Software implementation of Post-Quantum Cryptography

Peter Schwabe

Radboud University Nijmegen, The Netherlands

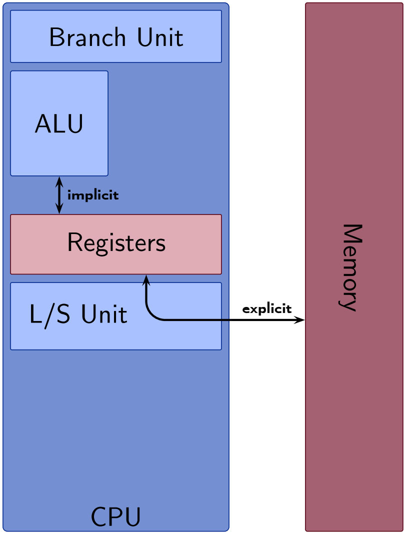
October 20, 2013

ASCrypto 2013, Florianópolis, Brazil

Part I

Optimizing cryptographic software with vector instructions

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

Throughput and latency

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

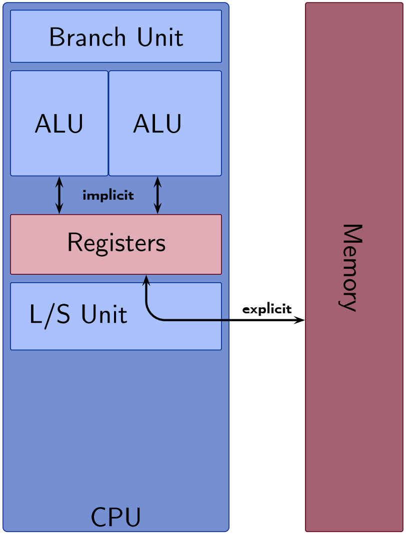
Throughput and latency

- 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

Throughput and latency

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

An example computer
Still highly simplified



Latencies and throughputs

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

Adding up 1000 integers on this computer

- Need at least 1000 load instructions: ≥ 1000 cycles

Latencies and throughputs

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

Adding up 1000 integers on this computer

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

Latencies and throughputs

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

Adding up 1000 integers on this computer

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

Latencies and throughputs

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

Adding up 1000 integers on this computer

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

Latencies and throughputs

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

How about our program?

```
int32 result
int32 tmp
int32 ctr

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

How about our program?

```
int32 result
int32 tmp
int32 ctr

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

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

How about our program?

```
int32 result
int32 tmp
int32 ctr

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

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

Making the program fast

Step 1 – Unrolling

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

...

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

- Remove all the loop control: *unrolling*

Making the program fast

Step 1 – Unrolling

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

...

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

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

Making the program fast
Step 1 – Unrolling

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

...

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

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

Making the program fast
Step 2 – Instruction Scheduling

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

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

...

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

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

Making the program fast

Step 2 – Instruction Scheduling

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

...

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

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

Making the program fast

Step 3 – More Instruction Scheduling (two accumulators)

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

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

...

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

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

Summary of what we did

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

Summary of what we did

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

Summary of what we did

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

Summary of what we did

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

Summary of what we did

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

Summary of what we did

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

Summary of what we did

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

Summary of what we did

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

Architectures and microarchitectures

What instructions and how many registers do we have?

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

Architectures and microarchitectures

What instructions and how many registers do we have?

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

What determines latencies etc?

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

Out-of-order execution

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

Out-of-order execution

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

Out-of-order execution

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

Out-of-order execution

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

Out-of-order execution

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

Optimizing Crypto vs. optimizing “something”

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

Optimizing Crypto vs. optimizing "something"

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

Optimizing Crypto vs. optimizing “something”

- So far there was nothing crypto-specific in this talk
- Is optimizing crypto the same as optimizing any other software?
- No. Cryptographic software deals with secret data (keys)
- Information about secret data must not leak through side channels

Optimizing Crypto vs. optimizing "something"

- So far there was nothing crypto-specific in this talk
- Is optimizing crypto the same as optimizing any other software?
- No. Cryptographic software deals with secret data (keys)
- Information about secret data must not leak through side channels
- Most critical for software implementations on "large" CPUs: software must take constant time (independent of secret data)

Timing leakage part I

- Consider the following piece of code:

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

Timing leakage part I

- Consider the following piece of code:

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

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

Timing leakage part I

- Consider the following piece of code:

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

- General structure of any conditional branch
- A and B can be large computations, r can be a large state
- This code takes different amount of time, depending on s
- Obvious timing leak if s is secret

Timing leakage part I

- Consider the following piece of code:

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

- General structure of any conditional branch
- A and B can be large computations, r can be a large state
- This code takes different amount of time, depending on s
- Obvious timing leak if s is secret
- Even if A and B take the same amount of cycles this is *not* constant time!
- Reason: Conditional branch takes different amount of cycles whether taken or not
- **Never use secret-data-dependent branch conditions**

Eliminating branches

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

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

Eliminating branches

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

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

- Replace by

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

Eliminating branches

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

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

- Replace by

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

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

Eliminating branches

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

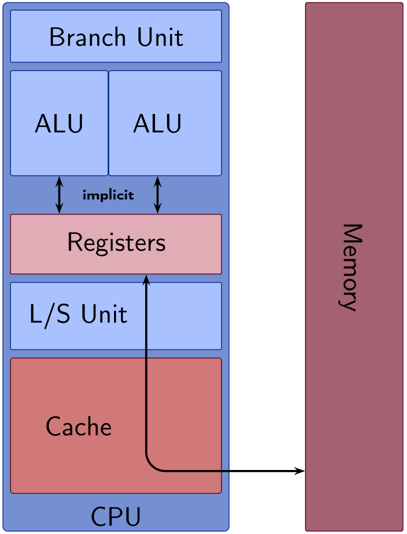
```
if $s$ then
    $r \leftarrow A$
else
    $r \leftarrow B$
end if
```

- Replace by

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

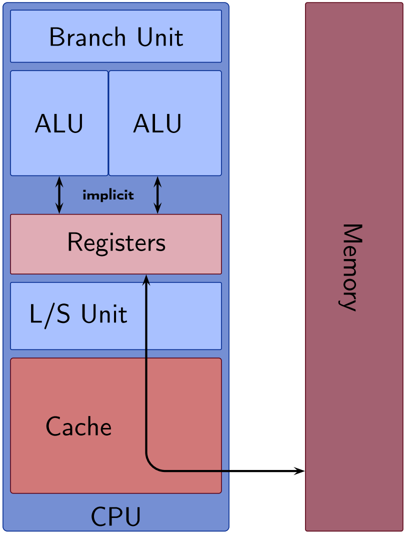
- Can expand s to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication
- For very fast A and B this can even be faster

Cached memory access



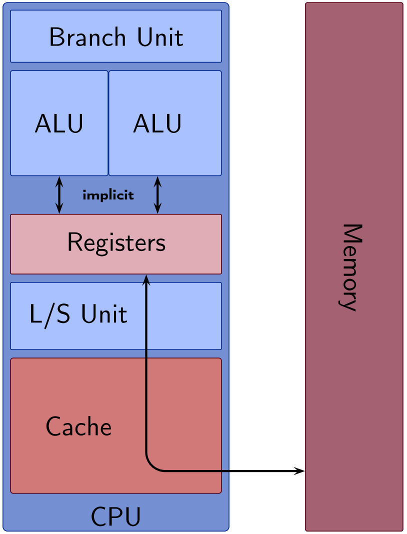
- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data

Cached memory access



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

Cached memory access



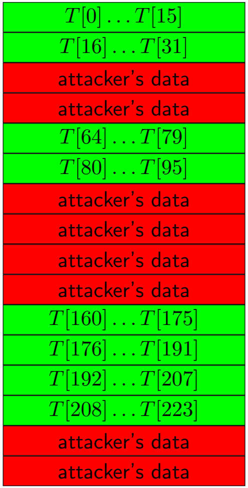
- Memory access goes through a **cache**
- Small but fast transparent memory for frequently used data
- A load from memory places data also in the cache
- Data remains in cache until it's replaced by other data
- Loading data is fast if data is in the cache (**cache hit**)
- Loading data is slow if data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

Timing leakage part II

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

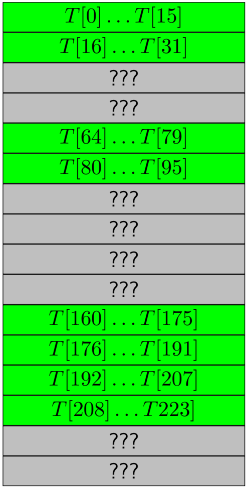
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache

Timing leakage part II



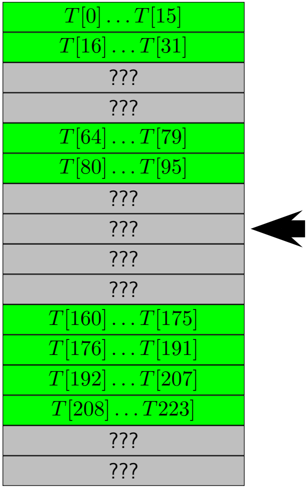
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines

Timing leakage part II



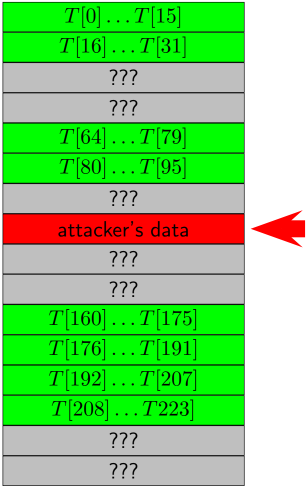
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again

Timing leakage part II



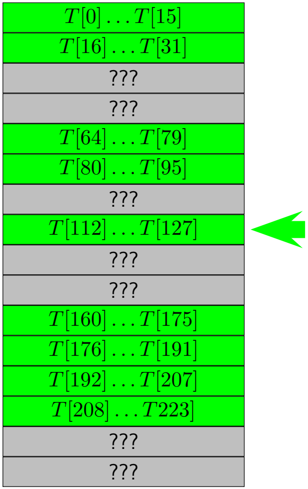
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads his data:

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:

```
static unsigned long long eq(uint32_t a, uint32_t b)
{
  unsigned long long t = a ^ b;
  t = (-t) >> 63;
  return 1-t;
}
```

Timing leakage part III

- Are secret branch conditions and secret load/store addresses the only problem?

Timing leakage part III

- Are secret branch conditions and secret load/store addresses the only problem?
- Answer by Langley: "That's assuming that the fundamental processor instructions are constant time, but that's true for all sane CPUs."

Timing leakage part III

- Are secret branch conditions and secret load/store addresses the only problem?
- Answer by Langley: “That’s assuming that the fundamental processor instructions are constant time, but that’s true for all sane CPUs.”
- Some architectures have *non-constant-time* arithmetic
- Examples:
  - `UMULL`/`SMULL` and `UMLAL`/`SMLAL` on ARM Cortex-M3
  - `DIV` instruction on Intel processors, see also https://www.imperialviolet.org/2013/02/04/luckythirteen.html

Timing leakage part III

- Are secret branch conditions and secret load/store addresses the only problem?
- Answer by Langley: "That's assuming that the fundamental processor instructions are constant time, but that's true for all sane CPUs."
- Some architectures have *non-constant-time* arithmetic
- Examples:
  - `UMULL`/`SMULL` and `UMLAL`/`SMLAL` on ARM Cortex-M3
  - `DIV` instruction on Intel processors, see also https://www.imperialviolet.org/2013/02/04/luckythirteen.html

Summary

- Writing efficient constant-time code is hard
- Typically requires reconsiderations through all optimization levels

SIMD computations

"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.

SISD

Example: 32-bit integer addition

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

SIMD with vector instructions

Example: 4 32-bit integer additions

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

Back to adding 1000 integers

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

Back to adding 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
- Lower bound of 250 cycles

Back to adding 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
- 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 Nehalem processor

Is it really that efficient?

- 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 it really that efficient?

- 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 it really that efficient?

- 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 almost as fast as scalar instructions but do $4\times$ the work**

Is it really that efficient?

- 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 almost as fast as scalar instructions but do $4\times$ the work**
- Situation on other architectures/microarchitectures is similar
- Reason: cheap way to increase arithmetic throughput (less decoding, address computation, etc.)

More reasons for using vector arithmetic

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

More reasons for using vector arithmetic

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

More reasons for using vector arithmetic

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

Vectorization problems I

Carry handling

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

Vectorization problems I

Carry handling

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

Vectorization problems I

Carry handling

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

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/Salsa/Blake
- Vectorization of ChaCha and Salsa can resort to higher-level parallelism (multiple blocks)
- Harder for Blake: each block depends on the previous one

Vectorization problems III

Data shuffeling

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

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

Vectorization problems III

Data shuffeling

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

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

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

Vectorization problems III

Data shuffeling

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

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

Vectorization problems III

Data shuffeling

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

- 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?

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?
- 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.

Efficient vectorization

- Most important question: Where does the parallelism come from?
- Easiest answer: Consider multiple batched encryptions, decryptions, signature computations, verifications, etc.
- 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.
- 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, synergie with constant-time algorithms

Going binary

- So far: considered vectors of integers
- How about arithmetic in binary fields?

Going binary

- So far: considered vectors of integers
- How about arithmetic in binary fields?
- Think of an n-bit register as a vector register with n 1-bit entries
- Operations are now bitwise XOR, AND, OR, etc.

Going binary

- So far: considered vectors of integers
- How about arithmetic in binary fields?
- Think of an n-bit register as a vector register with n 1-bit entries
- Operations are now bitwise XOR, AND, OR, etc.
- This is called *bitslicing*, introduced by Biham in 1997 for DES

Going binary

- So far: considered vectors of integers
- How about arithmetic in binary fields?
- Think of an n-bit register as a vector register with n 1-bit entries
- Operations are now bitwise XOR, AND, OR, etc.
- This is called *bitslicing*, introduced by Biham in 1997 for DES
- Other views on bitslicing:
  - Simulation of hardware implementations in software

Going binary

- So far: considered vectors of integers
- How about arithmetic in binary fields?
- Think of an n-bit register as a vector register with n 1-bit entries
- Operations are now bitwise XOR, AND, OR, etc.
- This is called *bitslicing*, introduced by Biham in 1997 for DES
- Other views on bitslicing:
  - Simulation of hardware implementations in software
  - Computations on a transposition of data

Bitslicing issues

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

Bitslicing issues

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

Introducing AVX

- Vector instruction set introduced by Intel with Sandy Bridge and Ivy Bridge
- 256-bit vector registers `YMM0`,...,`YMM15`
- Overlap with 128-bit `XMM` registers

Introducing AVX

- Vector instruction set introduced by Intel with Sandy Bridge and Ivy Bridge
- 256-bit vector registers `YMM0`,...,`YMM15`
- Overlap with 128-bit `XMM` registers
- Instruction set only supports floating-point vector instructions on `YMM` registers

Introducing AVX

- Vector instruction set introduced by Intel with Sandy Bridge and Ivy Bridge
- 256-bit vector registers `YMM0`,...,`YMM15`
- Overlap with 128-bit `XMM` registers
- Instruction set only supports floating-point vector instructions on `YMM` registers
- Integer-vector instructions follow with AVX2 (Haswell)

Introducing AVX

- Vector instruction set introduced by Intel with Sandy Bridge and Ivy Bridge
- 256-bit vector registers `YMM0`,...,`YMM15`
- Overlap with 128-bit `XMM` registers
- Instruction set only supports floating-point vector instructions on `YMM` registers
- Integer-vector instructions follow with AVX2 (Haswell)
- Very powerful arithmetic: 1 double-precision vector multiplication and 1 double-precision vector addition per cycle (8 FLOPs per cycle per core)

Introducing AVX

- Vector instruction set introduced by Intel with Sandy Bridge and Ivy Bridge
- 256-bit vector registers `YMM0`,...,`YMM15`
- Overlap with 128-bit `XMM` registers
- Instruction set only supports floating-point vector instructions on `YMM` registers
- Integer-vector instructions follow with AVX2 (Haswell)
- Very powerful arithmetic: 1 double-precision vector multiplication and 1 double-precision vector addition per cycle (8 FLOPs per cycle per core)
- Also supported: XOR, AND, OR on `YMM` registers (1 per cycle)

Introducing AVX

- Vector instruction set introduced by Intel with Sandy Bridge and Ivy Bridge
- 256-bit vector registers `YMM0`,...,`YMM15`
- Overlap with 128-bit `XMM` registers
- Instruction set only supports floating-point vector instructions on `YMM` registers
- Integer-vector instructions follow with AVX2 (Haswell)
- Very powerful arithmetic: 1 double-precision vector multiplication and 1 double-precision vector addition per cycle (8 FLOPs per cycle per core)
- Also supported: XOR, AND, OR on `YMM` registers (1 per cycle)
- Alternative: XOR, AND, OR on `XMM` registers (3 per cycle)

Introducing AVX

- Vector instruction set introduced by Intel with Sandy Bridge and Ivy Bridge
- 256-bit vector registers `YMM0`,...,`YMM15`
- Overlap with 128-bit `XMM` registers
- Instruction set only supports floating-point vector instructions on `YMM` registers
- Integer-vector instructions follow with AVX2 (Haswell)
- Very powerful arithmetic: 1 double-precision vector multiplication and 1 double-precision vector addition per cycle (8 FLOPs per cycle per core)
- Also supported: XOR, AND, OR on `YMM` registers (1 per cycle)
- Alternative: XOR, AND, OR on `XMM` registers (3 per cycle)
- However, don't mix `XMM` and `YMM` instruction (context-switch penalty)

Part II

Fast Lattice-Based Signatures

joint work with Tim Güneysu, Tobias Oder, and Thomas Pöppelmann

Introduction

- Consider lattice-based signature scheme proposed by Güneysu, Lyubashevsky, and Pöppelmann at CHES 2012
- "Aggressively optimized" version of scheme by Lyubashevsky (Eurocrypt 2012)

Introduction

- Consider lattice-based signature scheme proposed by Güneysu, Lyubashevsky, and Pöppelmann at CHES 2012
- "Aggressively optimized" version of scheme by Lyubashevsky (Eurocrypt 2012)
- Security level with the implemented parameters:
  - original estimate: 100 bits (against traditional computers)

Introduction

- Consider lattice-based signature scheme proposed by Güneysu, Lyubashevsky, and Pöppelmann at CHES 2012
- "Aggressively optimized" version of scheme by Lyubashevsky (Eurocrypt 2012)
- Security level with the implemented parameters:
  - original estimate: 100 bits (against traditional computers)
  - Lyubashevsky in 2013: 80 bits

Introduction

- Consider lattice-based signature scheme proposed by Güneysu, Lyubashevsky, and Pöppelmann at CHES 2012
- "Aggressively optimized" version of scheme by Lyubashevsky (Eurocrypt 2012)
- Security level with the implemented parameters:
  - original estimate: 100 bits (against traditional computers)
  - Lyubashevsky in 2013: 80 bits
  - 2014: ...?

Introduction

- Consider lattice-based signature scheme proposed by Güneysu, Lyubashevsky, and Pöppelmann at CHES 2012
- "Aggressively optimized" version of scheme by Lyubashevsky (Eurocrypt 2012)
- Security level with the implemented parameters:
    - original estimate: 100 bits (against traditional computers)
    - Lyubashevsky in 2013: 80 bits
    - 2014: . . . ?
- This is not a mature, well understood cryptosystem
- Don't use it in applications, but study it!

Introduction

- Consider lattice-based signature scheme proposed by Güneysu, Lyubashevsky, and Pöppelmann at CHES 2012
- "Aggressively optimized" version of scheme by Lyubashevsky (Eurocrypt 2012)
- Security level with the implemented parameters:
  - original estimate: 100 bits (against traditional computers)
  - Lyubashevsky in 2013: 80 bits
  - 2014: . . . ?
- This is not a mature, well understood cryptosystem
- Don't use it in applications, but study it!
- Implementation techniques are applicable more generally

Notation

- n is a power of 2
- p is a prime congruent to 1 modulo $2n$ (necessary for efficiency)
- $\mathcal{R}$ is the ring $\mathbb{F}_p[x]/\langle x^n + 1\rangle$
- $\mathcal{R}_k$ subset of $\mathcal{R}$ with coefficients in $[-k, k]$.

Lattice hardness assumptions

Standard lattice hardness assumption

Decisional Ring-LWE:

Given $(a_1, t_1), ..., (a_m, t_m) \in \mathcal{R} \times \mathcal{R}$. Decide whether

- $t_i = a_i s + e_i$ where $s, e_1, ..., e_m \leftarrow D_\sigma$ and $a_i \xleftarrow{\$} \mathcal{R}$ (D_σ denotes a Gaussian distribution), or
- (a_i, t_i) uniformly random from $\mathcal{R} \times \mathcal{R}$.

Lattice hardness assumptions

Standard lattice hardness assumption

Decisional Ring-LWE:
Given $(a_1, t_1), ..., (a_m, t_m) \in \mathcal{R} \times \mathcal{R}$. Decide whether

- $t_i = a_i s + e_i$ where $s, e_1, ..., e_m \leftarrow D_\sigma$ and $a_i \overset{\$}{\leftarrow} \mathcal{R}$ (D_σ denotes a Gaussian distribution), or
- (a_i, t_i) uniformly random from $\mathcal{R} \times \mathcal{R}$.

More "aggressive" hardness assumption

Decisional Compact Knapsack Problem (DCKP):
Given $(a, t) \in \mathcal{R} \times \mathcal{R}$.

- Decide whether $t = as_1 + s_2$ where $s_1, s_2 \overset{\$}{\leftarrow} \mathcal{R}_1$ and $a \overset{\$}{\leftarrow} \mathcal{R}$, or
- (a, t) uniformly random from $\mathcal{R} \times \mathcal{R}$.

System parameters

Parameters

- $n = 2^{\ell_1}$
- Prime p with $2n|(p-1)$
- $k = 2^{\ell_2}$ with $\sqrt{p} < k \ll p$
- "Random" $a \in \mathcal{R}$
- Hash function H to elements of $\mathcal{R}_1$ with at most 32 non-zero coefficients

Example

- $n = 512$
- $p = 8383489$ (23 bits)
- $k = 2^{14}$
- Fixed random a
- $\ldots$ more later

Key generation

Secret key

- s_1, s_2 sampled uniformly at random from $\mathcal{R}_1$

Public key

- $t = as_1 + s_2$

Signing (simplified)

Compute a signature σ on a message M as follows:

1. Generate y_1, y_2 uniformly at random from $\mathcal{R}_k$
2. Compute $c = H(ay_1 + y_2, M)$
3. Compute $z_1 = s_1c + y_1$ and $z_2 = s_2c + y_2$
4. If z_1 or $z_2 \notin \mathcal{R}_{k-32}$, goto step 1
5. Return $\sigma = (z_1, z_2, c)$

Verification (simplified)

Check signature $\sigma = (z_1, z_2, c)$ on M as follows:

1. If z_1 or $z_2 \notin \mathcal{R}_{k-32}$, reject
2. Else if $c \neq H(az_1 + z_2 - tc, M)$, reject
3. Else accept

Verification (simplified)

Check signature $\sigma = (z_1, z_2, c)$ on M as follows:

1. If z_1 or $z_2 \notin \mathcal{R}_{k-32}$, reject
2. Else if $c \neq H(az_1 + z_2 - tc, M)$, reject
3. Else accept

Correctness

$$\begin{aligned}
&az_1 + z_2 - tc \\
&= a(s_1c + y_1) + (s_2c + y_2) - (as_1 + s_2)c \\
&= as_1c + ay_1 + s_2c + y_2 - as_1c - s_2c \\
&= ay_1 + y_2
\end{aligned}$$

Software implementation, first considerations

Key generation

- Main operation: sampling random coefficients in $\{-1, 0, 1\}$
- One multiplication of fixed a by s_1

Software implementation, first considerations

Key generation

- Main operation: sampling random coefficients in $\{-1, 0, 1\}$
- One multiplication of fixed a by s_1

Signing

- Expected number of signing attempts: 7
- Each attempt:
  - Sample y_1, y_2 uniformly at random from $\mathcal{R}_k$
  - Two sparse multiplications s_1c and s_2c
  - One multiplication ay_1 by constant a

Software implementation, first considerations

Key generation

- Main operation: sampling random coefficients in $\{-1, 0, 1\}$
- One multiplication of fixed a by s_1

Signing

- Expected number of signing attempts: 7
- Each attempt:
  - Sample y_1, y_2 uniformly at random from $\mathcal{R}_k$
  - Two sparse multiplications $s_1 c$ and $s_2 c$
  - One multiplication $a y_1$ by constant a

Verification

- One sparse multiplication ct
- One multiplication $a z_1$ by constant a

The function H

Need to hash an arbitrary string S to an element $c = (c_0 + c_1 x + \cdots + c_{511} x^{511})$ of $\mathcal{R}_1$ with at most 32 non-zero entries

The function H

Need to hash an arbitrary string S to an element $c = (c_0 + c_1 x + \cdots + c_{511} x^{511})$ of $\mathcal{R}_1$ with at most 32 non-zero entries

- First apply SHA-256, truncate to 160-bit hash h
- Map h injectively to c as follows:
  - Split $(h_0, \ldots, h_{31})$, each h_i with 5 bits
  - Split each h_i into (h_{i0}, h_{it}), where h_{i0} is one bit and h_{it} is a 4-bit integer
  - h_{it} indicates which of the 16 coefficients $c_{16i}, \ldots, c_{16i+15}$ is nonzero
  - If $h_{i0} = 0$ set this coefficient to -1 else to 1

Random sampling, 1st approach

- How do we get an integer, uniformly at random from $[0, m-1]$?
- Let's say that $m-1$ has ℓ bits
- Let's say that we can get random bits (e.g., from `/dev/urandom`)

Random sampling, 1st approach

- How do we get an integer, uniformly at random from $[0, m-1]$?
- Let's say that $m-1$ has ℓ bits
- Let's say that we can get random bits (e.g., from `/dev/urandom`)
- Two answers:
    1. Obtain a random ℓ-bit integer, reject until it is in $[0, m-1]$

Random sampling, 1st approach

- How do we get an integer, uniformly at random from $[0, m-1]$?
- Let's say that $m-1$ has ℓ bits
- Let's say that we can get random bits (e.g., from `/dev/urandom`)
- Two answers:
  1. Obtain a random ℓ-bit integer, reject until it is in $[0, m-1]$
  2. Obtain a much larger integer, reduce mod m (close to uniform)

Random sampling, 1st approach

- How do we get an integer, uniformly at random from $[0, m-1]$?
- Let's say that $m-1$ has ℓ bits
- Let's say that we can get random bits (e.g., from `/dev/urandom`)
- Two answers:
  1. Obtain a random ℓ-bit integer, reject until it is in $[0, m-1]$
  2. Obtain a much larger integer, reduce mod m (close to uniform)
- Probability of rejection in 1. depends on m, it's between 0 and $1/2$

Random sampling, 1st approach

- How do we get an integer, uniformly at random from $[0, m-1]$?
- Let's say that $m-1$ has ℓ bits
- Let's say that we can get random bits (e.g., from `/dev/urandom`)
- Two answers:
    1. Obtain a random ℓ-bit integer, reject until it is in $[0, m-1]$
    2. Obtain a much larger integer, reduce mod m (close to uniform)
- Probability of rejection in 1. depends on m, it's between 0 and $1/2$
- Problem with both 1. and 2.: `/dev/urandom` is slow

Faster random sampling

- Only read seed from `/dev/urandom`, use fast Salsa20 stream cipher
- Salsa20 fast only for long streams, 3 bytes cost as much as 64

Faster random sampling

- Only read seed from `/dev/urandom`, use fast Salsa20 stream cipher
- Salsa20 fast only for long streams, 3 bytes cost as much as 64
- We want truly uniform distribution from $[-k, k]$, recall that $k = 2^{14}$
- We want only one call to Salsa20

Faster random sampling

- Only read seed from `/dev/urandom`, use fast Salsa20 stream cipher
- Salsa20 fast only for long streams, 3 bytes cost as much as 64
- We want truly uniform distribution from $[-k, k]$, recall that $k = 2^{14}$
- We want only one call to Salsa20
- Combine approaches 1 and 2 as follows:
  1. Obtain $4 \cdot (528)$ random bytes from Salsa20
  2. Interpret these bytes as 528 32-bit integers
  3. Discard integers $\geq (2k+1) \cdot \lfloor 2^{32}/(2k+1) \rfloor$.
  4. Probability to discard an integer: 2^{-30}
  5. We have 16 additional integers, replace discarded integers by those
  6. If more than 16 integers are discarded, restart with step 1
  7. For each integer r compute $r \mod (2k+1) - k$

Faster random sampling

- Only read seed from `/dev/urandom`, use fast Salsa20 stream cipher
- Salsa20 fast only for long streams, 3 bytes cost as much as 64
- We want truly uniform distribution from $[-k, k]$, recall that $k = 2^{14}$
- We want only one call to Salsa20
- Combine approaches 1 and 2 as follows:
  1. Obtain $4 \cdot (528)$ random bytes from Salsa20
  2. Interpret these bytes as 528 32-bit integers
  3. Discard integers $\geq (2k+1) \cdot \lfloor 2^{32}/(2k+1) \rfloor$.
  4. Probability to discard an integer: 2^{-30}
  5. We have 16 additional integers, replace discarded integers by those
  6. If more than 16 integers are discarded, restart with step 1
  7. For each integer r compute $r \mod (2k+1) - k$
- Similar approach to sample coefficients in $\{-1, 0, 1\}$
- Only difference: Use bytes instead of 32-bit integers

Representation of elements of $\mathcal{R}$

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \ldots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

Representation of elements of $\mathcal{R}$

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \ldots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

- Use AVX double-precision instructions for addition and multiplication of coefficients

Representation of elements of $\mathcal{R}$

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \dots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

- Use AVX double-precision instructions for addition and multiplication of coefficients
- Modular reduction of a coefficient a:
    - Precompute double-precision approximation $\overline{p^{-1}}$ of p^{-1}

Representation of elements of $\mathcal{R}$

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \dots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

- Use AVX double-precision instructions for addition and multiplication of coefficients
- Modular reduction of a coefficient a:
  - Precompute double-precision approximation $\overline{p^{-1}}$ of p^{-1}
  - Compute $c \leftarrow a \cdot \overline{p^{-1}}$
  - Round c (high-throughput `vroundpd` instruction)
  - Compute $c \leftarrow c \cdot p$
  - Subtract c from a

Representation of elements of $\mathcal{R}$

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \dots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

- Use AVX double-precision instructions for addition and multiplication of coefficients
- Modular reduction of a coefficient a:
  - Precompute double-precision approximation $\overline{p^{-1}}$ of p^{-1}
  - Compute $c \leftarrow a \cdot \overline{p^{-1}}$
  - Round c (high-throughput `vroundpd` instruction)
  - Compute $c \leftarrow c \cdot p$
  - Subtract c from a
  - Rounding mode determines whether this maps to $[-\frac{p-1}{2}, \frac{p-1}{2}]$ or to $[0, p-1]$

Representation of elements of $\mathcal{R}$

- represent $a = \sum_{i=0}^{511} a_i X^i$ as $(a_0, \ldots, a_{511})$:

```
typedef double __attribute__ ((aligned (32))) r_elem[512];
```

- Use AVX double-precision instructions for addition and multiplication of coefficients
- Modular reduction of a coefficient a:
  - Precompute double-precision approximation $\overline{p^{-1}}$ of p^{-1}
  - Compute $c \leftarrow a \cdot \overline{p^{-1}}$
  - Round c (high-throughput `vroundpd` instruction)
  - Compute $c \leftarrow c \cdot p$
  - Subtract c from a
  - Rounding mode determines whether this maps to $[-\frac{p-1}{2}, \frac{p-1}{2}]$ or to $[0, p-1]$
- Use lazy reduction: product of two 22-bit numbers has 44 bits, quite some space in the 53-bit mantissa

Multiplication in $\mathcal{R}$

- Let ω be a 512th root of unity in $\mathbb{F}_p$ and $\psi^2 = \omega$
- The number-theoretic transform NTT_ω of $a = (a_0, \ldots, a_{511})$ is defined as

$$\mathsf{NTT}_\omega(a) = (A_0, \ldots, A_{511}) \text{ with } A_i = \sum_{j=0}^{511} a_j \omega^{ij}$$

Multiplication in $\mathcal{R}$

- Let ω be a 512th root of unity in $\mathbb{F}_p$ and $\psi^2 = \omega$
- The number-theoretic transform NTT_ω of $a = (a_0, \ldots, a_{511})$ is defined as

$$\mathsf{NTT}_\omega(a) = (A_0, \ldots, A_{511}) \text{ with } A_i = \sum_{j=0}^{511} a_j \omega^{ij}$$

- Consider multiplication $d = a \cdot b$ in $\mathcal{R}$
- Compute

$$\bar{a} = (a_0, \psi a_1, \ldots, \psi^{511} a_{511}) \text{ and}$$
$$\bar{b} = (b_0, \psi b_1, \ldots, \psi^{511} b_{511})$$

Multiplication in $\mathcal{R}$

- Let ω be a 512th root of unity in $\mathbb{F}_p$ and $\psi^2 = \omega$
- The number-theoretic transform NTT_ω of $a = (a_0, \ldots, a_{511})$ is defined as

$$\mathsf{NTT}_\omega(a) = (A_0, \ldots, A_{511}) \text{ with } A_i = \sum_{j=0}^{511} a_j \omega^{ij}$$

- Consider multiplication $d = a \cdot b$ in $\mathcal{R}$
- Compute

$$\bar{a} = (a_0, \psi a_1, \ldots, \psi^{511} a_{511}) \text{ and}$$
$$\bar{b} = (b_0, \psi b_1, \ldots, \psi^{511} b_{511})$$

- Obtain $\bar{d} = (d_0, \psi d_1, \ldots, \psi^{511} d_{511})$ as

$$\bar{d} = \mathsf{NTT}_\omega^{-1}(\mathsf{NTT}_\omega(\bar{a}) \circ \mathsf{NTT}_\omega(\bar{b})),$$

where $\circ$ denotes component-wise multiplication

Multiplication in $\mathcal{R}$

- Let ω be a 512th root of unity in $\mathbb{F}_p$ and $\psi^2 = \omega$
- The number-theoretic transform NTT_ω of $a = (a_0, \dots, a_{511})$ is defined as

$$\mathsf{NTT}_\omega(a) = (A_0, \dots, A_{511}) \text{ with } A_i = \sum_{j=0}^{511} a_j \omega^{ij}$$

- Consider multiplication $d = a \cdot b$ in $\mathcal{R}$
- Compute

$$\bar{a} = (a_0, \psi a_1, \dots, \psi^{511} a_{511}) \text{ and}$$
$$\bar{b} = (b_0, \psi b_1, \dots, \psi^{511} b_{511})$$

- Obtain $\bar{d} = (d_0, \psi d_1, \dots, \psi^{511} d_{511})$ as

$$\bar{d} = \mathsf{NTT}_\omega^{-1}(\mathsf{NTT}_\omega(\bar{a}) \circ \mathsf{NTT}_\omega(\bar{b})),$$

  where $\circ$ denotes component-wise multiplication
- Component-wise multiplication is trivially vectorizable

The (NTT)

- FFT in a finite field
- Evaluate polynomial $f = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + xf_1(x^2)$

The (NTT)

- FFT in a finite field
- Evaluate polynomial $f = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$
  - Huge overlap between evaluating

$$f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \text{ and}$$
$$f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)$$

The (NTT)

- FFT in a finite field
- Evaluate polynomial $f = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$
  - Huge overlap between evaluating

$$f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \text{ and}$$
$$f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)$$

  - f_0 has $n/2$ coefficients
  - Evaluate f_0 at all $(n/2)$-th roots of unity by recursive application

The (NTT)

- FFT in a finite field
- Evaluate polynomial $f = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$
  - Huge overlap between evaluating

$$f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \text{ and}$$
$$f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)$$

  - f_0 has $n/2$ coefficients
  - Evaluate f_0 at all $(n/2)$-th roots of unity by recursive application
  - Same for f_1

The (NTT)

- FFT in a finite field
- Evaluate polynomial $f = a_0 + a_1x + \dots + a_{n-1}x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + xf_1(x^2)$
  - Huge overlap between evaluating

$$f(\beta) = f_0(\beta^2) + \beta f_1(\beta^2) \text{ and}$$
$$f(-\beta) = f_0(\beta^2) - \beta f_1(\beta^2)$$

  - f_0 has $n/2$ coefficients
  - Evaluate f_0 at all $(n/2)$-th roots of unity by recursive application
  - Same for f_1
- For $n = 512$ we have 9 levels of recursion

NTT in AVX (Part I)

- First thing to do: replace recursion by iteration
- Loop over 9 levels with 256 "butterfly transformations" each
- Butterfly on level k:
  - Pick up a_i and a_{i+2^k}
  - Multiply a_{i+2^k} by a power of ω to obtain t
  - Compute $a_{i+2^k} \leftarrow a_i - t$
  - Compute $a_i \leftarrow a_i + t$
- Easy vectorization on levels $k = 2, \dots, 8$:
  - Pick up $v_0 = a_i, a_{i+1}, a_{i+2}, a_{i+3}$ and $v_1 = a_{i+2^k}, a_{i+2^k+1}, a_{i+2^k+2}, a_{i+2^k+3}$
  - Perform all operations on v_0 and v_1
- Levels 0 and 1: More tricky: Use permutation instructions and "horizontal additions"

NTT in AVX (Part II)

- Main bottleneck of NTT: memory access

NTT in AVX (Part II)

- Main bottleneck of NTT: memory access
- On one level of butterfly, pairs of values interact
- Through two levels, 4-tuples interact
- Through three levels, 8-tuples interact, etc.

NTT in AVX (Part II)

- Main bottleneck of NTT: memory access
- On one level of butterfly, pairs of values interact
- Through two levels, 4-tuples interact
- Through three levels, 8-tuples interact, etc.
- Merge 3 levels: Load $8 \cdot 4 = 32$ values, perform arithmetic, store the results

NTT in AVX (Part II)

- Main bottleneck of NTT: memory access
- On one level of butterfly, pairs of values interact
- Through two levels, 4-tuples interact
- Through three levels, 8-tuples interact, etc.
- Merge 3 levels: Load $8 \cdot 4 = 32$ values, perform arithmetic, store the results
- Final performance for NTT: 4484 cycles on Ivy Bridge
- Performance for multiplication in $\mathcal{R}$: 16096 cycles
- Multiplication by constant a: 11044 cycles

Results

- Keypair generation: 31140 cycles on Intel Ivy Bridge
- Signing: 634988 cycles on average
- Verification: 45036 cycles

Results

- Keypair generation: 31140 cycles on Intel Ivy Bridge
- Signing: 634988 cycles on average
- Verification: 45036 cycles
- Public key: 1536 bytes
- Secret key: 256 bytes
- Signature: 1184 bytes

Comparison

Software	Cycles		Sizes	
Our work	**sign:**	634988	**pk:**	1536
	verify:	45036	**sk:**	256
			sig:	1184
`mqqsig160`	**sign:**	1996	**pk:**	206112
	verify:	33220	**sk:**	401
			sig:	20
`rainbow5640`	**sign:**	53872	**pk:**	44160
	verify:	34808	**sk:**	86240
			sig:	37
`pflash1`	**sign:**	1473364	**pk:**	72124
	verify:	286168	**sk:**	5550
			sig:	37
`tts6440`	**sign:**	33728	**pk:**	57600
	verify:	49248	**sk:**	16608
			sig:	43
XMSS	**sign:**	7261100^*	**pk:**	912
($H = 20, w = 4$, AES-128)	**verify:**	556600^*	**sk:**	19
			sig:	2451

References

- Tim Güneysu, Tobias Oder, Thomas Pöppelmann, and Peter Schwabe. *Software speed records for lattice-based signatures.*, PQCrypto 2013. `http://cryptojedi.org/papers/#lattisigns`
- Software is online (public domain) at `http://cryptojedi.org/crypto/#lattisigns`

Part III

McBits: Fast code-based cryptography

joint work with Daniel J. Bernstein and Tung Chou

Public-key encryption

- Alice generates a key pair (sk, pk), publishes pk, keeps sk secret

Public-key encryption

- Alice generates a key pair (sk, pk), publishes pk, keeps sk secret
- Bob takes some message M and pk and computes **ciphertext** C, sends C to Alice

Public-key encryption

- Alice generates a key pair (sk, pk), publishes pk, keeps sk secret
- Bob takes some message M and pk and computes **ciphertext** C, sends C to Alice
- Alice uses sk decrypt C

System parameters

Parameters

- Integers m, q, n, t, k, such that
  - $n \leq q = 2^m$
  - $k = n - mt$
  - $t \geq 2$

Example

- $m = 12$,
  $n = q = 4096$
  $k = 3604$
  $t = 41$

System parameters

Parameters

- Integers m, q, n, t, k, such that
  - $n \leq q = 2^m$
  - $k = n - mt$
  - $t \geq 2$
- An s-bit-key stream cipher S

Example

- $m = 12$,
  $n = q = 4096$
  $k = 3604$
  $t = 41$
- $S =$ Salsa20 $(s = 256)$

System parameters

Parameters

- Integers m, q, n, t, k, such that
  - $n \leq q = 2^m$
  - $k = n - mt$
  - $t \geq 2$
- An s-bit-key stream cipher S
- An a-bit-key authenticator (MAC) A

Example

- $m = 12$,
  $n = q = 4096$
  $k = 3604$
  $t = 41$
- $S =$ Salsa20 $(s = 256)$
- $A =$ Poly1305 $(a = 256)$

System parameters

Parameters

- Integers m, q, n, t, k, such that
  - $n \leq q = 2^m$
  - $k = n - mt$
  - $t \geq 2$
- An s-bit-key stream cipher S
- An a-bit-key authenticator (MAC) A
- An $(s + a)$-bit-output hash function H

Example

- $m = 12$,
  $n = q = 4096$
  $k = 3604$
  $t = 41$
- $S =$ Salsa20 $(s = 256)$
- $A =$ Poly1305 $(a = 256)$
- $H =$ SHA-512

Key generation

Secret key

- A random sequence $(\alpha_1, \ldots, \alpha_n)$ of distinct elements in $\mathbb{F}_q$
- A irreducible degree-t polynomial $g \in \mathbb{F}_q[x]$

Key generation

Secret key

- A random sequence $(\alpha_1, \ldots, \alpha_n)$ of distinct elements in $\mathbb{F}_q$
- A irreducible degree-t polynomial $g \in \mathbb{F}_q[x]$
- Compute the secret matrix

$$\begin{pmatrix} 1/g(\alpha_1) & 1/g(\alpha_2) & \cdots & 1/g(\alpha_n) \\ \alpha_1/g(\alpha_1) & \alpha_2/g(\alpha_2) & \cdots & \alpha_n/g(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1}/g(\alpha_1) & \alpha_2^{t-1}/g(\alpha_2) & \cdots & \alpha_n^{t-1}/g(\alpha_n) \end{pmatrix} \in \mathbb{F}_q^{t\times n}$$

Key generation

Secret key

- A random sequence $(\alpha_1, \dots, \alpha_n)$ of distinct elements in $\mathbb{F}_q$
- A irreducible degree-t polynomial $g \in \mathbb{F}_q[x]$
- Compute the secret matrix

$$\begin{pmatrix} 1/g(\alpha_1) & 1/g(\alpha_2) & \cdots & 1/g(\alpha_n) \\ \alpha_1/g(\alpha_1) & \alpha_2/g(\alpha_2) & \cdots & \alpha_n/g(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1}/g(\alpha_1) & \alpha_2^{t-1}/g(\alpha_2) & \cdots & \alpha_n^{t-1}/g(\alpha_n) \end{pmatrix} \in \mathbb{F}_q^{t \times n}$$

- Replace all entries by a column of m bits in a standard basis of $\mathbb{F}_q$ over $\mathbb{F}_2$
- Obtain a matrix $H \in \mathbb{F}_2^{mt \times n}$

Key generation

Secret key

- A random sequence $(\alpha_1, \ldots, \alpha_n)$ of distinct elements in $\mathbb{F}_q$
- A irreducible degree-t polynomial $g \in \mathbb{F}_q[x]$
- Compute the secret matrix

$$\begin{pmatrix} 1/g(\alpha_1) & 1/g(\alpha_2) & \cdots & 1/g(\alpha_n) \\ \alpha_1/g(\alpha_1) & \alpha_2/g(\alpha_2) & \cdots & \alpha_n/g(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1}/g(\alpha_1) & \alpha_2^{t-1}/g(\alpha_2) & \cdots & \alpha_n^{t-1}/g(\alpha_n) \end{pmatrix} \in \mathbb{F}_q^{t \times n}$$

- Replace all entries by a column of m bits in a standard basis of $\mathbb{F}_q$ over $\mathbb{F}_2$
- Obtain a matrix $H \in \mathbb{F}_2^{mt \times n}$
- H is a *secret* parity-check matrix of the Goppa code $\Gamma = \Gamma_2(\alpha_1, \ldots, \alpha_n, g)$

Key generation

Secret key

- A random sequence $(\alpha_1, \ldots, \alpha_n)$ of distinct elements in $\mathbb{F}_q$
- A irreducible degree-t polynomial $g \in \mathbb{F}_q[x]$
- Compute the secret matrix

$$\begin{pmatrix} 1/g(\alpha_1) & 1/g(\alpha_2) & \cdots & 1/g(\alpha_n) \\ \alpha_1/g(\alpha_1) & \alpha_2/g(\alpha_2) & \cdots & \alpha_n/g(\alpha_n) \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1}/g(\alpha_1) & \alpha_2^{t-1}/g(\alpha_2) & \cdots & \alpha_n^{t-1}/g(\alpha_n) \end{pmatrix} \in \mathbb{F}_q^{t \times n}$$

- Replace all entries by a column of m bits in a standard basis of $\mathbb{F}_q$ over $\mathbb{F}_2$
- Obtain a matrix $H \in \mathbb{F}_2^{mt \times n}$
- H is a *secret* parity-check matrix of the Goppa code $\Gamma = \Gamma_2(\alpha_1, \ldots, \alpha_n, g)$
- The secret key is $(\alpha_1, \ldots, \alpha_n, g)$

Key generation

Public key

- Perform Gaussian elimination on H to obtain a matrix K whose left $tm \times tm$ submatrix is the identity matrix
- K is a *public* parity-check matrix for Γ
- The public key is K

Encryption

- Generate a random weight-t vector $e \in \mathbb{F}_2^n$
- Compute $w = Ke$
- Compute $H(e)$ to obtain an $(s + a)$-bit string (k_{enc}, k_{auth})
- Encrypt the message M with the stream cipher S under key k_{enc} to obtain ciphertext C
- Compute authentication tag a on C using A with key k_{auth}
- Send (a, w, C)

Decryption

- Receive (a, w, C)
- Decode w to obtain weight-t string e
- Hash e with H to obtain (k_{enc}, k_{auth})
- Verify that a is a valid authentication tag on C using A with k_{auth}
- Use S with k_{enc} to decrypt and obtain M

Software implementation, first considerations

Key generation

- Key generation is not performance critical
- Some hassle to make constant-time, but possible

Software implementation, first considerations

Key generation

- Key generation is not performance critical
- Some hassle to make constant-time, but possible

Encryption

- Computation of Ke is simply XORing t columns of mt bits each
- In our example $mt = 492$, almost 512; great for fast vector XORs
- But: have to be careful to not leak information about e
- This talk: ignore implementation of H, S, and A

Software implementation, first considerations

Key generation

- Key generation is not performance critical
- Some hassle to make constant-time, but possible

Encryption

- Computation of Ke is simply XORing t columns of mt bits each
- In our example $mt = 492$, almost 512; great for fast vector XORs
- But: have to be careful to not leak information about e
- This talk: ignore implementation of H, S, and A

Decryption

- Decryption is mainly decoding, lots of operations $\mathbb{F}_q$
- Decryption has to run in constant time!
- Obviously, decoding of w is the interesting part

A closer look at decoding

- Start with *some* $v \in \mathbb{F}_2^n$, such that $Kv = w$

A closer look at decoding

- Start with *some* $v \in \mathbb{F}_2^n$, such that $Kv = w$
- Compute a Goppa syndrome $s_0, \dots, s_{2t-1}$
- Use Berlekamp-Massey algorithm to obtain error-locator polynomial f of degree t

A closer look at decoding

- Start with *some* $v \in \mathbb{F}_2^n$, such that $Kv = w$
- Compute a Goppa syndrome $s_0, \dots, s_{2t-1}$
- Use Berlekamp-Massey algorithm to obtain error-locator polynomial f of degree t
- Compute t roots of this polynomial
- For each root $r_j = \alpha_i$, set error bit at position i in e

A closer look at decoding

- Start with *some* $v \in \mathbb{F}_2^n$, such that $Kv = w$
- Compute a Goppa syndrome $s_0, \ldots, s_{2t-1}$
- Use Berlekamp-Massey algorithm to obtain error-locator polynomial f of degree t
- Compute t roots of this polynomial
- For each root $r_j = \alpha_i$, set error bit at position i in e
- All these computation work on medium-size polynomials over $\mathbb{F}_q$

A closer look at decoding

- Start with *some* $v \in \mathbb{F}_2^n$, such that $Kv = w$
- Compute a Goppa syndrome $s_0, \dots, s_{2t-1}$
- Use Berlekamp-Massey algorithm to obtain error-locator polynomial f of degree t
- Compute t roots of this polynomial
- For each root $r_j = \alpha_i$, set error bit at position i in e
- All these computation work on medium-size polynomials over $\mathbb{F}_q$
- Let's now fix the example parameters from above $(q = 2^m = 4096, t = 41, n = q)$

Representing elements of $\mathbb{F}_p$

Option I

- Use 16-bit integer values (`unsigned short`)
- Addition is simply XOR (we really XOR 64 bits, but ignore most of those)

Representing elements of $\mathbb{F}_p$

Option I

- Use 16-bit integer values (`unsigned short`)
- Addition is simply XOR (we really XOR 64 bits, but ignore most of those)
- Multiplication:
  - Use table lookups (not constant time!)

Representing elements of $\mathbb{F}_p$

Option I

- Use 16-bit integer values (`unsigned short`)
- Addition is simply XOR (we really XOR 64 bits, but ignore most of those)
- Multiplication:
  - Use table lookups (not constant time!)
  - Use carryless multiplier, e.g., `pclmulqdq` (not available on most architectures, again ignores most of the 64×64-bit multiplication)

Representing elements of $\mathbb{F}_p$

Option I

- Use 16-bit integer values (`unsigned short`)
- Addition is simply XOR (we really XOR 64 bits, but ignore most of those)
- Multiplication:
  - Use table lookups (not constant time!)
  - Use carryless multiplier, e.g., `pclmulqdq` (not available on most architectures, again ignores most of the 64×64-bit multiplication)
  - Squaring uses the same algorithm as multiplication

Representing elements of $\mathbb{F}_p$

Option II

- Use bitsliced representation in 256-bit `YMM` (or 128-bit `XMM` registers)
- Needs many parallel computations, obtain parallelism from independent decryption operations
- We only really care about speed when we have *many* decryptions

Representing elements of $\mathbb{F}_p$

Option II

- Use bitsliced representation in 256-bit `YMM` (or 128-bit `XMM` registers)
- Needs many parallel computations, obtain parallelism from independent decryption operations
- We only really care about speed when we have *many* decryptions
- Addition is 12 vectors XORs for 256 parallel additions (much faster!)

Representing elements of $\mathbb{F}_p$

Option II

- Use bitsliced representation in 256-bit `YMM` (or 128-bit `XMM` registers)
- Needs many parallel computations, obtain parallelism from independent decryption operations
- We only really care about speed when we have *many* decryptions
- Addition is 12 vectors XORs for 256 parallel additions (much faster!)
- Multiplication is easily constant time, but is it fast?
- How about squaring, can it be faster?

Bitsliced multiplication in $\mathbb{F}_{2^{12}}$

- Split into 12-coefficient polynomial multiplication and subsequent reduction
- Reduction trinomial $x^{12} + x^3 + 1$

Bitsliced multiplication in $\mathbb{F}_{2^{12}}$

- Split into 12-coefficient polynomial multiplication and subsequent reduction
- Reduction trinomial $x^{12} + x^3 + 1$
- Schoolbook multiplication needs 144 ANDs and 121 XORs

Bitsliced multiplication in $\mathbb{F}_{2^{12}}$

- Split into 12-coefficient polynomial multiplication and subsequent reduction
- Reduction trinomial $x^{12} + x^3 + 1$
- Schoolbook multiplication needs 144 ANDs and 121 XORs
- Much better: Karatsuba
  - Karatsuba:

$$\begin{aligned} & (a_0 + x^n a_1)(b_0 + x^n b_1) \\ = \quad & a_0 b_0 + x^n((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1) + x^{2n} a_1 b_1 \end{aligned}$$

Bitsliced multiplication in $\mathbb{F}_{2^{12}}$

- Split into 12-coefficient polynomial multiplication and subsequent reduction
- Reduction trinomial $x^{12} + x^3 + 1$
- Schoolbook multiplication needs 144 ANDs and 121 XORs
- Much better: refined Karatsuba
  - Karatsuba:

$$\begin{aligned} & (a_0 + x^n a_1)(b_0 + x^n b_1) \\ = \quad & a_0 b_0 + x^n((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1) + x^{2n} a_1 b_1 \end{aligned}$$

  - Refined Karatsuba:

$$\begin{aligned} & (a_0 + x^n a_1)(b_0 + x^n b_1) \\ = \quad & (1 - x^n)(a_0 b_0 - x^n a_1 b_1) + x^n (a_0 + a_1)(b_0 + b_1) \end{aligned}$$

- Refined Karatsuba uses $M_{2n} = 3M_n + 7n - 3$ instead of $M_{2n} = 3M_n + 8n - 4$ bit operations
- For details see Bernstein "Batch binary Edwards", Crypto 2009

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling
- Reduction takes 24 XORs, a total of 246 bit operations
- On Ivy Bridge: 247 cycles for 256 multiplications

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling
- Reduction takes 24 XORs, a total of 246 bit operations
- On Ivy Bridge: 247 cycles for 256 multiplications
- Bitsliced squaring is only reduction: 7 XORs

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling
- Reduction takes 24 XORs, a total of 246 bit operations
- On Ivy Bridge: 247 cycles for 256 multiplications
- Bitsliced squaring is only reduction: 7 XORs
- Future work: Explore tower-field arithmetic, reduce bit operations

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling
- Reduction takes 24 XORs, a total of 246 bit operations
- On Ivy Bridge: 247 cycles for 256 multiplications
- Bitsliced squaring is only reduction: 7 XORs
- Future work: Explore tower-field arithmetic, reduce bit operations

Summary:

- Bitsliced *addition* is much faster than non bitsliced
- Bitsliced *multiplication* is competitive
- Bitsliced squaring is much faster (not very relevant)

Bitsliced performance

- One level of refined Karatsuba: 114 XORs, 108 ANDs
- 222 bit operations are worse than 208 by Bernstein 2009, but better scheduling
- Reduction takes 24 XORs, a total of 246 bit operations
- On Ivy Bridge: 247 cycles for 256 multiplications
- Bitsliced squaring is only reduction: 7 XORs
- Future work: Explore tower-field arithmetic, reduce bit operations

Summary:

- Bitsliced *addition* is much faster than non bitsliced
- Bitsliced *multiplication* is competitive
- Bitsliced squaring is much faster (not very relevant)
- In the following: High-level algorithms that drastically reduce the number of multiplications

Root finding, the classical way

- Task: Find all t roots of a degree-t error-locator polynomial f
- Let $f = c_{41}x^{41} + c_{40} + x^{40} + \cdots + c_0$

Root finding, the classical way

- Task: Find all t roots of a degree-t error-locator polynomial f
- Let $f = c_{41}x^{41} + c_{40} + x^{40} + \cdots + c_0$
- Try all elements of F_q, Horner scheme takes 41 mul, 41 add per element

Root finding, the classical way

- Task: Find all t roots of a degree-t error-locator polynomial f
- Let $f = c_{41}x^{41} + c_{40} + x^{40} + \cdots + c_0$
- Try all elements of F_q, Horner scheme takes 41 mul, 41 add per element
- Chien search: Compute $c_i g^i, c_i g^{2i}, c_i g^{3i}$ etc.
- Same operation count but different structure

Root finding, the classical way

- Task: Find all t roots of a degree-t error-locator polynomial f
- Let $f = c_{41}x^{41} + c_{40} + x^{40} + \cdots + c_0$
- Try all elements of F_q, Horner scheme takes 41 mul, 41 add per element
- Chien search: Compute $c_i g^i, c_i g^{2i}, c_i g^{3i}$ etc.
- Same operation count but different structure
- Berlekamp trace algorithm: not constant time

Remember the FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$

Remember the FFT

- Evaluate a polynomial $f = c_0 + c_1x + \cdots + c_{n-1}x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + xf_1(x^2)$
  - Huge overlap between evaluating

$$f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2) \text{ and}$$
$$f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2)$$

Remember the FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$
  - Huge overlap between evaluating

$$f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2) \text{ and}$$
$$f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2)$$

- Problem: We have a binary field, and $\alpha = -\alpha$

Remember the FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + xf_1(x^2)$
  - Huge overlap between evaluating

$$f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2) \text{ and}$$
$$f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2)$$

- Problem: We have a binary field, and $\alpha = -\alpha$
- Wang, Zhu 1988, and independently Cantor 1989: additive FFT in characteristic 2 (quite slow)

Remember the FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$
  - Huge overlap between evaluating

$$f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2) \text{ and}$$
$$f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2)$$

- Problem: We have a binary field, and $\alpha = -\alpha$
- Wang, Zhu 1988, and independently Cantor 1989: additive FFT in characteristic 2 (quite slow)
- von zur Gathen 1996: some improvements (still slow)

Remember the FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(x^2) + x f_1(x^2)$
  - Huge overlap between evaluating

$$f(\alpha) = f_0(\alpha^2) + \alpha f_1(\alpha^2) \text{ and}$$
$$f(-\alpha) = f_0(\alpha^2) - \alpha f_1(\alpha^2)$$

- Problem: We have a binary field, and $\alpha = -\alpha$
- Wang, Zhu 1988, and independently Cantor 1989: additive FFT in characteristic 2 (quite slow)
- von zur Gathen 1996: some improvements (still slow)
- Gao, Mateer 2010: Much faster additive FFT

Gao-Mateer additive FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1}$ on a size-n $\mathbb{F}_2$-linear space S
- Think of S as all subset sums of $\{\beta_1, \ldots, \beta_m\}, \beta_i \in \mathbb{F}_q$
- Idea: Write polynomial f as $f_0(x^2 + x) + x f_1(x^2 + x)$

Gao-Mateer additive FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$ on a size-n $\mathbb{F}_2$-linear space S
- Think of S as all subset sums of $\{\beta_1, \ldots, \beta_m\}, \beta_i \in \mathbb{F}_q$
- Idea: Write polynomial f as $f_0(x^2 + x) + x f_1(x^2 + x)$
- Big overlap between evaluating

$$f(\alpha) = f_0(\alpha^2 + \alpha) + \alpha f_1(\alpha^2 + \alpha) \text{ and}$$
$$f(\alpha + 1) = f_0(\alpha^2 + \alpha) + (\alpha + 1) f_1(\alpha^2 + \alpha)$$

Gao-Mateer additive FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1}x^{n-1}$ on a size-n $\mathbb{F}_2$-linear space S
- Think of S as all subset sums of $\{\beta_1, \ldots, \beta_m\}, \beta_i \in \mathbb{F}_q$
- Idea: Write polynomial f as $f_0(x^2 + x) + x f_1(x^2 + x)$
- Big overlap between evaluating

$$f(\alpha) = f_0(\alpha^2 + \alpha) + \alpha f_1(\alpha^2 + \alpha) \text{ and}$$
$$f(\alpha + 1) = f_0(\alpha^2 + \alpha) + (\alpha + 1) f_1(\alpha^2 + \alpha)$$

- Evaluate f_0 and f_1 at $\alpha^2 + \alpha$, obtain $f(\alpha)$ and $f(\alpha + 1)$ with only 1 multiplication and 2 additions

Gao-Mateer additive FFT

- Evaluate a polynomial $f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1}$ on a size-n $\mathbb{F}_2$-linear space S
- Think of S as all subset sums of $\{\beta_1, \ldots, \beta_m\}, \beta_i \in \mathbb{F}_q$
- Idea: Write polynomial f as $f_0(x^2 + x) + x f_1(x^2 + x)$
- Big overlap between evaluating

$$f(\alpha) = f_0(\alpha^2 + \alpha) + \alpha f_1(\alpha^2 + \alpha) \text{ and}$$
$$f(\alpha + 1) = f_0(\alpha^2 + \alpha) + (\alpha + 1) f_1(\alpha^2 + \alpha)$$

- Evaluate f_0 and f_1 at $\alpha^2 + \alpha$, obtain $f(\alpha)$ and $f(\alpha + 1)$ with only 1 multiplication and 2 additions
- Again: apply the idea recursively

Gao-Mateer for syndrome computation

- Application in decoding: much smaller degree of f
- Our paper: generalize the idea to small-degree f

Gao-Mateer for syndrome computation

- Application in decoding: much smaller degree of f
- Our paper: generalize the idea to small-degree f
- Recursion can stop much earlier

Gao-Mateer for syndrome computation

- Application in decoding: much smaller degree of f
- Our paper: generalize the idea to small-degree f
- Recursion can stop much earlier
- More improvements at the end of the recursion:
  - For constant f_1, simply return 2^m copies of $f_1(0) = c$

Gao-Mateer for syndrome computation

- Application in decoding: much smaller degree of f
- Our paper: generalize the idea to small-degree f
- Recursion can stop much earlier
- More improvements at the end of the recursion:
    - For constant f_1, simply return 2^m copies of $f_1(0) = c$
    - For 2-coefficient or 3-coefficient f, we have constant f_1
    - Need $2^{m-1} - 1$ multiplications αc

Gao-Mateer for syndrome computation

- Application in decoding: much smaller degree of f
- Our paper: generalize the idea to small-degree f
- Recursion can stop much earlier
- More improvements at the end of the recursion:
  - For constant f_1, simply return 2^m copies of $f_1(0) = c$
  - For 2-coefficient or 3-coefficient f, we have constant f_1
  - Need $2^{m-1} - 1$ multiplications αc
  - Instead perform $m - 1$ multiplications to obtain $c\beta_1, \dots, c\beta_{m-1}$ (assume that $\beta_m = 1$)
  - Obtain results as subset sums of $c\beta_1, \dots, c\beta_{m-1}$
  - Replace $2^{m-1} - m$ multiplications by additions

Gao-Mateer for syndrome computation

- Application in decoding: much smaller degree of f
- Our paper: generalize the idea to small-degree f
- Recursion can stop much earlier
- More improvements at the end of the recursion:
  - For constant f_1, simply return 2^m copies of $f_1(0) = c$
  - For 2-coefficient or 3-coefficient f, we have constant f_1
  - Need $2^{m-1} - 1$ multiplications αc
  - Instead perform $m - 1$ multiplications to obtain $c\beta_1, \dots, c\beta_{m-1}$ (assume that $\beta_m = 1$)
  - Obtain results as subset sums of $c\beta_1, \dots, c\beta_{m-1}$
  - Replace $2^{m-1} - m$ multiplications by additions
- Overall count: fewer additions and *much* fewer multiplications than Horner scheme or Chien search

Syndrome computation, the classical way

- Receive n-bit input word, scale bits by Goppa constants
- Apply linear map

$$M = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{2t-1} & \alpha_2^{2t-1} & \cdots & \alpha_n^{2t-1} \end{pmatrix}$$

Syndrome computation, the classical way

- Receive n-bit input word, scale bits by Goppa constants
- Apply linear map

$$M = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{2t-1} & \alpha_2^{2t-1} & \cdots & \alpha_n^{2t-1} \end{pmatrix}$$

- Can precompute matrix mapping bits to syndrome
- Similar to encryption, but input does not have weight t
- Needs to run in constant time!

Another look at syndrome computation

Look at the syndrome-computation map again:

$$M = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_n \\ \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{2t-1} & \alpha_2^{2t-1} & \cdots & \alpha_n^{2t-1} \end{pmatrix}$$

Consider the linear map $M^\intercal$:

$$\begin{pmatrix} 1 & \alpha_1 & \cdots & \alpha_1^{2t-1} \\ 1 & \alpha_2 & \cdots & \alpha_2^{2t-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_n & \cdots & \alpha_n^{2t-1} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_t \end{pmatrix} = \begin{pmatrix} v_1 + v_2\alpha_1 + \cdots + v_t\alpha_1^{2t-1} \\ v_1 + v_2\alpha_2 + \cdots + v_t\alpha_2^{2t-1} \\ \vdots \\ v_1 + v_2\alpha_n + \cdots + v_t\alpha_n^{2t-1} \end{pmatrix} = \begin{pmatrix} f(\alpha_1) \\ f(\alpha_2) \\ \vdots \\ f(\alpha_n) \end{pmatrix}$$

- This transposed linear map is actually doing multipoint evaluation
- Syndrome computation is a transposed multipoint evaluation

Transposing linear algorithms

- A linear algorithm computes a linear map
- Allowed operations: add or multiply by a constant

Transposing linear algorithms

- A linear algorithm computes a linear map
- Allowed operations: add or multiply by a constant

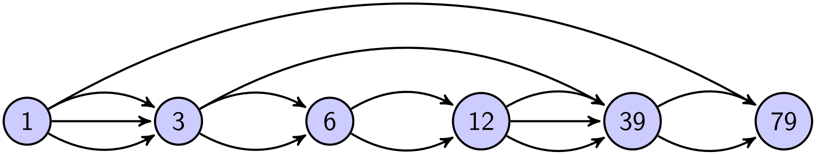
Example: An addition chain for 79

$$\begin{tikzcd}
1 \arrow[r, bend left] \arrow[r] \arrow[r, bend right] \arrow[rrrrr, bend left=40] & 3 \arrow[r, bend left] \arrow[r, bend right] \arrow[rrr, bend left=35] & 6 \arrow[r, bend left] \arrow[r, bend right] & 12 \arrow[r, bend left] \arrow[r] \arrow[r, bend right] & 39 \arrow[r, bend left] \arrow[r, bend right] & 79
\end{tikzcd}$$

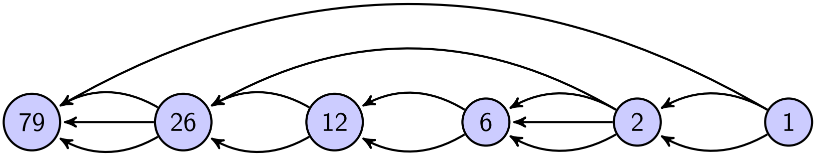
Transposing linear algorithms

- A linear algorithm computes a linear map
- Allowed operations: add or multiply by a constant

Example: An addition chain for 79



By reversing the edges, we get another addition chain for 79:



A larger example

- A linear map: $a_0, a_1 \to a_0b_0, a_0b_1 + a_1b_0, a_1b_1$

$$\begin{tikzcd}
\mathsf{in}_1 = a_0 \arrow[rr, "b_0"] \arrow[dr] & & a_0b_0 \arrow[r] \arrow[d] & \mathsf{out}_1 = a_0b_0 \\
& a_0 + a_1 \arrow[r, "b_0 + b_1"] & \mathsf{out}_2 = a_0b_1 + a_1b_0 & \\
\mathsf{in}_2 = a_1 \arrow[rr, "b_1"] \arrow[ur] & & a_1b_1 \arrow[r] \arrow[u] & \mathsf{out}_3 = a_1b_1
\end{tikzcd}$$

A larger example

- A linear map: $a_0, a_1 \to a_0b_0, a_0b_1 + a_1b_0, a_1b_1$

$$\begin{tikzcd}
\mathsf{in}_1 = a_0 \arrow[rr, "b_0"] \arrow[dr] & & a_0b_0 \arrow[r] \arrow[d] & \mathsf{out}_1 = a_0b_0 \\
& a_0 + a_1 \arrow[r, "b_0 + b_1"] & \mathsf{out}_2 = a_0b_1 + a_1b_0 & \\
\mathsf{in}_2 = a_1 \arrow[rr, "b_1"] \arrow[ur] & & a_1b_1 \arrow[r] \arrow[u] & \mathsf{out}_3 = a_1b_1
\end{tikzcd}$$

- Reversing the edges: $c_0, c_1, c_2 \to b_0c_0 + b_1c_1, b_0c_1 + b_1c_2$

$$\begin{tikzcd}
\mathsf{out}_1 = b_0c_0 + b_1c_1 & & c_0 + c_1 \arrow[ll, "b_0"'] & \mathsf{in}_1 = c_0 \arrow[l] \\
& (b_0 + b_1)c_1 \arrow[ul] \arrow[dl] & \mathsf{in}_2 = c_1 \arrow[l, "b_0 + b_1"'] \arrow[u] \arrow[d] & \\
\mathsf{out}_2 = b_0c_1 + b_1c_2 & & c_1 + c_2 \arrow[ll, "b_1"'] & \mathsf{in}_3 = c_2 \arrow[l]
\end{tikzcd}$$

What did we just do?

- The original linear map:

$$\begin{pmatrix} a_0b_0 \\ a_0b_1 + a_1b_0 \\ a_1b_1 \end{pmatrix} = \begin{pmatrix} b_0 & 0 \\ b_1 & b_0 \\ 0 & b_1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$$

- The transposed map:

$$\begin{pmatrix} b_0c_0 + b_1c_1 \\ b_0c_1 + b_1c_2 \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & 0 \\ 0 & b_0 & b_1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix}$$

What did we just do?

- The original linear map:

$$\begin{pmatrix} a_0 b_0 \\ a_0 b_1 + a_1 b_0 \\ a_1 b_1 \end{pmatrix} = \begin{pmatrix} b_0 & 0 \\ b_1 & b_0 \\ 0 & b_1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$$

- The transposed map:

$$\begin{pmatrix} b_0 c_0 + b_1 c_1 \\ b_0 c_1 + b_1 c_2 \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & 0 \\ 0 & b_0 & b_1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix}$$

- Reversing the edges automatically gives an algorithm for the transposed map
- This is called the *transposition principle*

What did we just do?

- The original linear map:

$$\begin{pmatrix} a_0b_0 \\ a_0b_1 + a_1b_0 \\ a_1b_1 \end{pmatrix} = \begin{pmatrix} b_0 & 0 \\ b_1 & b_0 \\ 0 & b_1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}$$

- The transposed map:

$$\begin{pmatrix} b_0c_0 + b_1c_1 \\ b_0c_1 + b_1c_2 \end{pmatrix} = \begin{pmatrix} b_0 & b_1 & 0 \\ 0 & b_0 & b_1 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix}$$

- Reversing the edges automatically gives an algorithm for the transposed map
- This is called the *transposition principle*
- Preserves number of multiplications
- References: Fiduccia 1972, Bordewijk 1956, Lupanov 1956

Transposing the additive FFT

The naive approach

- Idea: Compute syndrome by transposing the additive FFT
- Start with additive FFT program (sequence of additions and constant multiplications)
- Convert to directed acyclic graph (rename variables to remove cycles)
- Reverse edges, convert to C program
- Compile with gcc

Transposing the additive FFT

The naive approach

- Idea: Compute syndrome by transposing the additive FFT
- Start with additive FFT program (sequence of additions and constant multiplications)
- Convert to directed acyclic graph (rename variables to remove cycles)
- Reverse edges, convert to C program
- Compile with gcc
- Problems:
  - Huge program (all loops and function calls removed)

Transposing the additive FFT

The naive approach

- Idea: Compute syndrome by transposing the additive FFT
- Start with additive FFT program (sequence of additions and constant multiplications)
- Convert to directed acyclic graph (rename variables to remove cycles)
- Reverse edges, convert to C program
- Compile with gcc
- Problems:
    - Huge program (all loops and function calls removed)
    - At $m = 13$ or $m = 14$ gcc runs out of memory

Transposing the additive FFT

The naive approach

- Idea: Compute syndrome by transposing the additive FFT
- Start with additive FFT program (sequence of additions and constant multiplications)
- Convert to directed acyclic graph (rename variables to remove cycles)
- Reverse edges, convert to C program
- Compile with gcc
- Problems:
  - Huge program (all loops and function calls removed)
  - At $m = 13$ or $m = 14$ gcc runs out of memory
  - Can use better register allocators, but the program is still huge

Transposing the additive FFT

A better approach

- Analyze structure of additive FFT A: B, A_1, A_2, C
- A_1, A_2 are recursive calls

Transposing the additive FFT

A better approach

- Analyze structure of additive FFT A: B, A_1, A_2, C
- A_1, A_2 are recursive calls
- Transposition has structure C^T, A_2^T, A_1^T, B^T
- Use recursive calls to reduce code size

Secret permutations

- FFT evaluates f at elements in *standard order*
- We need output in a secret order
- Same problem for input of transposed FFT
- Similar problem during key generation (secret random permutation)

Secret permutations

- FFT evaluates f at elements in *standard order*
- We need output in a secret order
- Same problem for input of transposed FFT
- Similar problem during key generation (secret random permutation)
- Typical solution for permutation π: load from position i, store at position $\pi(i)$

Secret permutations

- FFT evaluates f at elements in *standard order*
- We need output in a secret order
- Same problem for input of transposed FFT
- Similar problem during key generation (secret random permutation)
- Typical solution for permutation π: load from position i, store at position $\pi(i)$
- This leaks through timing information
- We need to apply a secret permutation in constant time

Secret permutations

- FFT evaluates f at elements in *standard order*
- We need output in a secret order
- Same problem for input of transposed FFT
- Similar problem during key generation (secret random permutation)
- Typical solution for permutation π: load from position i, store at position $\pi(i)$
- This leaks through timing information
- We need to apply a secret permutation in constant time
- Solution: sorting networks

Sorting networks

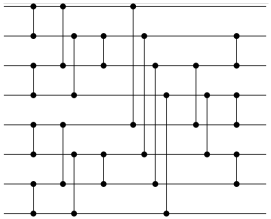
A *sorting network* sorts an array S of elements by using a sequence of *comparators*.

- A comparator can be expressed by a pair of indices (i, j).
- A comparator swaps $S[i]$ and $S[j]$ if $S[i] > S[j]$.

Sorting networks

A *sorting network* sorts an array S of elements by using a sequence of *comparators*.

- A comparator can be expressed by a pair of indices (i, j).
- A comparator swaps $S[i]$ and $S[j]$ if $S[i] > S[j]$.
- Efficient sorting network: Batcher sort (Batcher, 1968)



Batcher sorting network for sorting 8 elements
http://en.wikipedia.org/wiki/Batcher%27s_sort

Permuting by sorting

Example

Computing b_3, b_2, b_1 from b_1, b_2, b_3 can be done by sorting the key-value pairs $(3, b_1), (2, b_2), (1, b_3)$ the output is $(1, b_3), (2, b_2), (3, b_1)$

Permuting by sorting

Example

Computing b_3, b_2, b_1 from b_1, b_2, b_3 can be done by sorting the key-value pairs $(3, b_1), (2, b_2), (1, b_3)$ the output is $(1, b_3), (2, b_2), (3, b_1)$

- All the output bits of $>$ comparisons only depend on the secret permutation
- Those bits can be precomputed during key generation

Permuting by sorting

Example

Computing b_3, b_2, b_1 from b_1, b_2, b_3 can be done by sorting the key-value pairs $(3, b_1), (2, b_2), (1, b_3)$ the output is $(1, b_3), (2, b_2), (3, b_1)$

- All the output bits of $>$ comparisons only depend on the secret permutation
- Those bits can be precomputed during key generation
- Do conditional swap of $b[i]$ and $b[j]$ with condition bit c as

$$y \leftarrow b[i] \oplus b[j]; \quad y \leftarrow cy; \quad b[i] \leftarrow b[i] \oplus y; \quad b[j] \leftarrow b[j] \oplus y;$$

Permuting by sorting

Example

Computing b_3, b_2, b_1 from b_1, b_2, b_3 can be done by sorting the key-value pairs $(3, b_1), (2, b_2), (1, b_3)$ the output is $(1, b_3), (2, b_2), (3, b_1)$

- All the output bits of $>$ comparisons only depend on the secret permutation
- Those bits can be precomputed during key generation
- Do conditional swap of $b[i]$ and $b[j]$ with condition bit c as

$$y \leftarrow b[i] \oplus b[j]; \quad y \leftarrow cy; \quad b[i] \leftarrow b[i] \oplus y; \quad b[j] \leftarrow b[j] \oplus y;$$

- Possibly better than Batcher sort: Beneš permutation network (work in progress)

Results

Throughput cycles on Ivy Bridge

- Input secret permutation: 8622
- Syndrome computation: 20846
- Berlekamp-Massey: 7714
- Root finding: 14794
- Output secret permutation: 8520
- Total: **60493**

Results

Throughput cycles on Ivy Bridge

- Input secret permutation: 8622
- Syndrome computation: 20846
- Berlekamp-Massey: 7714
- Root finding: 14794
- Output secret permutation: 8520
- Total: **60493**
- These are amortized cycle counts across 256 parallel computations

Results

Throughput cycles on Ivy Bridge

- Input secret permutation: 8622
- Syndrome computation: 20846
- Berlekamp-Massey: 7714
- Root finding: 14794
- Output secret permutation: 8520
- Total: **60493**
- These are amortized cycle counts across 256 parallel computations
- All computations with full timing-attack protection!

Comparison

Public-key decryption speeds from eBATS

- `ntruees787ep1`: 700512 cycles
- `mceliece:` 1219344 cycles
- `ronald1024:` 1340040 cycles
- `ronald3072:` 16052564 cycles

Comparison

Public-key decryption speeds from eBATS

- `ntruees787ep1`: 700512 cycles
- `mceliece`: 1219344 cycles
- `ronald1024`: 1340040 cycles
- `ronald3072`: 16052564 cycles

Diffie-Hellman shared-secret speeds from eBATS

- `gls254`: 77468 cycles
- `kumfp127g` 116944 cycles
- `curve25519`: 182632 cycles

More results

CFS code-based signatures

- Signature scheme introduced by Courtois, Finiasz, and Sendrier in 2001
- Verification is very fast
- Previous speed for signing: $\approx 4.2 \cdot 10^9$ cycles on Intel Westmere (at 80 bits of security, no timing-attack protection)
- Our new results:
  - Start with the same parameters
  - Apply bitslicing of field arithmetic
  - Convert all algorithms to constant time

More results

CFS code-based signatures

- Signature scheme introduced by Courtois, Finiasz, and Sendrier in 2001
- Verification is very fast
- Previous speed for signing: $\approx 4.2 \cdot 10^9$ cycles on Intel Westmere (at 80 bits of security, no timing-attack protection)
- Our new results:
  - Start with the same parameters
  - Apply bitslicing of field arithmetic
  - Convert all algorithms to constant time
  - Our speed: $0.425 \cdot 10^9$ cycles in Intel Ivy Bridge
  - This is latency, no batching required

Should you use McBits?

- McBits with the example parameters offers 128 bits of security
- Conservative design, we believe it's safe for use

Should you use McBits?

- McBits with the example parameters offers 128 bits of security
- Conservative design, we believe it's safe for use
- Problems (marketing department is going to kill me):
  - Large public-key size ($\approx 250\,$KB)

Should you use McBits?

- McBits with the example parameters offers 128 bits of security
- Conservative design, we believe it's safe for use
- Problems (marketing department is going to kill me):
  - Large public-key size (≈ 250 KB)
  - Record-setting performance only for large batches
  - Challenge: Apply optimization techniques (additive FFT, etc.) without massive batching, but still with constant running time.

Should you use McBits?

- McBits with the example parameters offers 128 bits of security
- Conservative design, we believe it's safe for use
- Problems (marketing department is going to kill me):
  - Large public-key size ($\approx 250\,\mathrm{KB}$)
  - Record-setting performance only for large batches
  - Challenge: Apply optimization techniques (additive FFT, etc.) without massive batching, but still with constant running time.
  - Software not *yet* available

Should you use McBits?

- McBits with the example parameters offers 128 bits of security
- Conservative design, we believe it's safe for use
- Problems (marketing department is going to kill me):
  - Large public-key size (≈ 250 KB)
  - Record-setting performance only for large batches
  - Challenge: Apply optimization techniques (additive FFT, etc.) without massive batching, but still with constant running time.
  - Software not *yet* available
- I would not consider CFS really practical
- Main concerns (aside from performance): Only 80 bits of security, 20 MB public key

Should you use McBits?

- McBits with the example parameters offers 128 bits of security
- Conservative design, we believe it's safe for use
- Problems (marketing department is going to kill me):
  - Large public-key size (≈ 250 KB)
  - Record-setting performance only for large batches
  - Challenge: Apply optimization techniques (additive FFT, etc.) without massive batching, but still with constant running time.
  - Software not *yet* available
- I would not consider CFS really practical
- Main concerns (aside from performance): Only 80 bits of security, 20 MB public key
- Estimates for 120 bits of security: ≈ 100 times slower signing, ≈ 500 MB public key

References

- Daniel J. Bernstein, Tung Chou, and Peter Schwabe. *McBits: fast constant-time code-based cryptography.*, CHES 2013. `http://cryptojedi.org/papers/#mcbits`
- Software will be online (public domain), for example, at `http://cryptojedi.org/crypto/#mcbits`