Introduction to software implementations

Radboud University, Nijmegen, The Netherlands

June 2, 2015

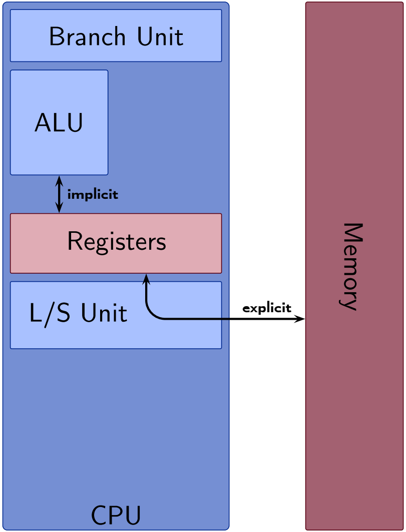
Summer school on real-world crypto and privacy
Šibenik, Croatia

Part I

Making software fast

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

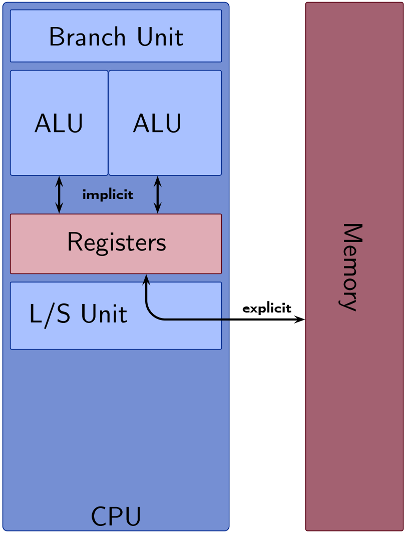
- While the ALU is executing an instruction the L/S and branch units are idle
- Idea: Duplicate fetch and decode, handle two or three instructions per cycle
- While we're at it: Why not deploy two ALUs
- This concept is called *superscalar* execution

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

"Thus we arbitrarily select a reference organization : the IBM 704-70927090. This organization is then regarded as the prototype of the class of machines which we label:
1) Single Instruction Stream–Single Data Stream (SISD).

Three additional organizational classes are evident.
2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream–Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD)"

– Michael J. Flynn. Very high-speed computing systems. 1966.

32-bit integer addition: SISD vs SIMD

SISD

```
int64 a
int64 b
a = mem32[addr1 + 0]
b = mem32[addr2 + 0]
(uint32) a += b
mem32[addr3 + 0] = a
```

SIMD

```
reg128 a
reg128 b
a = mem128[addr1 + 0]
b = mem128[addr2 + 0]
4x a += b
mem128[addr3 + 0] = a
```

Extending our machine...

- The two ALUs can now also do vector instructions
- Load/Store unit can also handle vector loads and stores
- Vector-arithmetic latency : 2 cycles
- Vector-load latency: 3 cycles
- Vector-store latency: 3 cycles

Adding 1000 integers with vector instructions

```
vresult0 = mem128[START +   0]
vtmp0    = mem128[START + 16]
vresult1 = mem128[START + 32]
vtmp1    = mem128[START + 48]
vtmp2    = mem128[START + 64]

4x vresult0 += vtmp0
vtmp0 = mem128[START +  80]
4x vresult1 += vtmp1
vtmp1 = mem128[START +  96]
4x vresult0 += vtmp2
vtmp2 = mem128[START + 112]

...

4x vresult0 += vtmp1
vtmp1 = mem128[START+3984]
4x vresult1 += vtmp2
4x vresult0 += vtmp0
4x vresult1 += vtmp1
4x vresult0 += vresult1
```

- Essentially the same as before
- Always load/add 4 integers
- Produces 4 independent sums in `vresult0`

Adding 1000 integers with vector instructions

```
...

mem128[TMP + 0] = vresult0
result = mem32[TMP +  0]
tmp0   = mem32[TMP +  4]
tmp1   = mem32[TMP +  8]
tmp2   = mem32[TMP + 12]
result += tmp0
result += tmp1
result += tmp2
```

- Essentially the same as before
- Always load/add 4 integers
- Produces 4 independent sums in `vresult0`
- Need to add horizontally across elements in `vresult0`
- Can do that by storing, loading, adding
- Total cost: 266 cycles

Is this realistic?

- Consider the Intel Nehalem processor:

Is this realistic?

- Consider the Intel Nehalem processor:
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle

Is this realistic?

- Consider the Intel Nehalem processor:
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle
  - 128-bit load throughput: 1 per cycle
  - $4\times$ 32-bit add throughput: 2 per cycle
  - 128-bit store throughput: 1 per cycle

Is this realistic?

- Consider the Intel Nehalem processor:
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle
  - 128-bit load throughput: 1 per cycle
  - $4\times$ 32-bit add throughput: 2 per cycle
  - 128-bit store throughput: 1 per cycle
- **Vector instructions are about as fast as scalar instructions but do $4\times$ the work**

Is this realistic?

- Consider the Intel Nehalem processor:
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle
  - 128-bit load throughput: 1 per cycle
  - $4\times$ 32-bit add throughput: 2 per cycle
  - 128-bit store throughput: 1 per cycle
- **Vector instructions are about as fast as scalar instructions but do $4\times$ the work**
- Situation on other architectures/microarchitectures is similar
- Reason: cheapest way to increase computational power

Is this realistic?

- Consider the Intel Nehalem processor:
  - 32-bit load throughput: 1 per cycle
  - 32-bit add throughput: 3 per cycle
  - 32-bit store throughput: 1 per cycle
  - 128-bit load throughput: 1 per cycle
  - $4\times$ 32-bit add throughput: 2 per cycle
  - 128-bit store throughput: 1 per cycle
- **Vector instructions are about as fast as scalar instructions but do $4\times$ the work**
- Situation on other architectures/microarchitectures is similar
- Reason: cheapest way to increase computational power

Why isn't all software vectorized?

Vectorization issues I

Branches and lookups

- Data-dependent branches are tricky
- Only efficient if all vector elements branch in the same direction
- Otherwise: compute both parts of the branch, mask out results

Vectorization issues I

Branches and lookups

- Data-dependent branches are tricky
- Only efficient if all vector elements branch in the same direction
- Otherwise: compute both parts of the branch, mask out results
- Only consecutive loads are cheap
- Variably indexed loads are expensive
- Vectorization does not really like lookup-table-based implementations

Vectorization issues I
Branches and lookups

- Data-dependent branches are tricky
- Only efficient if all vector elements branch in the same direction
- Otherwise: compute both parts of the branch, mask out results
- Only consecutive loads are cheap
- Variably indexed loads are expensive
- Vectorization does not really like lookup-table-based implementations
- Compilers only perform very simple vectorization efficiently
- Typically requires re-thinking data structures and algorithms

Vectorization issues II
Specific parallel computations

- Need *data-level parallelism*
- Non-vectorized software turns data-level parallelism into instruction-level parallelism
- Instruction-level parallelism is important for efficient pipelined and superscalar execution
- Vectorization may conflict with efficient pipelined and superscalar execution

Vectorization issues III

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 issues III

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 issues III

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 issues III

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
- In particular interesting for big-integer arithmetic (see my talk on thursday)

Vectorization issues IV

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 issues IV

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 issues IV

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 $(f_0 g_0, f_1 g_1, f_2 g_2, f_3 g_3)$

Vectorization issues IV

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 issues IV

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!

Summary

- To optimize software, understand algorithms in terms of machine instructions
- Optimization:
  - Pick suitable instructions
  - Instruction scheduling
  - Register allocation

Summary

- To optimize software, understand algorithms in terms of machine instructions
- Optimization:
  - Pick suitable instructions
  - Instruction scheduling
  - Register allocation
- Next level: think vectorized
  - Consider data-level parallelism
  - Think branch-free
  - Think lookup-table free

Part II

Making software secure

Timing Attacks

General idea of those attacks

- Secret data has influence on timing of software
- Attacker measures timing
- Attacker computes influence^{-1} to obtain secret data

Timing Attacks

General idea of those attacks

- Secret data has influence on timing of software
- Attacker measures timing
- Attacker computes influence^{-1} to obtain secret data

Two kinds of remote...

- Timing attacks are a type of side-channel attacks
- Unlike other side-channel attacks, they work remotely:
  - Some need to run attack code in parallel to the target software
  - Attacker can log in remotely (ssh)

Timing Attacks

General idea of those attacks

- Secret data has influence on timing of software
- Attacker measures timing
- Attacker computes influence^{-1} to obtain secret data

Two kinds of remote...

- Timing attacks are a type of side-channel attacks
- Unlike other side-channel attacks, they work remotely:
  - Some need to run attack code in parallel to the target software
  - Attacker can log in remotely (ssh)
  - Some attacks work by measuring network delays
  - Attacker does not even need an account on the target machine

Timing Attacks

General idea of those attacks

- Secret data has influence on timing of software
- Attacker measures timing
- Attacker computes influence^{-1} to obtain secret data

Two kinds of remote…

- Timing attacks are a type of side-channel attacks
- Unlike other side-channel attacks, they work remotely:
  - Some need to run attack code in parallel to the target software
  - Attacker can log in remotely (ssh)
  - Some attacks work by measuring network delays
  - Attacker does not even need an account on the target machine
- Can't protect against timing attacks by locking a room

Timing Attacks

General idea of those attacks

- Secret data has influence on timing of software
- Attacker measures timing
- Attacker computes influence^{-1} to obtain secret data

Two kinds of remote...

- Timing attacks are a type of side-channel attacks
- Unlike other side-channel attacks, they work remotely:
  - Some need to run attack code in parallel to the target software
  - Attacker can log in remotely (ssh)
  - Some attacks work by measuring network delays
  - Attacker does not even need an account on the target machine
- Can't protect against timing attacks by locking a room
- We *can* systematically eliminate all timing attacks!

Exponentiation

- Core operation in RSA, DSA, ElGamal, ECC: exponentiation (or scalar multiplication) with secret exponent (or scalar).

Exponentiation

- Core operation in RSA, DSA, ElGamal, ECC: exponentiation (or scalar multiplication) with secret exponent (or scalar).

Example: exponent 105

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$

Exponentiation

- Core operation in RSA, DSA, ElGamal, ECC: exponentiation (or scalar multiplication) with secret exponent (or scalar).

Example: exponent 105

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
- $105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

Exponentiation

- Core operation in RSA, DSA, ElGamal, ECC: exponentiation (or scalar multiplication) with secret exponent (or scalar).

Example: exponent 105

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
- $105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- $105 = ((((((((((1 \cdot 2 + 1) \cdot 2) + 0) \cdot 2) + 1) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + 1$ (Horner's rule)
- $a^{105} = ((((((((((a^2 \cdot a)^2) \cdot 1)^2) \cdot a)^2) \cdot 1)^2) \cdot 1)^2) \cdot a$

Exponentiation

- Core operation in RSA, DSA, ElGamal, ECC: exponentiation (or scalar multiplication) with secret exponent (or scalar).

Example: exponent 105

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
- $105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
- $105 = ((((((((((1 \cdot 2 + 1) \cdot 2) + 0) \cdot 2) + 1) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + 1$ (Horner's rule)
- $a^{105} = ((((((((((a^2 \cdot a)^2) \cdot 1)^2) \cdot a)^2) \cdot 1)^2) \cdot 1)^2) \cdot a$
- Cost: 6 squarings, 3 multiplications
- More generally: 1 squaring per bit, 1 multiplication per 1-bit

Example: exponentiation mod $2^{31} - 1$

```
// Multiplicative group of integers mod 2^31-1
typedef uint32_t group_t;

/* Modular multiplication */
static void group_mul(group_t *r, const group_t *x, const group_t *y)
{
  *r = ((uint64_t) *x * *y) % 0x7FFFFFFF;
}
```

Example: exponentiation mod $2^{31} - 1$

```
// Multiplicative group of integers mod 2^31-1
typedef uint32_t group_t;

/* Modular multiplication */
static void group_mul(group_t *r, const group_t *x, const group_t *y)
{
  *r = ((uint64_t) *x * *y) % 0x7FFFFFFF;
}
```

- Group is way too small for cryptographic purposes
- Exponentation in this group is just fine to illustrate timing leaks

Example: exponentiation mod $2^{31} - 1$

```
// Multiplicative group of integers mod 2^31-1
typedef uint32_t group_t;

/* Modular multiplication */
static void group_mul(group_t *r, const group_t *x, const group_t *y)
{
  *r = ((uint64_t) *x * *y) % 0x7FFFFFFF;
}
```

- Group is way too small for cryptographic purposes
- Exponentation in this group is just fine to illustrate timing leaks
- From now on consider C code

Square-and-multiply

```
void group_exp(group_t *r, const group_t *x, const uint8_t e[EXPBYTES])
{
  int i,j;

  group_setone(r);
  for(i=EXPBYTES-1;i>=0;i--) {
    for(j=7;j>=0;j--) {
      group_mul(r, r, r);
      if(e[i]>>j & 1) {
        group_mul(r, r, x);
      }
    }
  }
}
```

Square-and-multiply

```
void group_exp(group_t *r, const group_t *x, const uint8_t e[EXPBYTES])
{
  int i,j;

  group_setone(r);
  for(i=EXPBYTES-1;i>=0;i--) {
    for(j=7;j>=0;j--) {
      group_mul(r, r, r);
      if(e[i]>>j & 1) {
        group_mul(r, r, x);
      }
    }
  }
}
```

- Secret branch condition leaks through timing!
- Idea: Always perform multiplication by `x`

Square-and-multiply-always

```
void group_exp(group_t *r, const group_t *x, const uint8_t e[EXPBYTES])
{
  int i,j;
  group_t t;

  group_setone(r);
  for(i=EXPBYTES;i>=0;i--) {
    for(j=7;j>=0;j--) {
      group_mul(r,r,r);
      if((e[i]>>j)&1)
        group_mul(r,r,x);
      else
        group_mul(&t,r,x);
    }
  }
}
```

Square-and-multiply-always

```
void group_exp(group_t *r, const group_t *x, const uint8_t e[EXPBYTES])
{
  int i,j;
  group_t t;

  group_setone(r);
  for(i=EXPBYTES;i>=0;i--) {
    for(j=7;j>=0;j--) {
      group_mul(r,r,r);
      if((e[i]>>j)&1)
        group_mul(r,r,x);
      else
        group_mul(&t,r,x);
    }
  }
}
```

- Compiler may optimize `else` clause away, but can avoid that

Square-and-multiply-always

```
void group_exp(group_t *r, const group_t *x, const uint8_t e[EXPBYTES])
{
  int i,j;
  group_t t;

  group_setone(r);
  for(i=EXPBYTES;i>=0;i--) {
    for(j=7;j>=0;j--) {
      group_mul(r,r,r);
      if((e[i]>>j)&1)
        group_mul(r,r,x);
      else
        group_mul(&t,r,x);
    }
  }
}
```

- Compiler may optimize `else` clause away, but can avoid that
- Still not constant time, reasons:
  - Branch prediction
  - Instruction cache

Eliminating branches

- So, what do we do with code like this?

  if s **then**

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if

Eliminating branches

- So, what do we do with code like this?

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

- Replace by

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

Eliminating branches

- So, what do we do with code like this?

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

- Replace by

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

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

Eliminating branches

- So, what do we do with code like this?

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

- Replace by

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

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

Fixing Square-and-multiply-always

```
void group_exp(group_t *r, const group_t *x, const uint8_t e[EXPBYTES])
{
  int i,j;
  group_t t;

  group_setone(r);
  for(i=EXPBYTES;i>=0;i--) {
    for(j=7;j>=0;j--) {
      group_mul(r,r,r);
      group_mul(&t,r,x);
      group_cmov(r, &t, (e[i]>>j)&1);
    }
  }
}
```

cmov

```
/* decision bit b has to be either 0 or 1 */
void group_cmov(group_t *r, const group_t *a, uint32_t b)
{
  group_t t;

  b = -b; /* Now b is either 0 or 0xffffffff */
  t = (*r ^ *a) & b;
  *r ^= t;
}
```

Faster exponentiation

- Idea: precompute some multiples of x
- Process multiple bits in parallel
- "Fixed-window method"
- Let's process chunks of 4 bits of the exponent

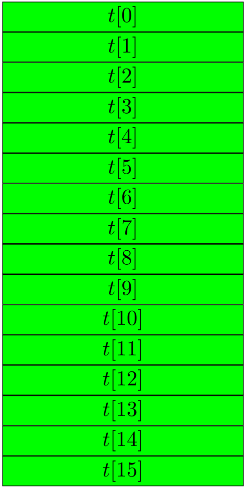
Fixed-window exponentiation

```
void group_exp(group_t *r, const group_t *x, const uint8_t e[EXPBYTES])
{
  int i,j;
  group_t t[16];
  group_setone(&t[0]);
  t[1] = *x;
  for(i=2;i<16;i++)
    group_mul(&t[i], &t[i-1], x);

  group_setone(r);
  for(i=EXPBYTES;i>=0;i--) {

    for(j=0;j<4;j++)
      group_mul(r,r,r);
    group_mul(r,r,&t[e[i]>>4]);

    for(j=0;j<4;j++)
      group_mul(r,r,r);
    group_mul(r,r,&t[e[i]&0xf]);
  }
}
```

Problem

```
table[secret]
```

Cache-timing attacks



- Crypto and the attacker's program run on the same CPU
- Table is in cache
- Simplification: each table entry takes exactly one cache line

Cache-timing attacks

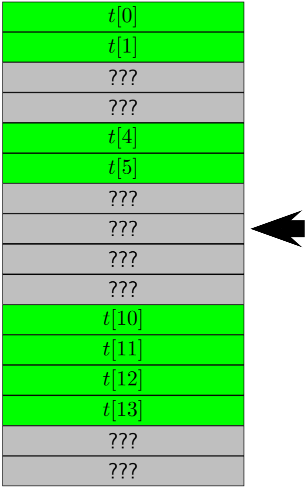
- Crypto and the attacker's program run on the same CPU
- Table is in cache
- Simplification: each table entry takes exactly one cache line
- The attacker's program replaces some cache lines

Cache-timing attacks

$t[0]$
$t[1]$
???
???
$t[4]$
$t[5]$
???
???
???
???
$t[10]$
$t[11]$
$t[12]$
$t[13]$
???
???

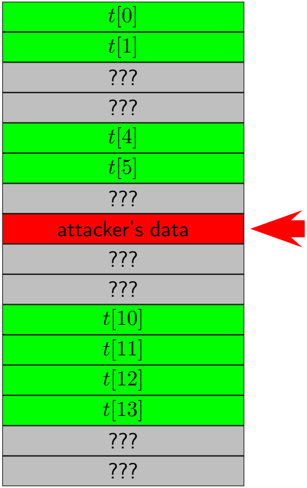
- Crypto and the attacker's program run on the same CPU
- Table is in cache
- Simplification: each table entry takes exactly one cache line
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again

Cache-timing attacks



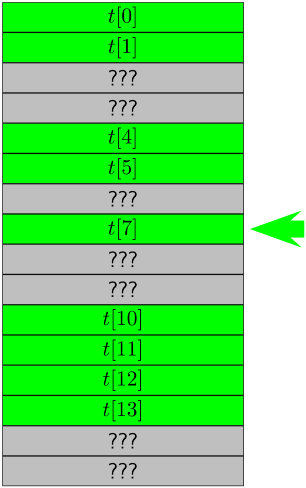
- Crypto and the attacker's program run on the same CPU
- Table is in cache
- Simplification: each table entry takes exactly one cache line
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads his data:

Cache-timing attacks



- Crypto and the attacker's program run on the same CPU
- Table is in cache
- Simplification: each table entry takes exactly one cache line
- 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)

Cache-timing attacks



- Crypto and the attacker's program run on the same CPU
- Table is in cache
- Simplification: each table entry takes exactly one cache line
- 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)

“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe

“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe... or are they?

“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe... or are they?
- Bernstein, 2005: *“Does this guarantee constant-time S-box lookups? No!”*

“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe… or are they?
- Bernstein, 2005: *“Does this guarantee constant-time S-box lookups? No!”*
- Osvik, Shamir, Tromer, 2006: *“This is insufficient on processors which leak low address bits”*

“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe… or are they?
- Bernstein, 2005: *“Does this guarantee constant-time S-box lookups? No!”*
- Osvik, Shamir, Tromer, 2006: *“This is insufficient on processors which leak low address bits”*
- Reasons:
  - Cache-bank conflicts
  - Failed store-to-load forwarding
  - …

“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe... or are they?
- Bernstein, 2005: *“Does this guarantee constant-time S-box lookups? No!”*
- Osvik, Shamir, Tromer, 2006: *“This is insufficient on processors which leak low address bits”*
- Reasons:
  - Cache-bank conflicts
  - Failed store-to-load forwarding
  - ...
- OpenSSL is using it in `BN_mod_exp_mont_consttime`

"Countermeasure"

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe… or are they?
- Bernstein, 2005: *"Does this guarantee constant-time S-box lookups? No!"*
- Osvik, Shamir, Tromer, 2006: *"This is insufficient on processors which leak low address bits"*
- Reasons:
  - Cache-bank conflicts
  - Failed store-to-load forwarding
  - …
- OpenSSL is using it in `BN_mod_exp_mont_consttime`
- Brickell (Intel), 2011: yeah, it's fine as a countermeasure

"Countermeasure"

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe... or are they?
- Bernstein, 2005: *"Does this guarantee constant-time S-box lookups? No!"*
- Osvik, Shamir, Tromer, 2006: *"This is insufficient on processors which leak low address bits"*
- Reasons:
  - Cache-bank conflicts
  - Failed store-to-load forwarding
  - ...
- OpenSSL is using it in `BN_mod_exp_mont_consttime`
- Brickell (Intel), 2011: yeah, it's fine as a countermeasure
- Bernstein, Schwabe, 2013: Demonstrate timing variability for access within one cache line

“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe… or are they?
- Bernstein, 2005: *“Does this guarantee constant-time S-box lookups? No!”*
- Osvik, Shamir, Tromer, 2006: *“This is insufficient on processors which leak low address bits”*
- Reasons:
  - Cache-bank conflicts
  - Failed store-to-load forwarding
  - …
- OpenSSL is using it in `BN_mod_exp_mont_consttime`
- Brickell (Intel), 2011: yeah, it’s fine as a countermeasure
- Bernstein, Schwabe, 2013: Demonstrate timing variability for access within one cache line
- TODO: Real attack against, e.g., OpenSSL

Fixing fixed-window exponentiation

```
void group_exp(group_t *r, const group_t *x, const uint8_t e[EXPBYTES])
{
  int i,j; group_t t[16],d;
  group_setone(&t[0]);
  t[1] = *x;
  for(i=2;i<16;i++)
    group_mul(&t[i], &t[i-1], x);

  group_setone(r);
  for(i=EXPBYTES;i>=0;i--) {

    for(j=0;j<4;j++)
      group_mul(r,r,r);
    lookup(&d,t,e[i]>>4); group_mul(r,r,&d);

    for(j=0;j<4;j++)
      group_mul(r,r,r);
    lookup(&d,t,e[i]&0xf); group_mul(r,r,&d);
  }
}
```

Lookup

```
void lookup(group_t *r, const group_t *t, uint32_t pos)
{
  uint32_t i;
  group_t d;
  *r = t[0];
  for(i=1;i<16;i++)
  {
    d = t[i];
    group_cmov(r,&d, i==pos);
  }
}
```

Lookup

```
void lookup(group_t *r, const group_t *t, uint32_t pos)
{
  uint32_t i;
  group_t d;
  *r = t[0];
  for(i=1;i<16;i++)
  {
    d = t[i];
    group_cmov(r,&d, i==pos);
  }
}
```

Does this leak? Depends on how the compiler handles `i==pos`

Fixing lookup

```
void lookup(group_t *r, const group_t *t, uint32_t pos)
{
  uint32_t i;
  group_t d;
  *r = t[0];
  for(i=1;i<16;i++)
  {
    d = t[i];
    group_cmov(r,&d, uint_iseq(i,pos));
  }
}
```

Constant-time comparison

```
int uint_iseq(unsigned int a, unsigned int b)
{
  uint64_t t = a ^ b;
  t = -t; /* Assuming 2's complement */
  t >>= 63;
  return 1-t;
}
```

Is that all?

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?

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
- Huge synergies with vectorizing algorithms!

Is that all?

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
- Huge synergies with vectorizing algorithms!

"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?

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
- Huge synergies with vectorizing algorithms!

"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

Are we using any *DIV?

Are we using any *DIV?

Suspicious line of code

```
*r = ((uint64_t) *x * *y) % 0x7FFFFFFF;
```

Are we using any *DIV?

Suspicious line of code

```
*r = ((uint64_t) *x * *y) % 0x7FFFFFFF;
```

- Compiler actually optimizes for fixed modulus
- No *DIV in the disassembly

Are we using any *DIV?

Suspicious line of code

```
*r = ((uint64_t) *x * *y) % 0x7FFFFFFF;
```

- Compiler actually optimizes for fixed modulus
- No *DIV in the disassembly
- Generally better to avoid / and % with secret arguments:
  - Avoid issues with different compilers and options
  - Also simplifies static analysis on source level

Fixing our group multiplication

```
static void group_mul(group_t *r, const group_t *x, const group_t *y)
{
  uint64_t t,c;
  t = (uint64_t) *x * *y;
  c = t >> 31;
  *r = t & 0x7FFFFFFF;
  *r += c;
  c = *r >> 31;
  *r &= 0x7FFFFFFF;
  *r += c;
}
```

Summary

- Think about performance in terms of architectural bottlenecks

Summary

- Think about performance in terms of architectural bottlenecks
- Vectorization rocks
- Vectorization requires re-thinking algorithms

Summary

- Think about performance in terms of architectural bottlenecks
- Vectorization rocks
- Vectorization requires re-thinking algorithms
- Systematic timing-attack protection is doable, but requires care
- Timing-attack protection requires re-thinking algorithms

Summary

- Think about performance in terms of architectural bottlenecks
- Vectorization rocks
- Vectorization requires re-thinking algorithms
- Systematic timing-attack protection is doable, but requires care
- Timing-attack protection requires re-thinking algorithms
- Huge synergies between timing-attack protection and vectorization

Summary

- Think about performance in terms of architectural bottlenecks
- Vectorization rocks
- Vectorization requires re-thinking algorithms
- Systematic timing-attack protection is doable, but requires care
- Timing-attack protection requires re-thinking algorithms
- Huge synergies between timing-attack protection and vectorization

Design crypto as secret-branch-free, secret-lookup-free programs