Engineering Cryptographic Software

Cryptography in software – the basics

Radboud University, Nijmegen, The Netherlands

Winter 2024/25

The software arena(s)

Embedded microcontrollers

- This is what you're looking at in the software assignment
- Typically very tight size constraints (ROM and RAM)
- Different optimization targets: size, speed
- No (or very little) parallel computation capabilities

The software arena(s)

Embedded microcontrollers

- This is what you're looking at in the software assignment
- Typically very tight size constraints (ROM and RAM)
- Different optimization targets: size, speed
- No (or very little) parallel computation capabilities

Servers, workstations, laptops, smartphones

- No *serious* size constraints for crypto
- Optimization target: speed (high throughput or **low latency**)
- Various different levels of parallelism

The software arena(s)

Embedded microcontrollers

- This is what you're looking at in the software assignment
- Typically very tight size constraints (ROM and RAM)
- Different optimization targets: size, speed
- No (or very little) parallel computation capabilities

Servers, workstations, laptops, smartphones

- No *serious* size constraints for crypto
- Optimization target: speed (high throughput or **low latency**)
- Various different levels of parallelism

GPUs

- Special size restrictions apply for good performance
- Optimization target: speed (**high throughput** or low latency)
- Highly parallel architectures

Throughput vs. Latency

- Some software makes extensive use of *batching*
- Faster for many computations, if those are performed “together”

Throughput vs. Latency

- Some software makes extensive use of *batching*
- Faster for many computations, if those are performed “together”
- Example: McBits software (Bernstein, Chou, Schwabe, 2013):
  - 15486208 cycles on Intel Ivy Bridge for 256 decryptions
  - **NOT:** $15486208/256 = 60493$ cycles for one decryption.

Throughput vs. Latency

- Some software makes extensive use of *batching*
- Faster for many computations, if those are performed “together”
- Example: McBits software (Bernstein, Chou, Schwabe, 2013):
  - 15486208 cycles on Intel Ivy Bridge for 256 decryptions
  - **NOT:** $15486208/256 = 60493$ cycles for one decryption.
  - Software needs to wait until enough inputs are available
  - Delay from input to output is delay of 256 decryptions

Throughput vs. Latency

- Some software makes extensive use of *batching*
- Faster for many computations, if those are performed "together"
- Example: McBits software (Bernstein, Chou, Schwabe, 2013):
  - 15486208 cycles on Intel Ivy Bridge for 256 decryptions
  - **NOT:** $15486208/256 = 60493$ cycles for one decryption.
  - Software needs to wait until enough inputs are available
  - Delay from input to output is delay of 256 decryptions
- Highly parallel architectures (e.g., GPUs) focus on throughput
- This can be a problem for, e.g., low-latency network communication

Benchmarking software

- Tools like `time` or `time.h` have too low resolution
- For serious optimization need to count CPU cycles

Benchmarking software

- Tools like `time` or `time.h` have too low resolution
- For serious optimization need to count CPU cycles
- Use CPU's built-in cycle counter, e.g., on AMD64:

```
static long long cpucycles(void)
{
  unsigned long long result;
  asm volatile("rdtsc;"
               "shlq $32,%%rdx;"
               "orq %%rdx,%%rax"
               : "=a" (RES)
               :
               :  "%rdx");
  return result;
}
```

Benchmarking pitfalls

1. Your program is not running exclusively on the CPU, there may be interrupts
   Solution: Measure many times, take the *median* (not average!)
   Remark: Also report quartiles

Benchmarking pitfalls

1. Your program is not running exclusively on the CPU, there may be interrupts
   Solution: Measure many times, take the *median* (not average!)
   Remark: Also report quartiles
2. The `rdtsc` instruction reports *reference* cycles, your CPU may run at a different speed
   Solution: Switch off frequency scaling and TurboBoost/TurboCore

Benchmarking pitfalls

1. Your program is not running exclusively on the CPU, there may be interrupts
   Solution: Measure many times, take the *median* (not average!)
   Remark: Also report quartiles
2. The `rdtsc` instruction reports *reference* cycles, your CPU may run at a different speed
   Solution: Switch off frequency scaling and TurboBoost/TurboCore
3. Hyperthreading may run another process on the same physical core as your program
   Solution: Switch off hyperthreading

Benchmarking pitfalls

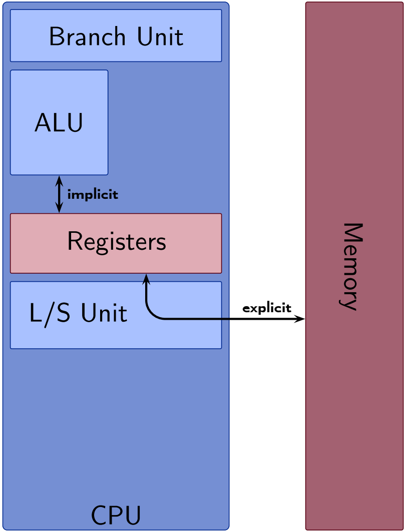
1. Your program is not running exclusively on the CPU, there may be interrupts
   Solution: Measure many times, take the *median* (not average!)
   Remark: Also report quartiles
2. The `rdtsc` instruction reports *reference* cycles, your CPU may run at a different speed
   Solution: Switch off frequency scaling and TurboBoost/TurboCore
3. Hyperthreading may run another process on the same physical core as your program
   Solution: Switch off hyperthreading
4. Getting reproducible, publicly verifiable benchmarks is hard
   Solution: Use public benchmarking framework SUPERCOP by Bernstein and Lange:

   `http://bench.cr.yp.to`

   Remark: Please submit cryptographic software to eBACS!

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"

A first program

Adding up 1000 integers

1. Set register R1 to zero
2. Set register R2 to zero
3. Load 32-bits from address START+R2 into register R3
4. Add 32-bit integers in R1 and R3, write the result in R1
5. Increase value in register R2 by 4
6. Compare value in register R2 to 4000
7. Goto line 3 if R2 was smaller than 4000

A first program

Adding up 1000 integers in readable syntax

```
int32 result
int32 tmp
int32 ctr

result  = 0
ctr     = 0
looptop:
tmp = mem32[START+ctr]
result += tmp
ctr += 4
unsigned<? ctr - 4000
goto looptop if unsigned<
```

Running the program

- Easy approach: Per “time-slot” (*cycle*) execute one instruction, then go for the next
- Cycles needs to be long enough to finish the most complex supported instruction

Running the program

- Easy approach: Per "time-slot" (*cycle*) execute one instruction, then go for the next
- Cycles needs to be long enough to finish the most complex supported instruction
- Other approach: Chop instructions into smaller tasks, e.g. for addition:
    1. Fetch instruction
    2. Decode instruction
    3. Fetch register arguments
    4. Execute (actual addition)
    5. Write back to register

Running the program

- Easy approach: Per “time-slot” (*cycle*) execute one instruction, then go for the next
- Cycles needs to be long enough to finish the most complex supported instruction
- Other approach: Chop instructions into smaller tasks, e.g. for addition:
  1. Fetch instruction
  2. Decode instruction
  3. Fetch register arguments
  4. Execute (actual addition)
  5. Write back to register
- Overlap instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)
- This is called pipelined execution (many more stages possible)
- Advantage: cycles can be much shorter (higher *clock speed*)

Running the program

- Easy approach: Per "time-slot" (*cycle*) execute one instruction, then go for the next
- Cycles needs to be long enough to finish the most complex supported instruction
- Other approach: Chop instructions into smaller tasks, e.g. for addition:
  1. Fetch instruction
  2. Decode instruction
  3. Fetch register arguments
  4. Execute (actual addition)
  5. Write back to register
- Overlap instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)
- This is called pipelined execution (many more stages possible)
- Advantage: cycles can be much shorter (higher *clock speed*)
- Requirement for overlapping execution: instructions have to be independent

Instruction throughput and latency

- While the ALU is executing an instruction the L/S and branch units are idle

Instruction throughput and latency

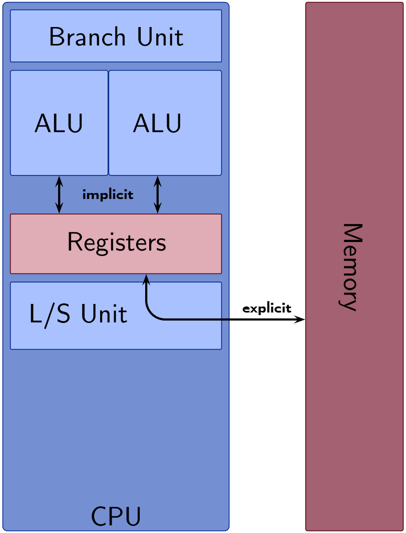
- While the ALU is executing an instruction the L/S and branch units are idle
- Idea: Duplicate fetch and decode, handle two or three instructions per cycle
- While we're at it: Why not deploy two ALUs
- This concept is called *superscalar* execution

Instruction throughput and latency

- While the ALU is executing an instruction the L/S and branch units are idle
- Idea: Duplicate fetch and decode, handle two or three instructions per cycle
- While we're at it: Why not deploy two ALUs
- This concept is called *superscalar* execution
- Number of independent instructions of one type per cycle: **throughput**
- Number of cycles that need to pass before the result can be used: **latency**

An example computer

Still highly simplified



Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle

Adding up 1000 integers on this computer

- Need at least 1000 load instructions: ≥ 1000 cycles

Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle

Adding up 1000 integers on this computer

- Need at least 1000 load instructions: ≥ 1000 cycles
- Need at least 999 addition instructions: ≥ 500 cycles

Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle

Adding up 1000 integers on this computer

- Need at least 1000 load instructions: ≥ 1000 cycles
- Need at least 999 addition instructions: ≥ 500 cycles
- At least 1999 instructions: ≥ 500 cycles

Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle

Adding up 1000 integers on this computer

- Need at least 1000 load instructions: ≥ 1000 cycles
- Need at least 999 addition instructions: ≥ 500 cycles
- At least 1999 instructions: ≥ 500 cycles
- **Lower bound**: 1000 cycles

Latencies and throughputs

- At most 4 instructions per cycle
- At most 1 Load/Store instruction per cycle
- At most 2 arithmetic instructions per cycle
- Arithmetic latency: 2 cycles
- Load latency: 3 cycles
- Branches have to be last instruction in a cycle

How about our program?

```
int32 result
int32 tmp
int32 ctr

result  = 0
ctr     = 0
looptop:
tmp = mem32[START+ctr]
result += tmp
ctr += 4
unsigned<? ctr - 4000
goto looptop if unsigned<
```

How about our program?

```
int32 result
int32 tmp
int32 ctr

result  = 0
ctr     = 0
looptop:
tmp = mem32[START+ctr]
# wait 2 cycles for tmp
result += tmp
ctr += 4
# wait 1 cycle for ctr
unsigned<? ctr - 4000
# wait 1 cycle for unsigned<
goto looptop if unsigned<
```

- Addition has to wait for load
- Comparison has to wait for addition
- Each iteration of the loop takes 8 cycles
- Total: > 8000 cycles

How about our program?

```
int32 result
int32 tmp
int32 ctr

result  = 0
ctr     = 0
looptop:
tmp = mem32[START+ctr]
# wait 2 cycles for tmp
result += tmp
ctr += 4
# wait 1 cycle for ctr
unsigned<? ctr - 4000
# wait 1 cycle for unsigned<
goto looptop if unsigned<
```

- Addition has to wait for load
- Comparison has to wait for addition
- Each iteration of the loop takes 8 cycles
- Total: > 8000 cycles
- **This program sucks!**

Making the program fast

Step 1 – Unrolling

```
result   = 0
tmp = mem32[START+0]
result += tmp
tmp = mem32[START+4]
result += tmp
tmp = mem32[START+8]
result += tmp

...

tmp = mem32[START+3996]
result += tmp
```

- Remove all the loop control: *unrolling*

Making the program fast

Step 1 – Unrolling

```
result  = 0
tmp = mem32[START+0]
# wait 2 cycles for tmp
result += tmp
tmp = mem32[START+4]
# wait 2 cycles for tmp
result += tmp
tmp = mem32[START+8]
# wait 2 cycles for tmp
result += tmp

...

tmp = mem32[START+3996]
# wait 2 cycles for tmp
result += tmp
```

- Remove all the loop control: *unrolling*
- Each load-and-add now takes 3 cycles
- Total: ≈ 3000 cycles

Making the program fast

Step 1 – Unrolling

```
result  = 0
tmp = mem32[START+0]
# wait 2 cycles for tmp
result += tmp
tmp = mem32[START+4]
# wait 2 cycles for tmp
result += tmp
tmp = mem32[START+8]
# wait 2 cycles for tmp
result += tmp

...

tmp = mem32[START+3996]
# wait 2 cycles for tmp
result += tmp
```

- Remove all the loop control: *unrolling*
- Each load-and-add now takes 3 cycles
- Total: ≈ 3000 cycles
- Better, but still too slow

Making the program fast

Step 2 – Instruction Scheduling

```
result = mem32[START + 0]
tmp0   = mem32[START + 4]
tmp1   = mem32[START + 8]
tmp2   = mem32[START +12]

result += tmp0
tmp0 = mem32[START+16]
result += tmp1
tmp1 = mem32[START+20]
result += tmp2
tmp2 = mem32[START+24]

...

result += tmp2
tmp2 = mem32[START+3996]
result += tmp0
result += tmp1
result += tmp2
```

- Load values earlier
- Load latencies are hidden
- Use more registers for loaded values (`tmp0`, `tmp1`, `tmp2`)
- Get rid of one addition to zero

Making the program fast

Step 2 – Instruction Scheduling

```
result = mem32[START + 0]
tmp0   = mem32[START + 4]
tmp1   = mem32[START + 8]
tmp2   = mem32[START +12]
result += tmp0
tmp0 = mem32[START+16]
# wait 1 cycle for result
result += tmp1
tmp1 = mem32[START+20]
# wait 1 cycle for result
result += tmp2
tmp2 = mem32[START+24]

...

result += tmp2
tmp2 = mem32[START+3996]
# wait 1 cycle for result
result += tmp0
# wait 1 cycle for result
result += tmp1
# wait 1 cycle for result
result += tmp2
```

- Load values earlier
- Load latencies are hidden
- Use more registers for loaded values (`tmp0`, `tmp1`, `tmp2`)
- Get rid of one addition to zero
- Now arithmetic latencies kick in
- Total: ≈ 2000 cycles

Making the program fast

Step 3 – More Instruction Scheduling (two accumulators)

```
result0 = mem32[START + 0]
tmp0    = mem32[START + 8]
result1 = mem32[START + 4]
tmp1    = mem32[START +12]
tmp2    = mem32[START +16]

result0 += tmp0
tmp0 = mem32[START+20]
result1 += tmp1
tmp1 = mem32[START+24]
result0 += tmp2
tmp2 = mem32[START+28]

...

result0 += tmp1
tmp1 = mem32[START+3996]
result1 += tmp2
result0 += tmp0
result1 += tmp1
result0 += result1
```

- Use one more accumulator register (`result1`)
- All latencies hidden
- Total: 1004 cycles
- Asymptotically n cycles for n additions

Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles

Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:

Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
    - Unroll the loop

Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop
  - Interleave independent instructions (**instruction scheduling**)

Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop
  - Interleave independent instructions (**instruction scheduling**)
  - Resulting program is larger and requires more registers!

Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop
  - Interleave independent instructions (**instruction scheduling**)
  - Resulting program is larger and requires more registers!
- Note: Good instruction scheduling typically requires more registers

Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop
  - Interleave independent instructions (**instruction scheduling**)
  - Resulting program is larger and requires more registers!
- Note: Good instruction scheduling typically requires more registers
- Opposing requirements to **register allocation** (assigning registers to live variables, minimizing memory access)

Summary of what we did

- Analyze the algorithm in terms of machine instructions
- Look at what the respective machine is able to do
- Compute a lower bound of the cycles
- Optimize until we (almost) reached the lower bound:
  - Unroll the loop
  - Interleave independent instructions (**instruction scheduling**)
  - Resulting program is larger and requires more registers!
- Note: Good instruction scheduling typically requires more registers
- Opposing requirements to **register allocation** (assigning registers to live variables, minimizing memory access)
- Both instruction scheduling and register allocation are NP hard
- So is the joint problem
- Many instances are efficiently solvable

Architectures and microarchitectures

What instructions and how many registers do we have?

- Instructions are defined by the **instruction set**
- Supported register names are defined by the **set of architectural registers**
- Instruction set and set of architectural registers together define the **architecture**
- Examples for architectures: x86, AMD64, ARMv6, ARMv7, UltraSPARC
- Sometimes base architectures are extended, e.g., MMX, SSE, NEON

Architectures and microarchitectures

What instructions and how many registers do we have?

- Instructions are defined by the **instruction set**
- Supported register names are defined by the **set of architectural registers**
- Instruction set and set of architectural registers together define the **architecture**
- Examples for architectures: x86, AMD64, ARMv6, ARMv7, UltraSPARC
- Sometimes base architectures are extended, e.g., MMX, SSE, NEON

What determines latencies etc?

- Different **microarchitectures** implement an architecture
- Latencies and throughputs are specific to a microarchitecture
- Example: Intel Core 2 Quad Q9550 implements the AMD64 architecture

Out-of-order execution

- Optimal instruction scheduling depends on the microarchitecture
- Code optimized for one microarchitecture may run at very bad performance on another microarchitecture
- Many software is shipped in binary form (cannot recompile)

Out-of-order execution

- Optimal instruction scheduling depends on the microarchitecture
- Code optimized for one microarchitecture may run at very bad performance on another microarchitecture
- Many software is shipped in binary form (cannot recompile)
- Idea: Let the processor reschedule instructions on the fly
- Look ahead a few instructions, pick one that can be executed
- This is called **out-of-order execution**

Out-of-order execution

- Optimal instruction scheduling depends on the microarchitecture
- Code optimized for one microarchitecture may run at very bad performance on another microarchitecture
- Many software is shipped in binary form (cannot recompile)
- Idea: Let the processor reschedule instructions on the fly
- Look ahead a few instructions, pick one that can be executed
- This is called **out-of-order execution**
- Typically requires more physical than architectural registers and **register renaming**

Out-of-order execution

- Optimal instruction scheduling depends on the microarchitecture
- Code optimized for one microarchitecture may run at very bad performance on another microarchitecture
- Many software is shipped in binary form (cannot recompile)
- Idea: Let the processor reschedule instructions on the fly
- Look ahead a few instructions, pick one that can be executed
- This is called **out-of-order execution**
- Typically requires more physical than architectural registers and **register renaming**
- Harder for the (assembly) programmer to understand what exactly will happen with the code
- Harder to come up with optimal scheduling

Out-of-order execution

- Optimal instruction scheduling depends on the microarchitecture
- Code optimized for one microarchitecture may run at very bad performance on another microarchitecture
- Many software is shipped in binary form (cannot recompile)
- Idea: Let the processor reschedule instructions on the fly
- Look ahead a few instructions, pick one that can be executed
- This is called **out-of-order execution**
- Typically requires more physical than architectural registers and **register renaming**
- Harder for the (assembly) programmer to understand what exactly will happen with the code
- Harder to come up with optimal scheduling
- Harder to screw up completely

Optimizing Crypto vs. optimizing "something"

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

Optimizing Crypto vs. optimizing “something”

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

Optimizing Crypto vs. optimizing “something”

- 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 Crypto vs. optimizing “something”

- 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
- Most critical for software implementations on “large” CPUs: software must take constant time (independent of secret data)

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?

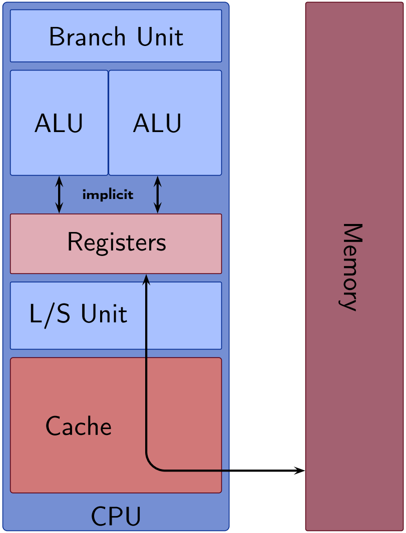
```
if $s$ then
    $r \leftarrow A$
else
    $r \leftarrow B$
end if
```

- Replace by

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

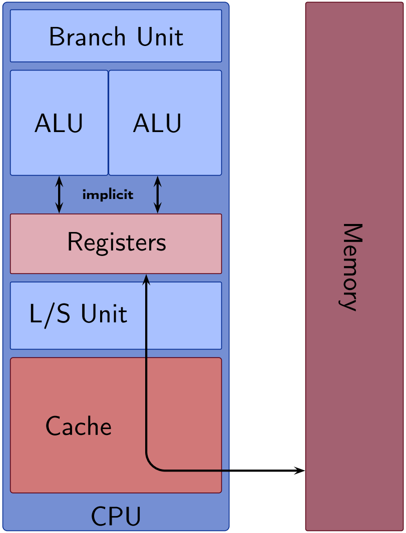
- Can expand s to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication
- For very fast A and B this can even be faster

Cached memory access



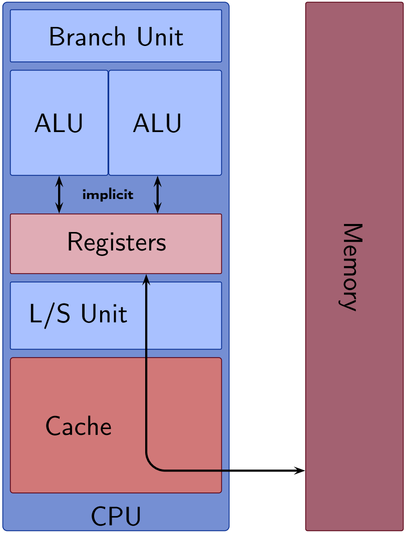
- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data

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



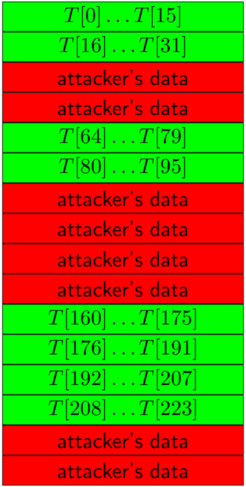
- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data
- A load from memory places data also in the cache
- Data remains in cache until it's replaced by other data
- Loading data is fast if data is in the cache (**cache hit**)
- Loading data is slow if data 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



$T[0] \dots T[15]$

$T[16] \dots T[31]$

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

$T[160] \dots T[175]$

$T[176] \dots T[191]$

$T[192] \dots T[207]$

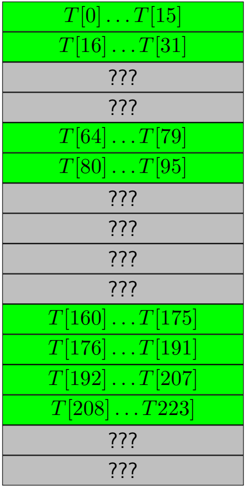
$T[208] \dots T[223]$

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



$T[0]\dots T[15]$

$T[16]\dots T[31]$

???

???

$T[64]\dots T[79]$

$T[80]\dots T[95]$

???

???

???

???

$T[160]\dots T[175]$

$T[176]\dots T[191]$

$T[192]\dots T[207]$

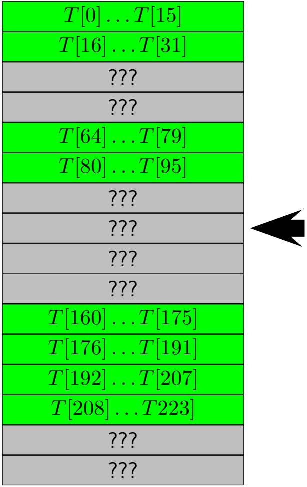
$T[208]\dots T223]$

???

???

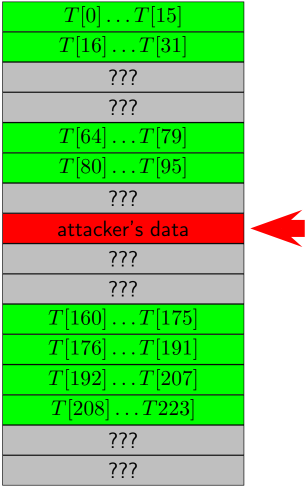
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again

Timing leakage part II



- 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



$T[0] \dots T[15]$

$T[16] \dots T[31]$

???

???

$T[64] \dots T[79]$

$T[80] \dots T[95]$

???

attacker's data

???

???

$T[160] \dots T[175]$

$T[176] \dots T[191]$

$T[192] \dots T[207]$

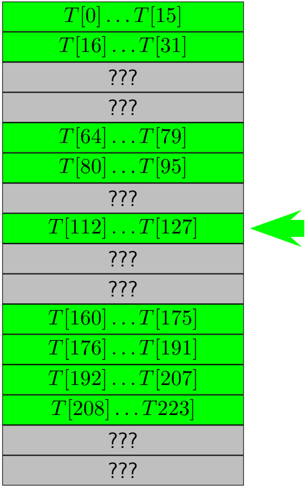
$T[208] \dots T223]$

???

???

- 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 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
- Test this with `valgrind` and *uninitialized secret data* (see `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
- Test this with `valgrind` and *uninitialized secret data* (see 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
- Test this with `valgrind` and *uninitialized secret data* (see `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

Compilers don't help here

- Example for dangerous code:

```
for(j=0;j<8;j++) {
  mask = -(int16_t)((msg[i] >> j)&1);
  r->coeffs[8*i+j] = mask & ((KYBER_Q+1)/2);
}
```

- Another example:

```
  t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
```

- Moritz Schneider, Daniele Lain Ivan Puddu Nicolas Dutly Srdjan Čapkun: *Breaking Bad: How Compilers Break Constant-Time Implementations* https://arxiv.org/pdf/2410.13489

“The multicore revolution”

- Until early years 2000 each new processor generation had higher clock speeds
- Nowadays: increase performance by number of cores:
  - My laptop has 2 physical (and 4 virtual) cores
  - Smartphones typically have 2 or 4 cores
  - Servers have 4, 8, 16,. . . cores
  - Special-purpose hardware (e.g., GPUs) often comes with many more cores
- Consequence: “The free lunch is over” (Herb Sutter, 2005)

“The multicore revolution”

- Until early years 2000 each new processor generation had higher clock speeds
- Nowadays: increase performance by number of cores:
  - My laptop has 2 physical (and 4 virtual) cores
  - Smartphones typically have 2 or 4 cores
  - Servers have 4, 8, 16,… cores
  - Special-purpose hardware (e.g., GPUs) often comes with many more cores
- Consequence: “The free lunch is over” (Herb Sutter, 2005)

“As a result, system designers and software engineers can no longer rely on increasing clock speed to hide software bloat. Instead, they must somehow learn to make effective use of increasing parallelism.”

—Maurice Herlihy: The Multicore Revolution, 2007

Why multicore doesn't matter…

… for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

Why multicore doesn't matter…

… for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

Why multicore doesn't matter…

… for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

- **If you perform only one crypto operation, you don't care**

Why multicore doesn't matter…

… for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

- **If you perform only one crypto operation, you don't care**
- **Many crypto operations are trivially parallel on multiple cores**

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 4 consecutive 32-bit integers (a_0, a_1, a_2, a_3)
- Load 4 consecutive 32-bit integers (b_0, b_1, b_2, b_3)
- Perform addition $(c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- Store 128-bit vector (c_0, c_1, c_2, c_3)

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 4 consecutive 32-bit integers (a_0, a_1, a_2, a_3)
- Load 4 consecutive 32-bit integers (b_0, b_1, b_2, b_3)
- Perform addition $(c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- Store 128-bit vector (c_0, c_1, c_2, c_3)

- Perform the same operations on independent data streams (SIMD)
- Vector instructions available on most "large" processors
- Instructions for vectors of bytes, integers, floats...

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 4 consecutive 32-bit integers (a_0, a_1, a_2, a_3)
- Load 4 consecutive 32-bit integers (b_0, b_1, b_2, b_3)
- Perform addition $(c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- Store 128-bit vector (c_0, c_1, c_2, c_3)

- Perform the same operations on independent data streams (SIMD)
- Vector instructions available on most "large" processors
- Instructions for vectors of bytes, integers, floats...
- Need to interleave data items (e.g., 32-bit integers) in memory
- Compilers will not help with vectorization

Vector computations

Scalar computation

- Load 32-bit integer a
- Load 32-bit integer b
- Perform addition $c \leftarrow a + b$
- Store 32-bit integer c

Vectorized computation

- Load 4 consecutive 32-bit integers (a_0, a_1, a_2, a_3)
- Load 4 consecutive 32-bit integers (b_0, b_1, b_2, b_3)
- Perform addition $(c_0, c_1, c_2, c_3) \leftarrow (a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- Store 128-bit vector (c_0, c_1, c_2, c_3)

- Perform the same operations on independent data streams (SIMD)
- Vector instructions available on most “large” processors
- Instructions for vectors of bytes, integers, floats. . .
- Need to interleave data items (e.g., 32-bit integers) in memory
- Compilers will not really help with vectorization

Back to adding up 1000 integers

- Imagine that
  - vector addition is as fast as scalar addition
  - vector loads are as fast as scalar loads

Back to adding up 1000 integers

- Imagine that
    - vector addition is as fast as scalar addition
    - vector loads are as fast as scalar loads
- Need only 250 vector additions, 250 vector loads ($+$ adding up 4 partial sums)
- Lower bound of 250 cycles

Back to adding up 1000 integers

- Imagine that
  - vector addition is as fast as scalar addition
  - vector loads are as fast as scalar loads
- Need only 250 vector additions, 250 vector loads ($+$ adding up 4 partial sums)
- Lower bound of 250 cycles
- Very straight-forward modification of the program
- Fully unrolled loop needs only $1/4$ of the space

Is it really that efficient?

- Consider the Intel Skylake processor with AVX2

Is it really that efficient?

- Consider the Intel Skylake processor with AVX2
    - 32-bit load throughput: 2 per cycle
    - 32-bit add throughput: 4 per cycle
    - 32-bit store throughput: 1 per cycle

Is it really that efficient?

- Consider the Intel Skylake processor with AVX2
  - 32-bit load throughput: 2 per cycle
  - 32-bit add throughput: 4 per cycle
  - 32-bit store throughput: 1 per cycle
  - 256-bit load throughput: 2 per cycle
  - 8× 32-bit add throughput: 3 per cycle
  - 256-bit store throughput: 1 per cycle

Is it really that efficient?

- Consider the Intel Skylake processor with AVX2
    - 32-bit load throughput: 2 per cycle
    - 32-bit add throughput: 4 per cycle
    - 32-bit store throughput: 1 per cycle
    - 256-bit load throughput: 2 per cycle
    - 8× 32-bit add throughput: 3 per cycle
    - 256-bit store throughput: 1 per cycle
- **AVX2 vector instructions are almost as fast as scalar instructions but do 8× the work**

Is it really that efficient?

- Consider the Intel Skylake processor with AVX2
  - 32-bit load throughput: 2 per cycle
  - 32-bit add throughput: 4 per cycle
  - 32-bit store throughput: 1 per cycle
  - 256-bit load throughput: 2 per cycle
  - 8× 32-bit add throughput: 3 per cycle
  - 256-bit store throughput: 1 per cycle
- **AVX2 vector instructions are almost as fast as scalar instructions but do 8× the work**
- Situation on other architectures/microarchitectures is similar
- Reason: cheap way to increase arithmetic throughput (less decoding, address computation, etc.)

More reasons for using vector arithmetic

- Data-dependent branches are expensive in SIMD
- Variably indexed loads (lookups) into vectors are expensive
- Need to rewrite algorithms to eliminate branches and lookups

More reasons for using vector arithmetic

- Data-dependent branches are expensive in SIMD
- Variably indexed loads (lookups) into vectors are expensive
- Need to rewrite algorithms to eliminate branches and lookups
- Secret-data-dependent branches and secret branch conditions are the major sources of timing-attack vulnerabilities

More reasons for using vector arithmetic

- Data-dependent branches are expensive in SIMD
- Variably indexed loads (lookups) into vectors are expensive
- Need to rewrite algorithms to eliminate branches and lookups
- Secret-data-dependent branches and secret branch conditions are the major sources of timing-attack vulnerabilities
- Strong synergies between speeding up code with vector instructions and protecting code!

Vectorization problems I

Carry handling

- When adding two 32-bit integers, the result may have 33 bits (32-bit result + carry)
- Scalar additions keep the carry in a special *flag register*
- Subsequent instructions can use this flag, e.g., “add with carry”

Vectorization problems I

Carry handling

- When adding two 32-bit integers, the result may have 33 bits (32-bit result + carry)
- Scalar additions keep the carry in a special *flag register*
- Subsequent instructions can use this flag, e.g., “add with carry”
- How about carries of vector additions?
  - Answer 1: Special “carry generate” instruction (e.g., CBE-SPU)

Vectorization problems I

Carry handling

- When adding two 32-bit integers, the result may have 33 bits (32-bit result + carry)
- Scalar additions keep the carry in a special *flag register*
- Subsequent instructions can use this flag, e.g., “add with carry”
- How about carries of vector additions?
    - Answer 1: Special “carry generate” instruction (e.g., CBE-SPU)
    - Answer 2: They’re lost, recomputation is expensive

Vectorization problems I

Carry handling

- When adding two 32-bit integers, the result may have 33 bits (32-bit result + carry)
- Scalar additions keep the carry in a special *flag register*
- Subsequent instructions can use this flag, e.g., "add with carry"
- How about carries of vector additions?
  - Answer 1: Special "carry generate" instruction (e.g., CBE-SPU)
  - Answer 2: They're lost, recomputation is expensive
- Need to *avoid carries* instead of handling them
- No problem for today's lecture, but requires care for big-integer arithmetic

Vectorization problems II

Removing instruction-level parallelism

- If we don't vectorize we perform multiple independent instructions
- We turn *data-level parallelism (DLP)* into *instruction-level parallelism (ILP)*

Vectorization problems II

Removing instruction-level parallelism

- If we don't vectorize we perform multiple independent instructions
- We turn *data-level parallelism (DLP)* into *instruction-level parallelism (ILP)*
- Pipelined and multiscalar execution need ILP
- Vectorization removes ILP
- Problematic for algorithms with, e.g., 4-way DLP

Vectorization problems II

Removing instruction-level parallelism

- If we don't vectorize we perform multiple independent instructions
- We turn *data-level parallelism (DLP)* into *instruction-level parallelism (ILP)*
- Pipelined and multiscalar execution need ILP
- Vectorization removes ILP
- Problematic for algorithms with, e.g., 4-way DLP
- Good example to see this: ChaCha vs. Blake
- Vectorization of ChaCha can resort to higher-level parallelism (multiple blocks)
- Harder for Blake: each block depends on the previous one

Vectorization problems III

Data shuffeling

- Consider multiplication of 4-coefficient polynomials $f = f_0 + f_1x + f_2x^2 + f_3x^3$ and $g = g_0 + g_1x + g_2x^2 + g_3x^3$:

$$\begin{aligned}
r_0 &= f_0g_0 \\
r_1 &= f_0g_1 + f_1g_0 \\
r_2 &= f_0g_2 + f_1g_1 + f_2g_0 \\
r_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 \\
r_4 &= f_1g_3 + f_2g_2 + f_3g_1 \\
r_5 &= f_2g_3 + f_3g_2 \\
r_6 &= f_3g_3
\end{aligned}$$

Vectorization problems III

Data shuffeling

- Consider multiplication of 4-coefficient polynomials $f = f_0 + f_1x + f_2x^2 + f_3x^3$ and $g = g_0 + g_1x + g_2x^2 + g_3x^3$:

$$
\begin{aligned}
r_0 &= f_0g_0\\
r_1 &= f_0g_1 + f_1g_0\\
r_2 &= f_0g_2 + f_1g_1 + f_2g_0\\
r_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0\\
r_4 &= f_1g_3 + f_2g_2 + f_3g_1\\
r_5 &= f_2g_3 + f_3g_2\\
r_6 &= f_3g_3
\end{aligned}
$$

- Ignore carries, overflows etc. for a moment
- 16 multiplications, 9 additions
- How to vectorize multiplications?

Vectorization problems III

Data shuffeling

$$\begin{aligned}
r_0 &= f_0g_0\\
r_1 &= f_0g_1 + f_1g_0\\
r_2 &= f_0g_2 + f_1g_1 + f_2g_0\\
r_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0\\
r_4 &= f_1g_3 + f_2g_2 + f_3g_1\\
r_5 &= f_2g_3 + f_3g_2\\
r_6 &= f_3g_3
\end{aligned}$$

- Can easily load (f_0, f_1, f_2, f_3) and (g_0, g_1, g_2, g_3)
- Multiply, obtain $(f_0g_0, f_1g_1, f_2g_2, f_3g_3)$

Vectorization problems III

Data shuffeling

$$
\begin{aligned}
r_0 &= f_0 g_0 \\
r_1 &= f_0 g_1 + f_1 g_0 \\
r_2 &= f_0 g_2 + f_1 g_1 + f_2 g_0 \\
r_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 \\
r_4 &= f_1 g_3 + f_2 g_2 + f_3 g_1 \\
r_5 &= f_2 g_3 + f_3 g_2 \\
r_6 &= f_3 g_3
\end{aligned}
$$

- Can easily load (f_0, f_1, f_2, f_3) and (g_0, g_1, g_2, g_3)
- Multiply, obtain $(f_0 g_0, f_1 g_1, f_2 g_2, f_3 g_3)$
- And now what?

Vectorization problems III

Data shuffeling

$$
\begin{aligned}
r_0 &= f_0g_0\\
r_1 &= f_0g_1 + f_1g_0\\
r_2 &= f_0g_2 + f_1g_1 + f_2g_0\\
r_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0\\
r_4 &= f_1g_3 + f_2g_2 + f_3g_1\\
r_5 &= f_2g_3 + f_3g_2\\
r_6 &= f_3g_3
\end{aligned}
$$

- Can easily load (f_0, f_1, f_2, f_3) and (g_0, g_1, g_2, g_3)
- Multiply, obtain $(f_0g_0, f_1g_1, f_2g_2, f_3g_3)$
- And now what?
- Answer: Need to *shuffle* data in input and output registers
- Significant overhead, not clear that vectorization speeds up computation!

Efficient vectorization

- Most important question: Where does the parallelism come from?
- Easiest answer: Consider multiple batched encryptions, decryptions, signature computations, verifications, etc. (but that increases latency)

Efficient vectorization

- Most important question: Where does the parallelism come from?
- Easiest answer: Consider multiple batched encryptions, decryptions, signature computations, verifications, etc. (but that increases latency)
- Often: Can exploit lower-level parallelism

Efficient vectorization

- Most important question: Where does the parallelism come from?
- Easiest answer: Consider multiple batched encryptions, decryptions, signature computations, verifications, etc. (but that increases latency)
- Often: Can exploit lower-level parallelism
- Rule of thumb: parallelize on an as high as possible level
- Vectorization is hard to do as “add-on” optimization
- Reconsider algorithms and data structures, synergy with constant-time algorithms

Bitslicing

- Imagine registers that have only one bit
- Perform arithmetic on those registers using `XOR`, `AND`, `OR`
- Essentially the same as hardware implementations

Bitslicing

- Imagine registers that have only one bit
- Perform arithmetic on those registers using `XOR`, `AND`, `OR`
- Essentially the same as hardware implementations
- But wait, registers are longer!
- Think of them as vectors of bits
- This needs transposition of the “binary data matrix”
- Perform the simulated hardware implementations on many independent data streams

Bitslicing

- Imagine registers that have only one bit
- Perform arithmetic on those registers using `XOR`, `AND`, `OR`
- Essentially the same as hardware implementations
- But wait, registers are longer!
- Think of them as vectors of bits
- This needs transposition of the “binary data matrix”
- Perform the simulated hardware implementations on many independent data streams
- Bitslicing works for every algorithm
- Bitslicing is inherently protected against timing attacks
- Efficient bitslicing needs a huge amount of data-level parallelism

Bitslicing binary polynomials

4-coefficient binary polynomials

$(a_3x^3 + a_2x^2 + a_1x + a_0)$, with $a_i \in \{0, 1\}$

4-coefficient bitsliced binary polynomials

```
typedef unsigned char poly4; /* 4 coefficients in the low 4 bits */
typedef unsigned long long poly4x64[4];

void poly4_bitslice(poly4x64 r, const poly4 x[64])
{
  int i,j;
  for(i=0;i<4;i++)
  {
    r[i] = 0;
    for(j=0;j<64;j++)
      r[i] |= (unsigned long long)(1 & (x[j] >> i))<<j;
  }
}
```

Bitsliced binary-polynomial multiplication

```
typedef unsigned long long poly4x64[4];
typedef unsigned long long poly7x64[7];

void poly4x64_mul(poly7x64 r, const poly4x64 a, const poly4x64 b)
{
  r[0] = a[0] & b[0];
  r[1] = (a[0] & b[1]) ^ (a[1] & b[0]);
  r[2] = (a[0] & b[2]) ^ (a[1] & b[1]) ^ (a[2] & b[0]);
  r[3] = (a[0] & b[3]) ^ (a[1] & b[2]) ^ (a[2] & b[1]) ^ (a[3] & b[0]);
  r[4] = (a[1] & b[3]) ^ (a[2] & b[2]) ^ (a[3] & b[1]);
  r[5] = (a[2] & b[3]) ^ (a[3] & b[2]);
  r[6] = (a[3] & b[3]);
}
```

Bitslicing issues

- XOR, AND, OR, etc are usually fast (e.g., 3 128-bit operations per cycle on Intel Core 2)
- Can be very fast for operations that are not natively supported (like arithmetic in binary fields)

Bitslicing issues

- XOR, AND, OR, etc are usually fast (e.g., 3 128-bit operations per cycle on Intel Core 2)
- Can be very fast for operations that are not natively supported (like arithmetic in binary fields)
- Active data set increases massively (e.g., $128\times$)
- For "normal" vector operations, register space is increased accordingly (e.g, 16 256-bit vector registers vs. 16 64-bit integer registers)
- For bitslicing: Need to fit more data into the same registers
- Typical consequence: more loads and stores (that easily become the performance bottleneck)