MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

High-assurance crypto – Part I

Peter Schwabe

January 30, 2023

Max-Planck Institute for Security and Privacy

- Founded in 2019
- Currently:
  - 2 directors
  - 7 research group leaders
  - ≈40 postdocs and Ph.D. students
- Long-term plan
  - 6 directors
  - 12 research group leaders
  - 200+ scientific staff

MPI-SP?

MPI-SP?

Crypto software

Web connections are secured by TLS

- Diffie-Hellman key exchange
- Digital Signatures
- Symmetric encryption
- Message authentication
- Hash functions

This is done using software libraries

- Client-side (browser):
  - BoringSSL (Chrome)
  - NSS (Firefox)
- Server-side:
  - OpenSSL
  - BoringSSL
  - SChannel

What properties would you expect from crypto software?

3 properties

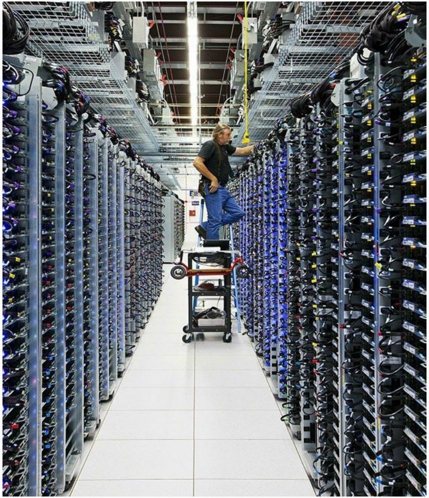
3. Correctness
   - Functionally correct
   - Memory safety
   - Thread safety
   - Termination

2. Security
   - Don't leak secrets
   - "Constant-time"
   - Resist Spectre attacks
   - Resist Power/EM attacks
   - Fault protection
   - Easy-to-use APIs

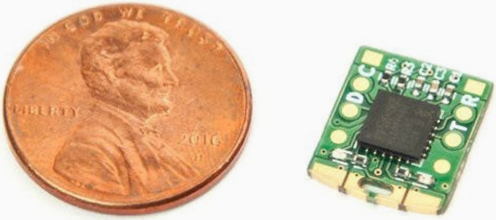
1. Efficiency
   - Speed (clock cycles)
   - RAM usage
   - Binary size
   - Energy consumption

Part I: Efficiency

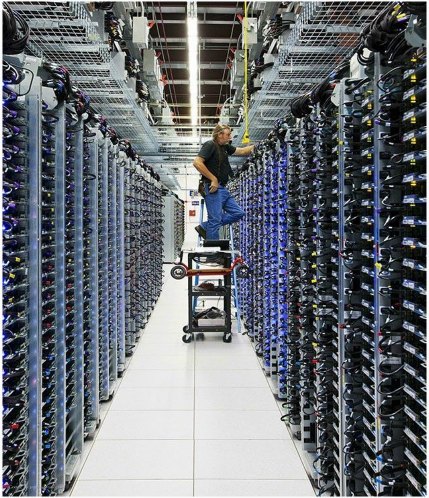
Why care about speed/efficiency?



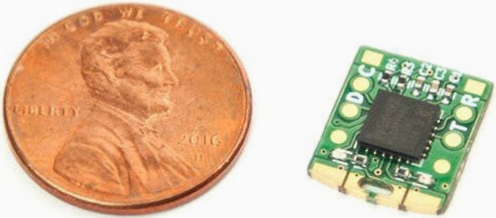
- 10% performance difference matters!
  - Reduce cost for busy servers
  - Fit into constrained devices



Why care about speed/efficiency?



- 10% performance difference matters!
  - Reduce cost for busy servers
  - Fit into constrained devices
- Low-level, heavy optimization
- **Real-world cost model** for algorithms!



What do we mean by “speed”?

Easy answer:

Time it takes between beginning and end of one crypto computation

What do we mean by “speed”?

Easy answer:

Time it takes between beginning and end of one crypto computation

. . . **or how about:**

Amount of crypto computations we can do per second or minute

What do we mean by “speed”?

Easy answer:

Time it takes between beginning and end of one crypto computation

. . . **or how about:**

Amount of crypto computations we can do per second or minute

- First definition is **latency**
- Second definition is **througput**

What do we mean by "speed"?

Easy answer:

Time it takes between beginning and end of one crypto computation

. . . **or how about:**

Amount of crypto computations we can do per second or minute

- First definition is **latency**
- Second definition is **througput**
- Careful: often n computations are faster than $n\times$ one computation

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

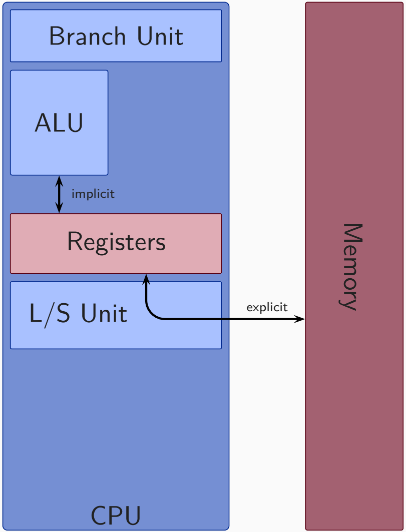
Benchmarking pitfalls

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

Computers and computer programs



- 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

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

```
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

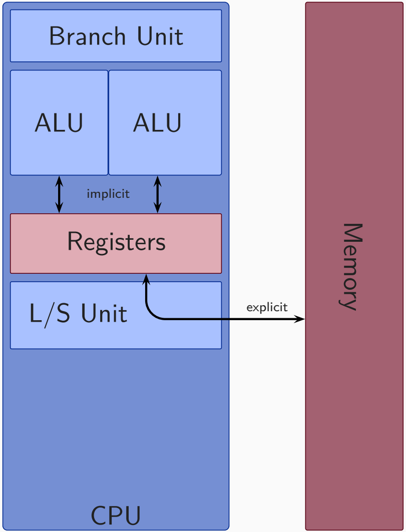
Instruction throughput and latency

- 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



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: $\geq$ 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: $\geq$ 1000 cycles
- Need at least 999 addition instructions: $\geq$ 500 cycles
- At least 1999 instructions: $\geq$ 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: $\geq$ 1000 cycles
- Need at least 999 addition instructions: $\geq$ 500 cycles
- At least 1999 instructions: $\geq$ 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

```
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

```
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: $\approx$ 3000 cycles

Making the program fast

```
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: $\approx$ 3000 cycles
- Better, but still too slow

Making the program fast

```
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

```
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: $\approx$ 2000 cycles

Making the program fast

```
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

“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. . .

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...

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...

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...

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
  - $8\times$ 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$\times$ 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$\times$ 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
  - $8\times$ 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 $8\times$ the work**
- Situation on other architectures/microarchitectures is similar
- Reason: cheap way to increase arithmetic throughput (less decoding, address computation, etc.)

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 $f = f_0 + f_1 x + f_2 x^2 + f_3 x^3$ and $g = g_0 + g_1 x + g_2 x^2 + g_3 x^3$:

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

Data shuffeling

- Consider multiplication of 4-coefficient polynomials $f = f_0 + f_1 x + f_2 x^2 + f_3 x^3$ and $g = g_0 + g_1 x + g_2 x^2 + g_3 x^3$:

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

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

Vectorization problems III

Data shuffeling

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

- 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 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

$$r_0 = f_0 g_0$$
$$r_1 = f_0 g_1 + f_1 g_0$$
$$r_2 = f_0 g_2 + f_1 g_1 + f_2 g_0$$
$$r_3 = f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0$$
$$r_4 = f_1 g_3 + f_2 g_2 + f_3 g_1$$
$$r_5 = f_2 g_3 + f_3 g_2$$
$$r_6 = f_3 g_3$$

- 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?
- 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

Summary

- Crypto optimization commonly on assembly level
- Think about algorithms in terms of machine instructions
  - Understand cycle lower bound
  - Carefully choose and schedule instructions
  - Take care of register allocation
- Vectorization is often key to high performance

MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

High-assurance crypto – Part II: Security

Peter Schwabe

January 30, 2023

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 – we must not leak secret data to an attacker!

Hello World – with a secret

```
#include <sys/random.h>
#include <unistd.h>
#include <stdio.h>

int main(void)
{
  unsigned char secret;
  getrandom(&secret, sizeof(secret), 0);
  secret &= 1;

  if(secret) sleep(3);

  printf("Hello World!\n");
}
```

Timing leakage part I

- Consider the following piece of code:

  if s **then**

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if

Timing leakage part I

- Consider the following piece of code:

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

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

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**

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if

Eliminating branches

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

  if s **then**
  $r \leftarrow A$
  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**

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if

- Replace by

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

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

Eliminating branches

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

  if s **then**

  $r \leftarrow A$

  else

  $r \leftarrow B$

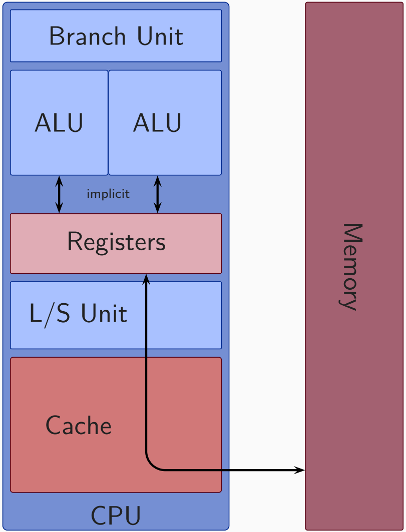
  end if

- Replace by

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

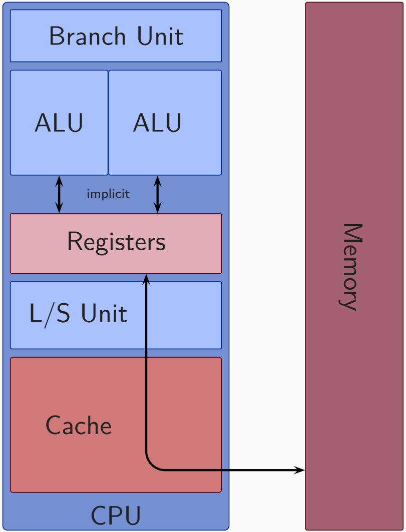
- Can expand s to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication
- For very fast A and B this can even be faster

Cached memory access



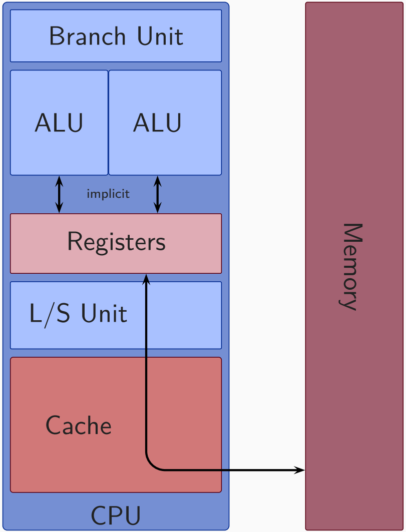
- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data

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



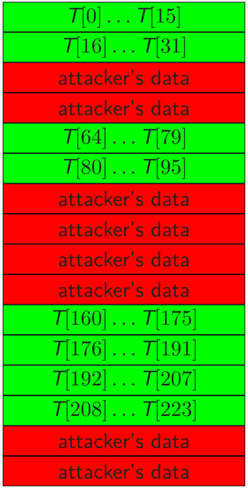
- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data
- A load from memory places data also in the cache
- Data remains in cache until it's replaced by other data
- Loading data is fast if data is in the cache (**cache hit**)
- Loading data is slow if data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

Timing leakage part II

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

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

Timing leakage part II



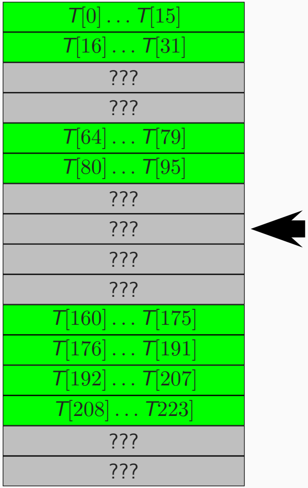
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines

Timing leakage part II

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

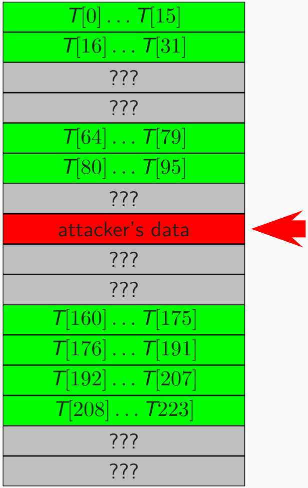
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again

Timing leakage part II



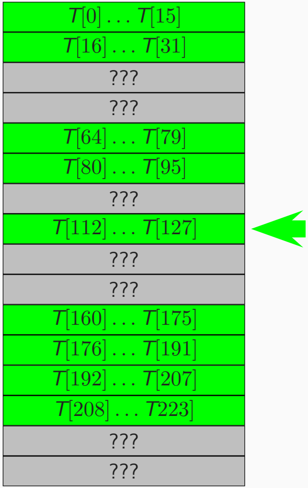
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads his data:

Timing leakage part II



- 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 as long as input size is public (or at least upper bounded)
- 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 as long as input size is public (or at least upper bounded)
- 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 as long as input size is public (or at least upper bounded)
- Cost highly depends on the algorithm

"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 as long as input size is public (or at least upper bounded)
- Cost highly depends on the algorithm

"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

Is "constant-time" enough?

- **Local** attacker may see much more than just timing (see Thursday lectures):
  - Power consumption
  - Electromagnetic radiation
  - Acoustic emissions
  - ...

Is “constant-time” enough?

- **Local** attacker may see much more than just timing (see Thursday lectures):
  - Power consumption
  - Electromagnetic radiation
  - Acoustic emissions
  - ...
- Sometimes constant-time also too weak against **remote** attackers (see Friday lecture):
  - Spectre and Meltdown
  - Hertzbleed
  - ...

Is "constant-time" enough?

- **Local** attacker may see much more than just timing (see Thursday lectures):
  - Power consumption
  - Electromagnetic radiation
  - Acoustic emissions
  - ...
- Sometimes constant-time also too weak against **remote** attackers (see Friday lecture):
  - Spectre and Meltdown
  - Hertzbleed
  - ...
- Constant-time remains important **base-line defense**

Summary

- Crypto software must avoid timing leaks
- Mostly two rules:
  1. **Never branch depending on secret data**
  2. **Never access memory at secret locations**
- Additionally: avoid variable-time arithmetic instructions

Summary

- Crypto software must avoid timing leaks
- Mostly two rules:
  1. **Never branch depending on secret data**
  2. **Never access memory at secret locations**
- Additionally: avoid variable-time arithmetic instructions
- This is **necessary base-line defense** for essentially all crypto software
- Does not protect against physical side-channel attacks
- Helps, but does not protect against advanced microarchitectural attacks

High-assurance crypto – Part III: Jasmin

Peter Schwabe

January 30, 2023

So, where are we?

Efficiency/Speed ✓

Security ?

Correctness ?

Security?

We know what to do

- No secret-dependent branches
- No secret-dependent memory indexing
- No variable-time arithmetic on secrets

. . . **but how do we make sure we get it right?**

Correctness?

"Are you actually sure that your software is correct?"

—prof. Gerhard Woeginger, Jan. 24, 2011.

#epicfail

```
mulq  crypto_sign_ed25519_amd64_64_38
add  %rax,%r13
adc %rdx,%r14
adc $0,%r14
mov  %r9,%rax
mulq  crypto_sign_ed25519_amd64_64_38
add  %rax,%r14
adc %rdx,%r15
adc $0,%r15
mov  %r10,%rax
mulq  crypto_sign_ed25519_amd64_64_38
add  %rax,%r15
adc %rdx,%rbx
adc $0,%rbx
mov  %r11,%rax
mulq  crypto_sign_ed25519_amd64_64_38
add  %rax,%rbx
mov  $0,%rsi
adc %rdx,%rsi
```

- Code snippet is from > 8000 lines of assembly
- Crypto **always** has more possible inputs than we can exhaustively test
- Some bugs are triggered with very low probability
- Testing won't catch these bugs
- Audits might, but this requires expert knowledge!

High-assurance crypto

Traditional approach is great at producing very efficient software that is neither (guaranteed to be) correct nor (guaranteed to be) secure.

High-assurance crypto

Traditional approach is great at producing very efficient software that is neither (guaranteed to be) correct nor (guaranteed to be) secure.

- Idea: Use tools/techniques from formal methods to prove
  - functional correctness (including e.g., safety);
  - certain implementation security properties; (and
  - cryptographic security through reductions)

High-assurance crypto

Traditional approach is great at producing very efficient software that is neither (guaranteed to be) correct nor (guaranteed to be) secure.

- Idea: Use tools/techniques from formal methods to prove
  - functional correctness (including e.g., safety);
  - certain implementation security properties; (and
  - cryptographic security through reductions)
- Crypto software is a special here in multiple ways:
  - Usually fairly little code (+)
  - Has precise formal specification (+)
  - Inherently security-critical (+)

High-assurance crypto

Traditional approach is great at producing very efficient software that is neither (guaranteed to be) correct nor (guaranteed to be) secure.

- Idea: Use tools/techniques from formal methods to prove
  - functional correctness (including e.g., safety);
  - certain implementation security properties; (and
  - cryptographic security through reductions)
- Crypto software is a special here in multiple ways:
  - Usually fairly little code (+)
  - Has precise formal specification (+)
  - Inherently security-critical (+)
  - Highly performance critical (−)

High-assurance crypto

Traditional approach is great at producing very efficient software that is neither (guaranteed to be) correct nor (guaranteed to be) secure.

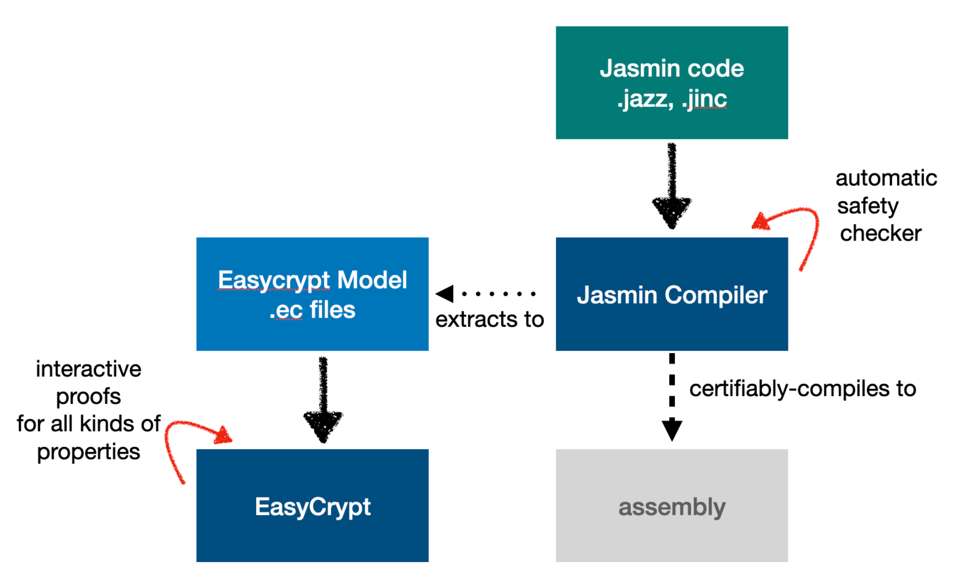
- Idea: Use tools/techniques from formal methods to prove
  - functional correctness (including e.g., safety);
  - certain implementation security properties; (and
  - cryptographic security through reductions)
- Crypto software is a special here in multiple ways:
  - Usually fairly little code (+)
  - Has precise formal specification (+)
  - Inherently security-critical (+)
  - Highly performance critical (−)

We want formal guarantees without giving up on performance.

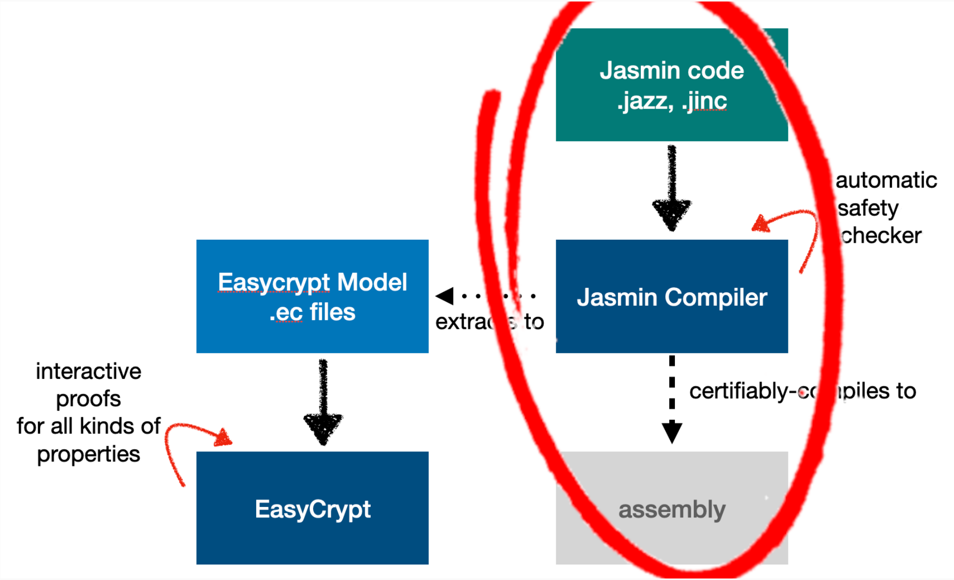
Formosa Crypto

- Effort to formally verify crypto
- Currently three main projects:
  - EasyCrypt proof assistant
  - jasmin programming language
  - libjade (PQ-)crpyto library
- Core community of $\approx$ 30–40 people
- Discussion forum with $>$100 people

The toolchain and workflow



The toolchain and workflow



Jasmin – assembly in your head

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, Pierre-Yves Strub: *Jasmin: High-Assurance and High-Speed Cryptography*. ACM CCS 2017

- Language with "C-like" syntax
- Programming in jasmin is much closer to assembly:
  - Generally: 1 line in jasmin → 1 line in asm
  - A few exceptions, but highly predictable
  - Compiler does not schedule code
  - Compiler does not spill registers

Jasmin – assembly in your head

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, Pierre-Yves Strub: *Jasmin: High-Assurance and High-Speed Cryptography*. ACM CCS 2017

- Language with “C-like” syntax
- Programming in jasmin is much closer to assembly:
  - Generally: 1 line in jasmin $\rightarrow$ 1 line in asm
  - A few exceptions, but highly predictable
  - Compiler does not schedule code
  - Compiler does not spill registers
- Compiler is formally proven to preserve semantics
- Compiler is formally proven to preserve constant-time property

Jasmin – assembly in your head

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, Pierre-Yves Strub: *Jasmin: High-Assurance and High-Speed Cryptography*. ACM CCS 2017

- Language with "C-like" syntax
- Programming in jasmin is much closer to assembly:
  - Generally: 1 line in jasmin $\rightarrow$ 1 line in asm
  - A few exceptions, but highly predictable
  - Compiler does not schedule code
  - Compiler does not spill registers
- Compiler is formally proven to preserve semantics
- Compiler is formally proven to preserve constant-time property
- Many new features since 2017 paper!

C code

```c
#include <stdio.h>

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

jasmin code

Jasmin – first steps

C code

```c
#include <stdio.h>

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

jasmin code

- We don't implement main in jasmin
- We don't have I/O in jasmin

Jasmin – first steps

```
export fn add42(reg u64 x) -> reg u64 {
  reg u64 r;
  r = x;
  r += 42;
  return r;
}
```

Jasmin – first steps

https://cryptojedi.org/programming/jasmin.shtml

Registers, stack, and arrays

- For each variable you need to decide if it is
  - living in a register: `reg`,
  - living on the stack: `stack`, or
  - replaced by immediates during compilation: `inline int`
- Integer types are called `u64`, `u32`, etc.
- Jasmin supports arrays of `reg` and `stack` variables:
  - `reg u32[10] a;`
  - `stack u64[100] b;`
- Arrays have **fixed** length
- Jasmin supports sub-arrays with fixed offsets and lengths, e.g. `b[16:32]` is the subarray of length 32 starting at index 16

Loops and conditionals

- Conditionals (`if`, `else`) like in C

Loops and conditionals

- Conditionals (`if`, `else`) like in C
- Two kinds of loops: `for` and `while`

Loops and conditionals

- Conditionals (`if`, `else`) like in C
- Two kinds of loops: `for` and `while`
- `for` loops are automatically unrolled
- `for` iterate over an `inline int`

Loops and conditionals

- Conditionals (`if`, `else`) like in C
- Two kinds of loops: `for` and `while`
- `for` loops are automatically unrolled
- `for` iterate over an `inline int`
- `while` loops are *real* loops with branch

Three kinds of “functions”

`export` functions

- Entry points into jasmin-generated code
- Need at least one `export` function in a jasmin program
- Follows (Linux) AMD64 C function-call ABI

Three kinds of “functions”

`export` functions

- Entry points into jasmin-generated code
- Need at least one `export` function in a jasmin program
- Follows (Linux) AMD64 C function-call ABI

`inline functions`

- Historically only non-`export` functions
- Can receive stack-array arguments

Three kinds of “functions”

`export` functions

- Entry points into jasmin-generated code
- Need at least one `export` function in a jasmin program
- Follows (Linux) AMD64 C function-call ABI

`inline functions`

- Historically only non-`export` functions
- Can receive stack-array arguments

“Regular” functions

- Array arguments passed through `reg ptr`
- `reg ptr` cannot be modifed through arithmetic
- No fixed function-call ABI (compilation has global view)
- Stack pointer decreased **by caller**

Jasmin errors

- Easy case: syntax errors

Jasmin errors

- Easy case: syntax errors
- Slighly tougher: missing casts, see, e.g.,

```
t0 = a.[u256 (int)(32 *64u i)];
```

Jasmin errors

- Easy case: syntax errors
- Slighly tougher: missing casts, see, e.g.,

  `t0 = a.[u256 (int)(32 *64u i)];`
- Most time-consuming to debug: register-allocation errors
- Example 1: constraints not satisfiable

```
export fn add42(reg u64 x) -> reg u64 {
  x += 42;
  return x;
}
```

Jasmin errors

- Easy case: syntax errors
- Slighly tougher: missing casts, see, e.g.,

  `t0 = a.[u256 (int)(32 *64u i)];`
- Most time-consuming to debug: register-allocation errors
- Example 1: constraints not satisfiable

```
export fn add42(reg u64 x) -> reg u64 {
  x += 42;
  return x;
}
```

- Example 2: Running out of registers

```
"kem.jazz", line 14 (1) to line 27 (1):
compilation error:
register allocation: variable shkp.3135 must be allocated to conflicting register RSI { RSI.83 }
make: *** [../../../../../Makefile.common:73: kem.s] Error 1
```

- Register allocation is global
  - Changes at one place may cause allocation to fail somewhere else
  - Error messages not super-helpful

Vectorization in jasmin

- Jasmin supports 128-bit XMM and 256-bit YMM registers: u128 and u256
- Operations through "intrinsics", e.g.,

```
reg u256 t0, t1;

for i = 0 to VLEN/8 {
  t0 = a.[u256 (int)(32 *64u i)];
  t1 = b.[u256 (int)(32 *64u i)];
  t0 = #VPADD_8u32(t0, t1);
  r.[u256 (int)(32 *64u i)] = t0;
}
```

Some current limitations

AMD64 only

- Full functionality only for AMD64 assembly
- ARMv7M (Cortex-M4) support in development branch
- Future directions: ARMv8, RISC-V, OpenTitan

Some current limitations

AMD64 only

- Full functionality only for AMD64 assembly
- ARMv7M (Cortex-M4) support in development branch
- Future directions: ARMv8, RISC-V, OpenTitan

No “slice” arguments

- Arrays have to have fixed length also in function arguments
- Separate function for each input length, e.g.

```
fn _ishake256_128_33(reg ptr u8[128] out, reg const ptr u8[33] in) -> stack u8[128]
```

- **Not** an issue for variable-length arguments to `export` functions

Some current limitations

No register-indexed subarrays

This works

```
stack u16[768] a;
inline int i;
for i=0 to 3
{
  a[i*256:256] = foo(a[i*256:256]);
}
```

This does not

```
stack u16[768] a;
reg u64 i;
i = 0;
while(i < 3)
{
  a[i*256:256] = foo(a[i*256:256]);
  i += 1;
}
```

Some current limitations

No typed export functions

- Inputs to `export` functions are of type `reg u64`
- Output is also a `reg u64`
- No argument passing over the stack
- No more than 6 arguments
- Distinguish between pointers and data only by usage/context

Memory and thread safety

- Jasmin does not support dynamic memory allocation
- All memory locations are either
  - external memory accessible through `export` function pointer arguments, or
  - allocated on the stack

Memory and thread safety

- Jasmin does not support dynamic memory allocation
- All memory locations are either
  - external memory accessible through `export` function pointer arguments, or
  - allocated on the stack
- Checking memory safety is separate compiler pass

```
jasminc -checksafety INPUT.jazz
```

- This typically takes a while to finish

Memory and thread safety

- Jasmin does not support dynamic memory allocation
- All memory locations are either
  - external memory accessible through `export` function pointer arguments, or
  - allocated on the stack
- Checking memory safety is separate compiler pass

  `jasminc -checksafety INPUT.jazz`
- This typically takes a while to finish
- Jasmin does not have global variables
- Thread safe (except if external memory is shared)

So, again, where are we?

Efficiency

- Some limitations compared to assembly for memory safety
- No limitations that (majorly) impact performance

Security

- ???

Correctness

- Functional correctness through EasyCrypt proofs (tomorrow)
- Thread and **memory safety** guaranteed by jasmin
- Still need to check that EC specification is correct!
- Could be addressed by machine-readable standards

Did we get it right?

Option 1: Auditing

"Originally, me, a glass of bourbon, and gdb were a good trio. But that got old pretty quick. (The manual analysis part – not the whiskey.)"

—Survey response in https://ia.cr./2021/1650

Did we get it right?

Option 1: Auditing

"Originally, me, a glass of bourbon, and gdb were a good trio. But that got old pretty quick. (The manual analysis part – not the whiskey.)"

—Survey response in https://ia.cr./2021/1650

Option 2: Check/verify

- Implement, use tool to check "constant-time" property
- Problems in practice:
  - Some tools not sound
  - Some tools not on binary/asm level
  - Some tools not usable

} **Fairly high on my whishlist**. . .

Did we get it right?

Option 1: Auditing

"Originally, me, a glass of bourbon, and gdb were a good trio. But that got old pretty quick. (The manual analysis part – not the whiskey.)"

—Survey response in https://ia.cr./2021/1650

Option 2: Check/verify

- Implement, use tool to check "constant-time" property
- Problems in practice:
  - Some tools not sound
  - Some tools not on binary/asm level
  - Some tools not usable

} **Fairly high on my whishlist**...

Option 3: Avoid variable-time code

- Prevent leaking patterns on source level
- Prove that compilation doesn't introduce leakage

Secret types

- Enforce "constant-time" on jasmin source level
- Every piece of data is either `secret` or `public`
- Flow of secret information is traced by type system

  "Any operation with a secret input produces a secret output"

Secret types

- Enforce "constant-time" on jasmin source level
- Every piece of data is either `secret` or `public`
- Flow of secret information is traced by type system

  "Any operation with a secret input produces a secret output"
- Branch conditions and memory indices need to be `public`

Secret types

- Enforce “constant-time” on jasmin source level
- Every piece of data is either `secret` or `public`
- Flow of secret information is traced by type system

  “Any operation with a secret input produces a secret output”
- Branch conditions and memory indices need to be `public`
- In principle can do this also in, e.g., Rust (`secret_integers` crate)

Secret types

- Enforce "constant-time" on jasmin source level
- Every piece of data is either `secret` or `public`
- Flow of secret information is traced by type system

  "Any operation with a secret input produces a secret output"
- Branch conditions and memory indices need to be `public`
- In principle can do this also in, e.g., Rust (`secret_integers` crate)
- **Jasmin compiler is verified to preserve constant-time!**

Gilles Barthe, Benjamin Gregoire, Vincent Laporte, and Swarn Priya. *Structured Leakage and Applications to Cryptographic Constant-Time and Cost*. CCS 2021. https://eprint.iacr.org/2021/650

Secret types

- Enforce "constant-time" on jasmin source level
- Every piece of data is either `secret` or `public`
- Flow of secret information is traced by type system

  "Any operation with a secret input produces a secret output"
- Branch conditions and memory indices need to be `public`
- In principle can do this also in, e.g., Rust (`secret_integers` crate)
- **Jasmin compiler is verified to preserve constant-time!**
- Explicit `#declassify` primitive to move from `secret` to `public`
- `#declassify` creates a proof obligation!

Gilles Barthe, Benjamin Gregoire, Vincent Laporte, and Swarn Priya. *Structured Leakage and Applications to Cryptographic Constant-Time and Cost*. CCS 2021. https://eprint.iacr.org/2021/650

Summary

- Jasmin is an “easy way to program on assembly level”
  - Easy way to implement conditionals, loops, functions
  - More readable syntax
  - Register allocation
- Guarantees of memory safety, thread safety
- Functional correctness proofs in EasyCrypt
- Constant-time ensured through type system

Jasmin exercise

1. Download https://cryptojedi.org/bkk-school-exercise1.tar.bz2
2. Check that you can build the code:

```
tar xjvf bkk-school-exercise1.tar.bz2
cd bkk-school-exercise1
make
./test
```

3. Make sure that `./test` no longer prints an error message:
   - Implement function `poly1305_verify_jasmin` in `jasmin/poly1305.jazz`
   - See function `poly1305_verify_c` in `c/poly1305.c`
4. Make your implementation pass constant-time check:
   - Check with `jasminc -checkCT jasmin/poly1305.jazz`
   - Hint: The C code is *not* constant time!

Jasmin exercise – Part II

1. Download https://cryptojedi.org/bkk-school-excercise2.tar.bz2
2. Check that you can build the code:

```
tar xjvf bkk-school-exercise2.tar.bz2
cd bkk-school-exercise2
make
./test
```

3. Make sure that `./test` no longer prints an error message:
   - Implement function `gimli_jasmin` in `jasmin/gimli.jazz`
   - See function `gimli_c` in `c/gimli.c`
4. Bonus: Make your Gimli implemetation faster
   - Use `./speed` to see cycle counts