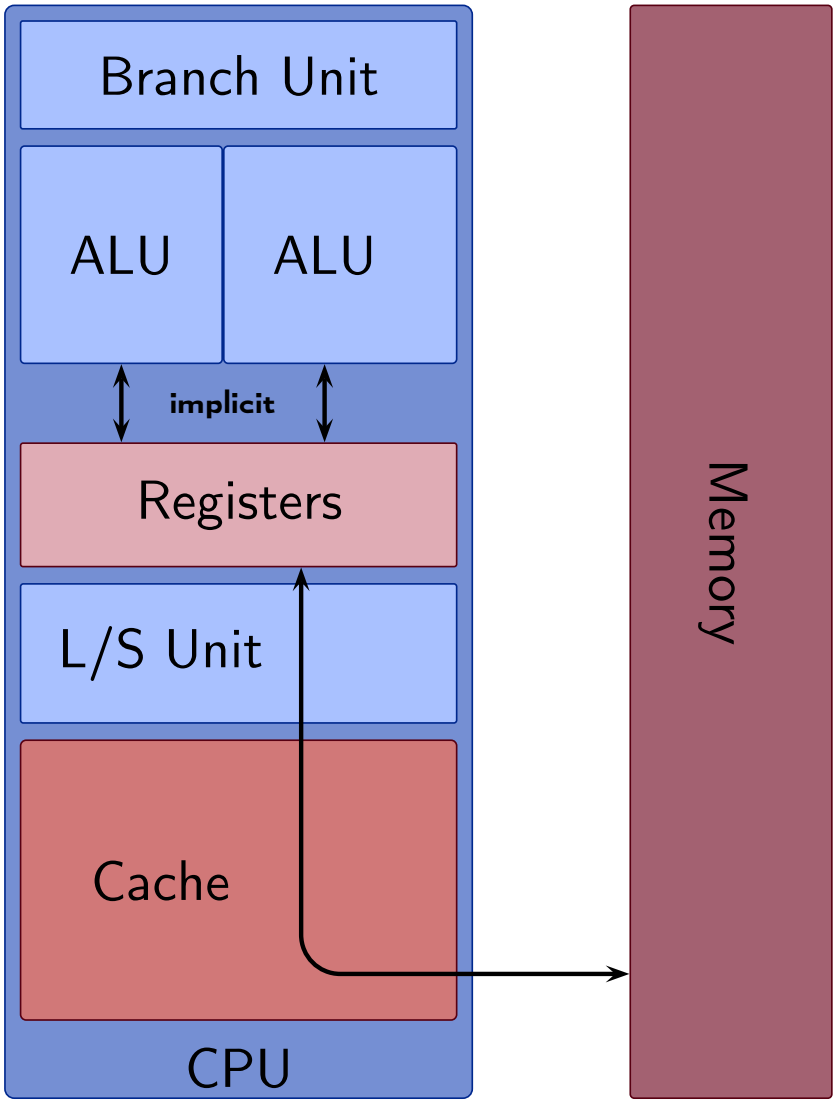
- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data

Cached memory access



- 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



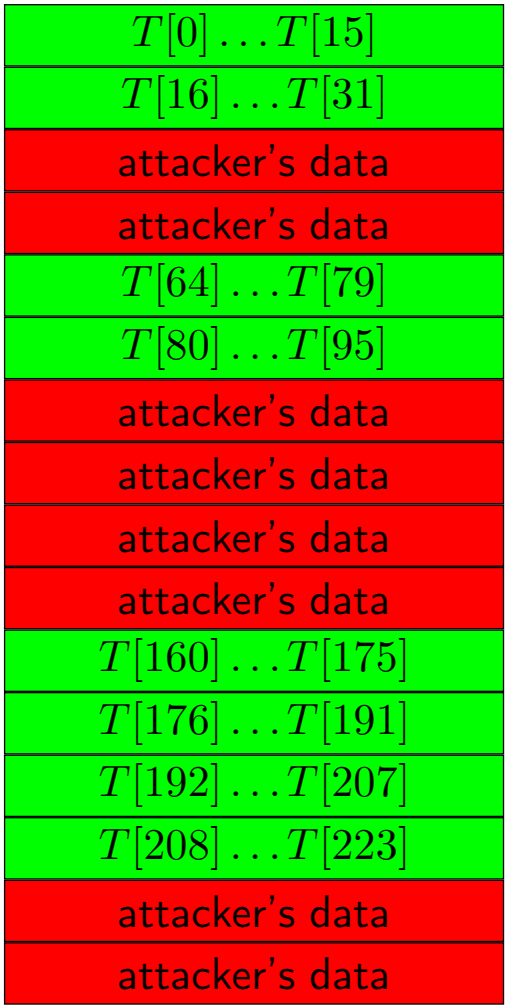
- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data
- A load from memory places data also in the cache
- Data remains in cache until it's replaced by other data
- Loading data is fast if data is in the cache (**cache hit**)
- Loading data is slow if data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

Timing leakage part II

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

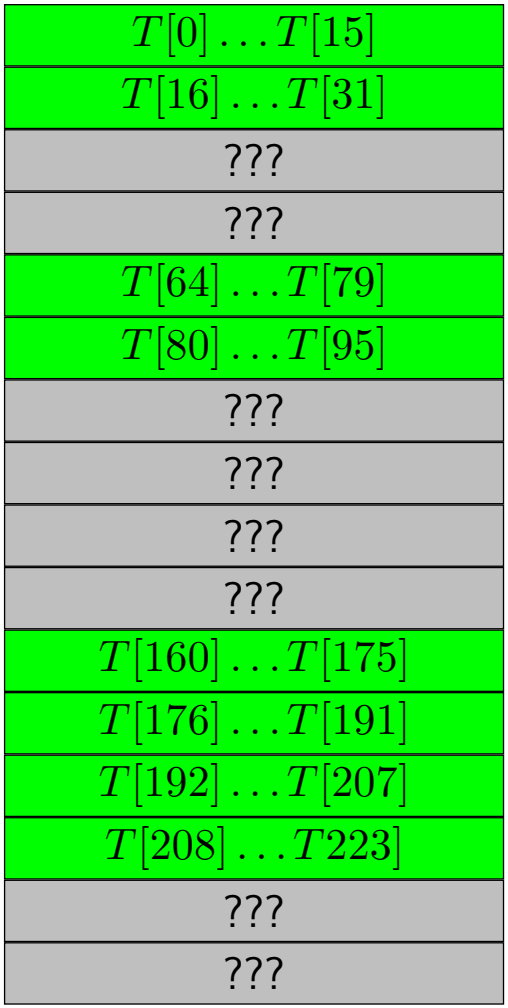
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache

Timing leakage part II



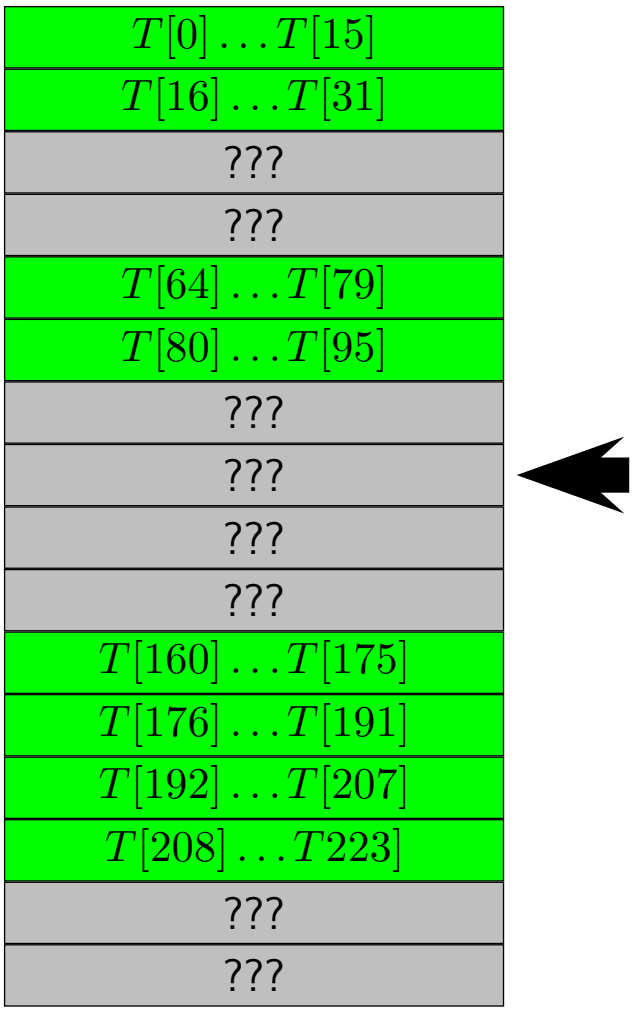
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines

Timing leakage part II



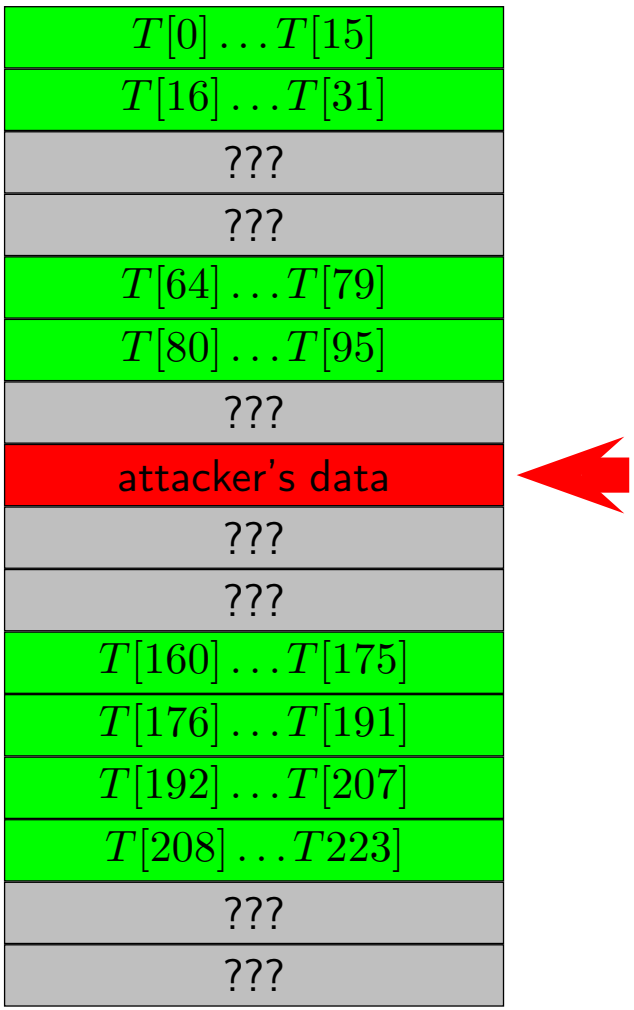
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again

Timing leakage part II



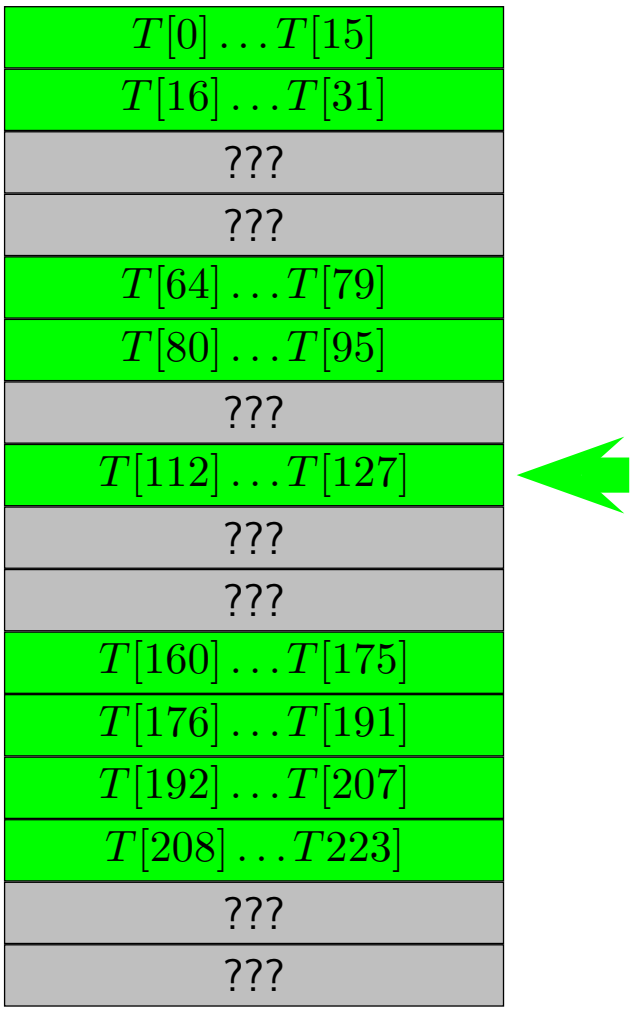
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads his data:

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 his 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 his 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
- Load/Store addresses influence timing in many different ways
- **Do not access memory at secret-data-dependent addresses**

Some comments on cache-timing

- This is only the *most basic* cache-timing attack
- Non-secret cache lines are not enough for security
- Load/Store addresses influence timing in many different ways
- **Do not access memory at secret-data-dependent addresses**
- 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
- Load/Store addresses influence timing in many different ways
- **Do not access memory at secret-data-dependent addresses**
- 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**
    $d \leftarrow T[i]$
    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: Comparisons are not constant 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;
}
```

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 `valgrind` and *uninitialized secret data* (or use Langley's `ctgrind`)

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 `valgrind` and *uninitialized secret data* (or use Langley's `ctgrind`)

"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 `valgrind` and *uninitialized secret data* (or use Langley's `ctgrind`)

"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

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;
```

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`

CHES 2017, Sep. 25–28

Conference Advertisement

Apply for student-stipends until Sep. 7!

Poly1305

- Designed by Bernstein in 2005
- Secret-key one-time authenticator based on arithmetic in $\mathbb{F}_p$ with $p = 2^{130} - 5$
- Key k and (padded) 16-byte ciphertext blocks $c_1, \ldots, c_k$ are in $\mathbb{F}_p$

Poly1305

- Designed by Bernstein in 2005
- Secret-key one-time authenticator based on arithmetic in $\mathbb{F}_p$ with $p = 2^{130} - 5$
- Key k and (padded) 16-byte ciphertext blocks $c_1, \ldots, c_k$ are in $\mathbb{F}_p$
- Main work: initialize authentication tag h with 0, then compute:

  for i from 1 to k **do**
  $h \leftarrow h + c_i$
  $h \leftarrow h \cdot k$
  end for

Poly1305

- Designed by Bernstein in 2005
- Secret-key one-time authenticator based on arithmetic in $\mathbb{F}_p$ with $p = 2^{130} - 5$
- Key k and (padded) 16-byte ciphertext blocks $c_1, \ldots, c_k$ are in $\mathbb{F}_p$
- Main work: initialize authentication tag h with 0, then compute:

  for i from 1 to k **do**
  $h \leftarrow h + c_i$
  $h \leftarrow h \cdot k$
  end for

- Per 16 bytes: 1 multiplication, 1 addition in $\mathbb{F}_{2^{130}-5}$
- Some (fast) finalization to produce 16-byte authentication tag

Poly1305

- Designed by Bernstein in 2005
- Secret-key one-time authenticator based on arithmetic in $\mathbb{F}_p$ with $p = 2^{130} - 5$
- Key k and (padded) 16-byte ciphertext blocks $c_1, \ldots, c_k$ are in $\mathbb{F}_p$
- Main work: initialize authentication tag h with 0, then compute:

  for i from 1 to k **do**
  $h \leftarrow h + c_i$
  $h \leftarrow h \cdot k$
  end for

- Per 16 bytes: 1 multiplication, 1 addition in $\mathbb{F}_{2^{130}-5}$
- Some (fast) finalization to produce 16-byte authentication tag

Multiprecision arithmetic in crypto

- Asymmetric cryptography heavily relies on arithmetic on "big integers"
- Example 1: RSA-2048 needs (modular) multiplication and squaring of 2048-bit numbers

Multiprecision arithmetic in crypto

- Asymmetric cryptography heavily relies on arithmetic on "big integers"
- Example 1: RSA-2048 needs (modular) multiplication and squaring of 2048-bit numbers
- Example 2:
  - Elliptic curves defined over finite fields
  - Typically use EC over large-characteristic prime fields
  - Typical field sizes: (160 bits, 192 bits), 256 bits, 448 bits ...

Multiprecision arithmetic in crypto

- Asymmetric cryptography heavily relies on arithmetic on "big integers"
- Example 1: RSA-2048 needs (modular) multiplication and squaring of 2048-bit numbers
- Example 2:
  - Elliptic curves defined over finite fields
  - Typically use EC over large-characteristic prime fields
  - Typical field sizes: (160 bits, 192 bits), 256 bits, 448 bits ...
- An integer is "big" if it's not natively supported by the machine architecture
- Example: AMD64 supports up to 64-bit integers, multiplication produces 128-bit result, but not bigger than that.
- We call arithmetic on such "big integers" *multiprecision arithmetic*

Multiprecision arithmetic in crypto

- Asymmetric cryptography heavily relies on arithmetic on "big integers"
- Example 1: RSA-2048 needs (modular) multiplication and squaring of 2048-bit numbers
- Example 2:
  - Elliptic curves defined over finite fields
  - Typically use EC over large-characteristic prime fields
  - Typical field sizes: (160 bits, 192 bits), 256 bits, 448 bits ...
- An integer is "big" if it's not natively supported by the machine architecture
- Example: AMD64 supports up to 64-bit integers, multiplication produces 128-bit result, but not bigger than that.
- We call arithmetic on such "big integers" *multiprecision arithmetic*
- Example architecture for multiprecison arithmetic: **AVR ATmega**

The first year of primary school

Available numbers (digits): $(0), 1, 2, 3, 4, 5, 6, 7, 8, 9$

The first year of primary school

Available numbers (digits): $(0), 1, 2, 3, 4, 5, 6, 7, 8, 9$

Addition

$3 + 5 = \quad ?$

$2 + 7 = \quad ?$

$4 + 3 = \quad ?$

The first year of primary school

Available numbers (digits): $(0), 1, 2, 3, 4, 5, 6, 7, 8, 9$

Addition

$3 + 5 = \quad ?$

$2 + 7 = \quad ?$

$4 + 3 = \quad ?$

Subtraction

$7 - 5 = \quad ?$

$5 - 1 = \quad ?$

$9 - 3 = \quad ?$

The first year of primary school

Available numbers (digits): $(0), 1, 2, 3, 4, 5, 6, 7, 8, 9$

Addition

$3 + 5 = \quad ?$

$2 + 7 = \quad ?$

$4 + 3 = \quad ?$

Subtraction

$7 - 5 = \quad ?$

$5 - 1 = \quad ?$

$9 - 3 = \quad ?$

- All results are in the set of available numbers
- No confusion for first-year school kids

Programming today

Available numbers: $0, 1, \ldots, 255$

Programming today

Available numbers: $0, 1, \ldots, 255$

Addition

```
uint8_t a = 42;
uint8_t b = 89;
uint8_t r = a + b;
```

Programming today

Available numbers: $0, 1, \ldots, 255$

Addition

```
uint8_t a = 42;
uint8_t b = 89;
uint8_t r = a + b;
```

Subtraction

```
uint8_t a = 157;
uint8_t b = 23;
uint8_t r = a - b;
```

Programming today

Available numbers: $0, 1, \ldots, 255$

Addition

```
uint8_t a = 42;
uint8_t b = 89;
uint8_t r = a + b;
```

Subtraction

```
uint8_t a = 157;
uint8_t b = 23;
uint8_t r = a - b;
```

- All results are in the set of available numbers
- Larger set of available numbers: `uint16_t`, `uint32_t`, `uint64_t`
- Basic principle is the same; for the moment stick with `uint8_t`

Still in the first year of primary school

Crossing the ten barrier

$$6 + 5 = \quad ?$$
$$9 + 7 = \quad ?$$
$$4 + 8 = \quad ?$$

Still in the first year of primary school

Crossing the ten barrier

$$6 + 5 = \quad ?$$
$$9 + 7 = \quad ?$$
$$4 + 8 = \quad ?$$

- Inputs to addition are still from the set of available numbers
- Results are allowed to be larger than 9

Still in the first year of primary school

Crossing the ten barrier

$$6 + 5 = \quad ?$$
$$9 + 7 = \quad ?$$
$$4 + 8 = \quad ?$$

- Inputs to addition are still from the set of available numbers
- Results are allowed to be larger than 9
- Addition is allowed to produce a *carry*

Still in the first year of primary school

Crossing the ten barrier

$$6 + 5 = \quad ?$$
$$9 + 7 = \quad ?$$
$$4 + 8 = \quad ?$$

- Inputs to addition are still from the set of available numbers
- Results are allowed to be larger than 9
- Addition is allowed to produce a *carry*

What happens with the carry?

- Introduce the decimal positional system
- Write an integer A in two digits $a_1 a_0$ with

$$A = 10 \cdot a_1 + a_0$$

- Note that at the moment $a_1 \in \{0, 1\}$

. . . back to programming

```
uint8_t a = 184;
uint8_t b = 203;
uint8_t r = a + b;
```

. . . back to programming

```
uint8_t a = 184;
uint8_t b = 203;
uint8_t r = a + b;
```

- The result `r` now has the value of 131
- The carry is lost, what do we do?

. . . back to programming

```
uint8_t a = 184;
uint8_t b = 203;
uint8_t r = a + b;
```

- The result `r` now has the value of 131
- The carry is lost, what do we do?
- Could cast to `uint16_t`, `uint32_t` etc.,
  but that solves the problem only for this `uint8_t` example
- We really want to obtain the carry, and put it into another `uint8_t`

The AVR ATmega

- 8-bit RISC architecture
- 32 registers `R0`...`R31`, some of those are "special":
  - (`R26`,`R27`) aliased as `X`
  - (`R28`,`R29`) aliased as `Y`
  - (`R30`,`R31`) aliased as `Z`
  - `X`, `Y`, `Z` are used for addressing
  - 2-byte output of a multiplication always in `R0`, `R1`
- Most arithmetic instructions cost 1 cycle
- Multiplication and memory access takes 2 cycles

$184 + 203$

```
LDI R5, 184
LDI R6, 203
ADD R5, R6   ; result in R5, sets carry flag
CLR R6       ; set R6 to zero
ADC R6,R6    ; add with carry, R6 now holds the carry
```

Later in primary school

Addition

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

Later in primary school

Addition

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

$$\begin{array}{rr} & 7862 \\ + & 5275 \\ \hline + & 7 \end{array}$$

Later in primary school

Addition

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

$$\begin{array}{rr} & 7862 \\ + & 5275 \\ \hline + & 37 \end{array}$$

Later in primary school

Addition

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

$$\begin{array}{rr} & 7862 \\ + & 5275 \\ \hline + & 137 \end{array}$$

Later in primary school

Addition

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

$$\begin{array}{rr} & 7862 \\ + & 5275 \\ \hline + & 13137 \end{array}$$

Later in primary school

Addition

$42 + 78 = \quad ?$

$789 + 543 = \quad ?$

$7862 + 5275 = \quad ?$

$$\begin{array}{rr} & 7862 \\ + & 5275 \\ \hline + & 13137 \end{array}$$

- Once school kids can add beyond 1000, they can add arbitrary numbers

Multiprecision addition is old

"Oh Līlāvatī, intelligent girl, if you understand addition and subtraction, tell me the sum of the amounts 2, 5, 32, 193, 18, 10, and 100, as well as [the remainder of] those when subtracted from 10000."

—"Līlāvatī" by Bhāskara (1150)

AVR multiprecision addition...

- Add two n-byte numbers, returning an $n+1$ byte result:
- Input pointers X,Y, output pointer Z

```
LD R5,X+
LD R6,Y+
ADD R5,R6
ST Z+,R5

LD R5,X+
LD R6,Y+
ADC R5,R6
ST Z+,R5
```

```
LD R5,X+
LD R6,Y+
ADC R5,R6
ST Z+,R5

LD R5,X+
LD R6,Y+
ADC R5,R6
ST Z+,R5

...
```

```
CLR R5
ADC R5,R5
ST Z+,R5
```

. . . and subtraction

- Subtract two n-byte numbers, returning an $n+1$ byte result:
- Input pointers `X`,`Y`, output pointer `Z`
- Use highest byte $= -1$ to indicate negative result

```
LD R5,X+
LD R6,Y+
SUB R5,R6
ST Z+,R5
```

```
LD R5,X+
LD R6,Y+
SBC R5,R6
ST Z+,R5
```

```
CLR R5
SBC R5,R5
ST Z+,R5
```

```
LD R5,X+
LD R6,Y+
SBC R5,R6
ST Z+,R5
```

```
LD R5,X+
LD R6,Y+
SBC R5,R6
ST Z+,R5

...
```

How about multiplication?

- Consider multiplication of 1234 by 789

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 6 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 06 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 106 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 11106 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$
\begin{array}{r}
1234 \cdot 789 \\
\hline
11106 \\
9872
\end{array}
$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$
\begin{array}{r}
1234 \cdot 789 \\
\hline
11106 \\
9872 \\
8638
\end{array}
$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$
\begin{array}{rr}
 & 1234 \cdot 789 \\
\hline
 & 11106 \\
+ & 9872 \\
+ & 8638 \\
\hline
 & 973626
\end{array}
$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 11106 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$
\begin{array}{rr}
 & 1234 \cdot 789 \\
\hline
 & 11106 \\
+ & 9872
\end{array}
$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 20978 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$
\begin{array}{rr}
 & 1234 \cdot 789 \\
\hline
 & 20978 \\
+ & 8638
\end{array}
$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 973626 \end{array}$$

How about multiplication?

- Consider multiplication of 1234 by 789

$$\begin{array}{r} 1234 \cdot 789 \\ \hline 973626 \end{array}$$

- This is also an old technique
- Earliest reference I could find is again the Līlāvatī (1150)

Let's do that on the AVR

```
LD R2, X+
LD R3, X+
LD R4, X+

LD R7, Y+

MUL R2,R7
ST Z+,R0
MOV R8,R1

MUL R3,R7
ADD R8,R0
CLR R9
ADC R9,R1

MUL R4,R7
ADD R9,R0
CLR R10
ADC R10,R1
```

Let's do that on the AVR

```
LD R2, X+
LD R3, X+
LD R4, X+

LD R7, Y+

MUL R2,R7
ST Z+,R0
MOV R8,R1

MUL R3,R7
ADD R8,R0
CLR R9
ADC R9,R1

MUL R4,R7
ADD R9,R0
CLR R10
ADC R10,R1
```

```
LD R7, Y+

MUL R2,R7
MOVW R12,R0

MUL R3,R7
ADD R13,R0
CLR R14
ADC R14,R1

MUL R4,R7
ADD R14,R0
CLR R15
ADC R15,R1

ADD R8,R12
ST Z+,R8
ADC R9,R13
ADC R10,R14
CLR R11
ADC R11,R15
```

Let's do that on the AVR

```
LD R2, X+
LD R3, X+
LD R4, X+

LD R7, Y+

MUL R2,R7
ST Z+,R0
MOV R8,R1

MUL R3,R7
ADD R8,R0
CLR R9
ADC R9,R1

MUL R4,R7
ADD R9,R0
CLR R10
ADC R10,R1
```

```
LD R7, Y+

MUL R2,R7
MOVW R12,R0

MUL R3,R7
ADD R13,R0
CLR R14
ADC R14,R1

MUL R4,R7
ADD R14,R0
CLR R15
ADC R15,R1

ADD R8,R12
ST Z+,R8
ADC R9,R13
ADC R10,R14
CLR R11
ADC R11,R15
```

```
LD R7, Y+

MUL R2,R7
MOVW R12,R0

MUL R3,R7
ADD R13,R0
CLR R14
ADC R14,R1

MUL R4,R7
ADD R14,R0
CLR R15
ADC R15,R1

ADC R9,R12
ST Z+,R9
ADC R10,R13
ADC R11,R14
CLR R12
ADC R12,R15
```

Let's do that on the AVR

```
LD R2, X+
LD R3, X+
LD R4, X+

LD R7, Y+

MUL R2,R7
ST Z+,R0
MOV R8,R1

MUL R3,R7
ADD R8,R0
CLR R9
ADC R9,R1

MUL R4,R7
ADD R9,R0
CLR R10
ADC R10,R1
```

```
LD R7, Y+

MUL R2,R7
MOVW R12,R0

MUL R3,R7
ADD R13,R0
CLR R14
ADC R14,R1

MUL R4,R7
ADD R14,R0
CLR R15
ADC R15,R1

ADD R8,R12
ST Z+,R8
ADC R9,R13
ADC R10,R14
CLR R11
ADC R11,R15
```

```
LD R7, Y+

MUL R2,R7
MOVW R12,R0

MUL R3,R7
ADD R13,R0
CLR R14
ADC R14,R1

MUL R4,R7
ADD R14,R0
CLR R15
ADC R15,R1

ADC R9,R12
ST Z+,R9
ADC R10,R13
ADC R11,R14
CLR R12
ADC R12,R15
```

```
ST Z+,R10
ST Z+,R11
ST Z+,R12
```

Let's do that on the AVR

- Problem: Need $3n + c$ registers for $n \times n$-byte multiplication

Let's do that on the AVR

- Problem: Need $3n + c$ registers for $n \times n$-byte multiplication
- Can add on the fly, get down to $2n + c$, but more carry handling

Can we do better?

> *"Again as the information is understood, the multiplication of 2345 by 6789 is proposed; therefore the numbers are written down; the 5 is multiplied by the 9, there will be 45; the 5 is put, the 4 is kept; and the 5 is multiplied by the 8, and the 9 by the 4 and the products are added to the kept 4; there will be 80; the 0 is put and the 8 is kept; and the 5 is multiplied by the 7 and the 9 by the 2 and the 4 by the 8, and the products are added to the kept 8; there will be 102; the 2 is put and the 10 is kept in hand. . ."*

From "Fibonacci's Liber Abaci" (1202) Chapter 2
(English translation by Sigler)

Product scanning on the AVR

```
LD R2, X+
LD R3, X+
LD R4, X+
LD R7, Y+
LD R8, Y+
LD R9, Y+


MUL R2, R7
MOV R13, R1
STD Z+0, R0
CLR R14
CLR R15

MUL R2, R8
ADD R13, R0
ADC R14, R1
MUL R3, R7
ADD R13, R0
ADC R14, R1
ADC R15, R5
STD Z+1, R13
CLR R16
```

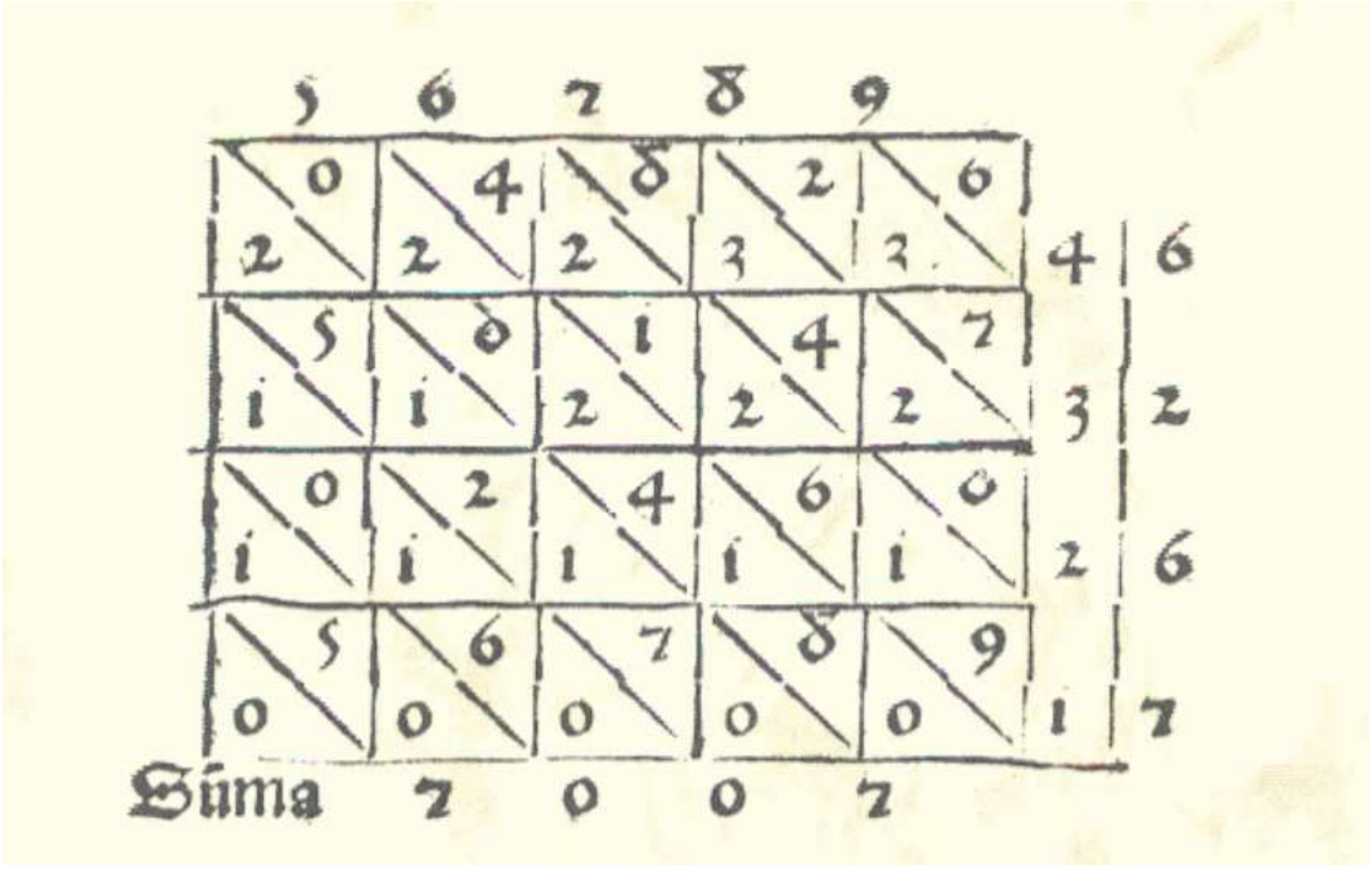
```
MUL R2, R9
ADD R14, R0
ADC R15, R1
ADC R16, R5
MUL R3, R8
ADD R14, R0
ADC R15, R1
ADC R16, R5
MUL R4, R7
ADD R14, R0
ADC R15, R1
ADC R16, R5
STD Z+2, R14
CLR R17
```

```
MUL R3, R9
ADD R15, R0
ADC R16, R1
ADC R17, R5
MUL R4, R8
ADD R15, R0
ADC R16, R1
ADC R17, R5
STD Z+3, R15

MUL R4, R9
ADD R16, R0
ADC R17, R1
STD Z+4, R16

STD Z+5, R17
```

Even better. . . ?



From the Treviso Arithmetic, 1478 (http://www.republicaveneta.com/doc/abaco.pdf)

Hybrid multiplication

- Idea: Chop whole multiplication into smaller blocks
- Compute each of the smaller multiplications by schoolbook
- Later add up to the full result
- See it as two nested loops:
  - Inner loop performs operand scanning
  - Outer loop performs product scanning

Hybrid multiplication

- Idea: Chop whole multiplication into smaller blocks
- Compute each of the smaller multiplications by schoolbook
- Later add up to the full result
- See it as two nested loops:
  - Inner loop performs operand scanning
  - Outer loop performs product scanning
- Originally proposed by Gura, Patel, Wander, Eberle, Chang Shantz, 2004

Hybrid multiplication

- Idea: Chop whole multiplication into smaller blocks
- Compute each of the smaller multiplications by schoolbook
- Later add up to the full result
- See it as two nested loops:
  - Inner loop performs operand scanning
  - Outer loop performs product scanning
- Originally proposed by Gura, Patel, Wander, Eberle, Chang Shantz, 2004
- Various improvements, consider 160-bit multiplication:
  - Originally: 3106 cycles
  - Uhsadel, Poschmann, Paar (2007): 2881 cycles
  - Scott, Szczechowiak (2007): 2651 cycles
  - Kargl, Pyka, Seuschek (2008): 2593 cycles

Operand-caching multiplication

- Hutter, Wenger, 2011: More efficient way to decompose multiplication
- Inside separate chunks use product-scanning
- Main idea: re-use values in registers for longer

Operand-caching multiplication

- Hutter, Wenger, 2011: More efficient way to decompose multiplication
- Inside separate chunks use product-scanning
- Main idea: re-use values in registers for longer
- Performance:
  - 2393 cycles for 160-bit multiplication
  - 6121 cycles for 256-bit multiplication

Operand-caching multiplication

- Hutter, Wenger, 2011: More efficient way to decompose multiplication
- Inside separate chunks use product-scanning
- Main idea: re-use values in registers for longer
- Performance:
  - 2393 cycles for 160-bit multiplication
  - 6121 cycles for 256-bit multiplication
- Followup-paper by Seo and Kim: "Consecutive operand caching":
  - 2341 cycles for 160-bit multiplication
  - 6115 cycles for 256-bit multiplication

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `MUL`s
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `MUL`s
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `MUL`s
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960
- Idea: write $A \cdot B$ as $(A_0 + 2^m A_1)(B_0 + 2^m B_1)$ for half-size A_0, B_0, A_1, B_1

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `MUL`s
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960
- Idea: write $A \cdot B$ as $(A_0 + 2^m A_1)(B_0 + 2^m B_1)$ for half-size A_0, B_0, A_1, B_1
- Compute

$$A_0 B_0 + \qquad 2^m (A_0 B_1 + B_0 A_1) \qquad + 2^{2m} A_1 B_1$$

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `MUL`s
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960
- Idea: write $A \cdot B$ as $(A_0 + 2^m A_1)(B_0 + 2^m B_1)$ for half-size A_0, B_0, A_1, B_1
- Compute

$$\begin{aligned} &A_0B_0 + && 2^m(A_0B_1 + B_0A_1) && + 2^{2m}A_1B_1 \\ =&A_0B_0 + 2^m((A_0 + A_1)(B_0 + B_1) - A_0B_0 - A_1B_1) + 2^{2m}A_1B_1 \end{aligned}$$

Multiplication complexity

- So far, multiplication of 2 n-byte numbers needs n^2 `MUL`s
- Kolmogorov conjectured 1952: You can't do better, multiplication has quadratic complexity
- Proven wrong by 23-year old student Karatsuba in 1960
- Idea: write $A \cdot B$ as $(A_0 + 2^m A_1)(B_0 + 2^m B_1)$ for half-size A_0, B_0, A_1, B_1
- Compute

$$\begin{aligned} &A_0B_0 + && 2^m(A_0B_1 + B_0A_1) && + 2^{2m}A_1B_1 \\ =&A_0B_0 + 2^m((A_0 + A_1)(B_0 + B_1) - A_0B_0 - A_1B_1) + 2^{2m}A_1B_1 \end{aligned}$$

- Recursive application yields $\Theta(n^{\log_2 3})$ runtime
- Can do more on Karatsuba on microcontrollers later...

What changes on the Cortex-M4?

- 32-bit registers; multiplier produces 64-bit result
- Different instruction set, and instruction timings

What changes on the Cortex-M4?

- 32-bit registers; multiplier produces 64-bit result
- Different instruction set, and instruction timings
- More flexibility to represent big integers!
- So far, represent, e.g., 130-bit integer A as $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{32i}$$

- Highest coefficient really needs only two bits
- Need to handle carries the whole time

What changes on the Cortex-M4?

- 32-bit registers; multiplier produces 64-bit result
- Different instruction set, and instruction timings
- More flexibility to represent big integers!
- So far, represent, e.g., 130-bit integer A as $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{32i}$$

- Highest coefficient really needs only two bits
- Need to handle carries the whole time
- Alternative representation: $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{26i}$$

What changes on the Cortex-M4?

- 32-bit registers; multiplier produces 64-bit result
- Different instruction set, and instruction timings
- More flexibility to represent big integers!
- So far, represent, e.g., 130-bit integer A as $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{32i}$$

- Highest coefficient really needs only two bits
- Need to handle carries the whole time
- Alternative representation: $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{26i}$$

- Now the representation is "redundant", e.g., 2^{27} can be written as $(2^{27}, 0, 0, 0, 0)$ or $(0, 2, 0, 0, 0)$

What changes on the Cortex-M4?

- 32-bit registers; multiplier produces 64-bit result
- Different instruction set, and instruction timings
- More flexibility to represent big integers!
- So far, represent, e.g., 130-bit integer A as $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{32i}$$

- Highest coefficient really needs only two bits
- Need to handle carries the whole time
- Alternative representation: $(a_0, a_1, a_2, a_3, a_4)$ with

$$A = \sum_{i=0}^{4} a_i 2^{26i}$$

- Now the representation is "redundant", e.g., 2^{27} can be written as $(2^{27}, 0, 0, 0, 0)$ or $(0, 2, 0, 0, 0)$
- Carry handling can be delayed ("carry-save representation")
- Much easier to write code in C!

Elements of $\mathbb{F}_{2^{130}-5}$ in radix-2^{26}

```
typedef struct {
  uint32_t v[5];
} gfe;

void gfe_add(gfe *r, const gfe *a, const gfe *b)
{
  int i;
  for(i=0;i<5;i++)
    r->v[i] = a->v[i] + b->v[i];
}
```

Note that this code would be the same for polynomial arithmetic!

. . . and multiplication

```
int i,j;
uint64_t t[9];
for(i=0;i<9;i++) t[i] = 0;

for(i=0;i<5;i++)
  for(j=0;j<5;j++)
    t[i+j] += (uint64_t)a->v[i] * b->v[j];

for(i=5;i<9;i++) t[i-5] += 5*t[i];

for(i=0;i<4;i++) {
  t[i+1] += t[i] >> 26;
  t[i] &= 0x3ffffff;
}
t[0] += 5*(t[4] >> 26);
t[4] &= 0x3ffffff;
t[1] += t[0] >> 26;
t[0] &= 0x3ffffff;

for(i=0;i<5;i++)
  r->v[i] = t[i];
```

Useful features of the M4

- Mainly the multiply and multiply-accumulate instructions UMULL and UMLAL
- UMULL produces multiplication result in two 32-bit registers
- UMLAL accumulates multiplication result into two 32-bit registers

Useful features of the M4

- Mainly the multiply and multiply-accumulate instructions UMULL and UMLAL
- UMULL produces multiplication result in two 32-bit registers
- UMLAL accumulates multiplication result into two 32-bit registers

Optimization strategy on the M4

- Reference implementation uses radix 2^8
- Change to radix 2^{26} (in C, see code from previous slides)
- Implement "unpack" and "pack" to convert from byte arrays
- Implement modular multiplication and addition
- Once this works in C, move to assembly ;-)

Let's start

- 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 and Poly1305