Implementing post-quantum cryptography

Peter Schwabe

Radboud University, Nijmegen, The Netherlands

June 28, 2018

PQCRYPTO Mini-School 2018, Taipei, Taiwan

Part I: How to make software secure

Timing Attacks

General idea of those attacks

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

Timing Attacks

General idea of those attacks

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

Two kinds of remote. . .

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

Timing Attacks

General idea of those attacks

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

Two kinds of remote…

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

Timing Attacks

General idea of those attacks

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

Two kinds of remote. . .

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

Problem No. 1

```
if(secret)
{
  do_A();
}
else
{
  do_B();
}
```

Examples

- Square-and-multiply (or double-and-add):

  “if s is one: multiply”

Examples

- Square-and-multiply (or double-and-add):

  “if s is one: multiply”

- Modular reduction:

  “if $a > q$: subtract q from a”

Examples

- Square-and-multiply (or double-and-add):

  “if s is one: multiply”

- Modular reduction:

  “if $a > q$: subtract q from a”

- Rejection sampling:

  “if $a < q$: accept a”

Examples

- Square-and-multiply (or double-and-add):

  “if s is one: multiply”

- Modular reduction:

  “if $a > q$: subtract q from a”

- Rejection sampling:

  “if $a < q$: accept a”

- Byte-array (tag) comparison:

  “if $a[i] \neq b[i]$: return”

Examples

- Square-and-multiply (or double-and-add):

  “if s is one: multiply”

- Modular reduction:

  “if $a > q$: subtract q from a”

- Rejection sampling:

  “if $a < q$: accept a”

- Byte-array (tag) comparison:

  “if $a[i] \neq b[i]$: return”

- Sorting and permuting:

  “if $a < b$: branch into subroutine”

Eliminating branches

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

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

Eliminating branches

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

```
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 code like this?

```
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 code like this?

```
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

Problem No. 2

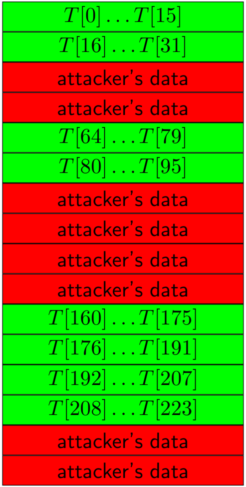
`table[secret]`

Timing leakage part II

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

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

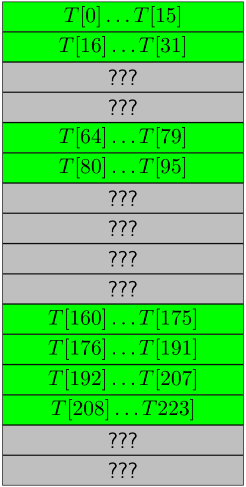
Timing leakage part II



$T[0] \ldots T[15]$
$T[16] \ldots T[31]$
attacker's data
attacker's data
$T[64] \ldots T[79]$
$T[80] \ldots T[95]$
attacker's data
attacker's data
attacker's data
attacker's data
$T[160] \ldots T[175]$
$T[176] \ldots T[191]$
$T[192] \ldots T[207]$
$T[208] \ldots T[223]$
attacker's data
attacker's data

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

Timing leakage part II



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

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

???

???

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

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

???

???

???

???

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

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

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

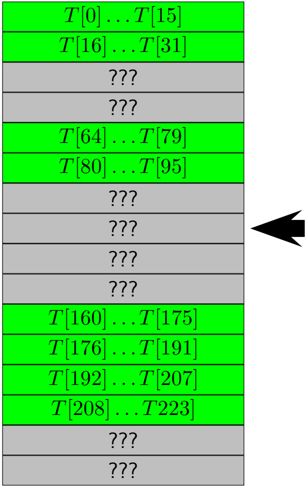
$T[208]\dots T223]$

???

???

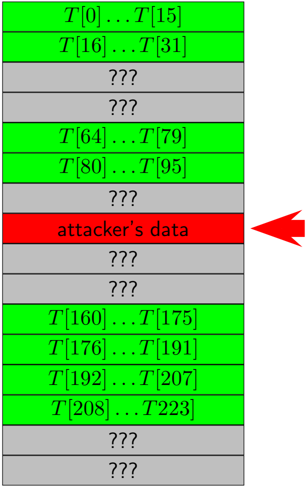
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again

Timing leakage part II



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

Timing leakage part II



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

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

???

???

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

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

???

attacker's data

???

???

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

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

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

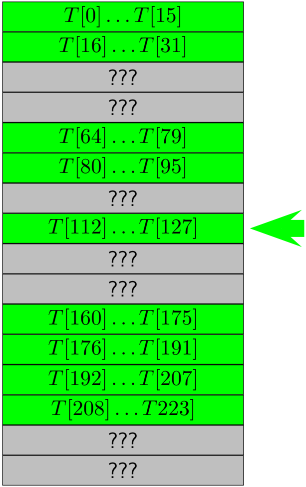
$T[208]\dots T223]$

???

???

- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads his data:
  - Fast: cache hit (crypto did not just load from this line)

Timing leakage part II



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

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

???

???

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

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

???

$T[112]\dots T[127]$

???

???

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

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

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

$T[208]\dots T223]$

???

???

- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads his data:
  - Fast: cache hit (crypto did not just load from this line)
  - Slow: cache miss (crypto just loaded from this line)

The general case

Loads from and stores to addresses that depend on secret data leak secret data.

“Countermeasure”

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

“Countermeasure”

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

“Countermeasure”

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

"Countermeasure"

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

“Countermeasure”

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

“Countermeasure”

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

“Countermeasure”

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

“Countermeasure”

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

“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe… or are they?
- Bernstein, 2005: *“Does this guarantee constant-time S-box lookups? No!”*
- Osvik, Shamir, Tromer, 2006: *“This is insufficient on processors which leak low address bits”*
- Reasons:
    - Cache-bank conflicts
    - Failed store-to-load forwarding
    - …
- OpenSSL is using it in `BN_mod_exp_mont_consttime`
- Brickell (Intel), 2011: yeah, it’s fine as a countermeasure
- Bernstein, Schwabe, 2013: Demonstrate timing variability for access within one cache line
- Yarom, Genkin, Heninger: CacheBleed attack *“is able to recover both 2048-bit and 4096-bit RSA secret keys from OpenSSL 1.0.2f running on Intel Sandy Bridge processors after observing only 16,000 secret-key operations (decryption, signatures).”*

Countermeasure

```
uint32_t table[TABLE_LENGTH];

uint32_t lookup(size_t pos)
{
  size_t i;
  int b;
  uint32_t r = table[0];
  for(i=1;i<TABLE_LENGTH;i++)
  {
    b = (i == pos);
    cmov(&r, &table[i], b); // See "eliminating branches"
  }
  return r;
}
```

Countermeasure

```
uint32_t table[TABLE_LENGTH];

uint32_t lookup(size_t pos)
{
  size_t i;
  int b;
  uint32_t r = table[0];
  for(i=1;i<TABLE_LENGTH;i++)
  {
    b = (i == pos); /* DON'T! Compiler may do funny things! */
    cmov(&r, &table[i], b);
  }
  return r;
}
```

Countermeasure

```
uint32_t table[TABLE_LENGTH];

uint32_t lookup(size_t pos)
{
  size_t i;
  int b;
  uint32_t r = table[0];
  for(i=1;i<TABLE_LENGTH;i++)
  {
    b = isequal(i, pos);
    cmov(&r, &table[i], b);
  }
  return r;
}
```

Countermeasure, part 2

```
int isequal(uint32_t a, uint32_t b)
{
  size_t i; uint32_t r = 0;
  unsigned char *ta = (unsigned char *)&a;
  unsigned char *tb = (unsigned char *)&b;
  for(i=0;i<sizeof(uint32_t);i++)
  {
    r |= (ta[i] ^ tb[i]);
  }
  r = (-r) >> 31;
  return (int)(1-r);
}
```

Part II: How to make software fast

Vector computations

Scalar computation

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

Vectorized computation

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

Vector computations

Scalar computation

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

Vectorized computation

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

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

Vector computations

Scalar computation

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

Vectorized computation

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

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

Vector computations

Scalar computation

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

Vectorized computation

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

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

Why is this so great?

- Consider the Intel Skylake processor

Why is this so great?

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

Why is this so great?

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

Why is this so great?

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

Why is this so great?

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

Take-home message

“Big multipliers are pre-quantum,
vectorization is post-quantum”

Standard-lattice-based schemes

- Standard-lattices operate on matrices over $\mathbb{Z}_q$, for “small” q
- These are trivially vectorizable
- So trivial that even compilers may do it!

Standard-lattice-based schemes

- Standard-lattices operate on matrices over $\mathbb{Z}_q$, for "small" q
- These are trivially vectorizable
- So trivial that even compilers may do it!
- Standard-lattice-based signatures (e.g., Bai-Galbraith):
  - Multiple attempts for signing (rejection sampling)
  - Each attempt: compute $\mathbf{Av}$ for fixed $\mathbf{A}$

Standard-lattice-based schemes

- Standard-lattices operate on matrices over $\mathbb{Z}_q$, for "small" q
- These are trivially vectorizable
- So trivial that even compilers may do it!
- Standard-lattice-based signatures (e.g., Bai-Galbraith):
  - Multiple attempts for signing (rejection sampling)
  - Each attempt: compute $\mathbf{Av}$ for fixed $\mathbf{A}$
- More efficient:
  - Compute multiple products $\mathbf{Av_i}$
  - Typically ignore some results

Standard-lattice-based schemes

- Standard-lattices operate on matrices over $\mathbb{Z}_q$, for "small" q
- These are trivially vectorizable
- So trivial that even compilers may do it!
- Standard-lattice-based signatures (e.g., Bai-Galbraith):
  - Multiple attempts for signing (rejection sampling)
  - Each attempt: compute $\mathbf{Av}$ for fixed $\mathbf{A}$
- More efficient:
  - Compute multiple products $\mathbf{Av_i}$
  - Typically ignore some results
- Reason: reuse coefficients of $\mathbf{A}$ in cache

Structured lattices

- Structured lattices (NTRU, RLWE, MLWE) work with polynomials
- Most important operation: multiply polynomials
- Obvious question: How do we vectorize polynomial multiplication?

Structured lattices

- Structured lattices (NTRU, RLWE, MLWE) work with polynomials
- Most important operation: multiply polynomials
- Obvious question: How do we vectorize polynomial multiplication?
- Let's take an example:

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

Structured lattices

- Structured lattices (NTRU, RLWE, MLWE) work with polynomials
- Most important operation: multiply polynomials
- Obvious question: How do we vectorize polynomial multiplication?
- Let's take an example:

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

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

Structured lattices

- Structured lattices (NTRU, RLWE, MLWE) work with polynomials
- Most important operation: multiply polynomials
- Obvious question: How do we vectorize polynomial multiplication?
- Let's take an example:

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

- 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?

Structured lattices

- Structured lattices (NTRU, RLWE, MLWE) work with polynomials
- Most important operation: multiply polynomials
- Obvious question: How do we vectorize polynomial multiplication?
- Let's take an example:

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

- 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?
- Looks like we need to *shuffle* a lot!

Karatsuba and Toom

- Our polynomials have many more coefficients (say, 256–1024)
- Idea: use Karatsuba's trick:
  - consider $n = 2k$-coefficient polynomials f and g
  - Split multiplication $f \cdot g$ into 3 half-size multiplications

$$\begin{aligned} & (f_\ell + X^k f_h) \cdot (g_\ell + X^k g_h) \\ = \; & f_\ell g_\ell + X^k (f_\ell g_h + f_h g_\ell) + X^n f_h g_h \\ = \; & f_\ell g_\ell + X^k ((f_\ell + f_h)(g_\ell + g_h) - f_\ell g_\ell - f_h g_h) + X^n f_h g_h \end{aligned}$$

Karatsuba and Toom

- Our polynomials have many more coefficients (say, 256–1024)
- Idea: use Karatsuba's trick:
  - consider $n = 2k$-coefficient polynomials f and g
  - Split multiplication $f \cdot g$ into 3 half-size multiplications

$$
\begin{aligned}
&(f_\ell + X^k f_h) \cdot (g_\ell + X^k g_h) \\
= \; & f_\ell g_\ell + X^k (f_\ell g_h + f_h g_\ell) + X^n f_h g_h \\
= \; & f_\ell g_\ell + X^k ((f_\ell + f_h)(g_\ell + g_h) - f_\ell g_\ell - f_h g_h) + X^n f_h g_h
\end{aligned}
$$

- Apply recursively to obtain 9 quarter-size multiplications, 27 eighth-size multiplications etc.

Karatsuba and Toom

- Our polynomials have many more coefficients (say, 256–1024)
- Idea: use Karatsuba's trick:
  - consider $n = 2k$-coefficient polynomials f and g
  - Split multiplication $f \cdot g$ into 3 half-size multiplications

$$\begin{aligned}
&(f_\ell + X^k f_h) \cdot (g_\ell + X^k g_h) \\
= \; & f_\ell g_\ell + X^k (f_\ell g_h + f_h g_\ell) + X^n f_h g_h \\
= \; & f_\ell g_\ell + X^k ((f_\ell + f_h)(g_\ell + g_h) - f_\ell g_\ell - f_h g_h) + X^n f_h g_h
\end{aligned}$$

- Apply recursively to obtain 9 quarter-size multiplications, 27 eighth-size multiplications etc.
- Generalization: Toom-Cook. Obtain, e.g., 5 third-size multiplications
- Split into sufficiently many "small" multiplications, vectorize across those

Transposing/Interleaving

- Small example: compute $a \cdot b$, $c \cdot d$, $e \cdot f$, $g \cdot h$
- Each factor with 3 coefficients, e.g., $a = a_0 + a_1X + a_2X^2$

Transposing/Interleaving

- Small example: compute $a \cdot b$, $c \cdot d$, $e \cdot f$, $g \cdot h$
- Each factor with 3 coefficients, e.g., $a = a_0 + a_1 X + a_2 X^2$
- Coefficients in memory:

`a0, a1, a2, b0, b1, b2, c0,..., h1, h2`

Transposing/Interleaving

- Small example: compute $a \cdot b$, $c \cdot d$, $e \cdot f$, $g \cdot h$
- Each factor with 3 coefficients, e.g., $a = a_0 + a_1 X + a_2 X^2$
- Coefficients in memory:

```
a0, a1, a2, b0, b1, b2, c0,..., h1, h2
```

- Problem:
  - Vector loads will yield

$$v_0 = (a_0, a_1, a_2, b_0) \qquad \dots \qquad v_6 = (g_2, h_0, h_1, h_2)$$

  - However, we need

$$v_0 = (a_0, c_0, e_0, h_0) \qquad \dots \qquad v_6 = (b_2, d_2, f_2, g_2)$$

Transposing/Interleaving

- Small example: compute $a \cdot b$, $c \cdot d$, $e \cdot f$, $g \cdot h$
- Each factor with 3 coefficients, e.g., $a = a_0 + a_1 X + a_2 X^2$
- Coefficients in memory:

  `a0, a1, a2, b0, b1, b2, c0,..., h1, h2`

- Problem:
  - Vector loads will yield

    $$v_0 = (a_0, a_1, a_2, b_0) \qquad \ldots \qquad v_6 = (g_2, h_0, h_1, h_2)$$

  - However, we need

    $$v_0 = (a_0, c_0, e_0, h_0) \qquad \ldots \qquad v_6 = (b_2, d_2, f_2, g_2)$$

- Solution: transpose data matrix (or interleave words):

  `a0, c0, e0, h0, a1, c1, e1,..., f2, g2`

Two applications of Karatsuba/Toom

Streamlined NTRU Prime 4591^{761}

- Multiply in the ring $\mathcal{R} = \mathbb{Z}_{4591}[X]/(X^{761} - X - 1)$
- Pad input polynomial to 768 coefficients
- 5 levels of Karatsuba: 243 multiplications of 24-coefficient polynomials
- Massively lazy reduction using double-precision floats
- $28\,682$ Haswell cycles for multiplication in $\mathcal{R}$

Two applications of Karatsuba/Toom

Streamlined NTRU Prime 4591^{761}

- Multiply in the ring $\mathcal{R} = \mathbb{Z}_{4591}[X]/(X^{761} - X - 1)$
- Pad input polynomial to 768 coefficients
- 5 levels of Karatsuba: 243 multiplications of 24-coefficient polynomials
- Massively lazy reduction using double-precision floats
- $28\,682$ Haswell cycles for multiplication in $\mathcal{R}$

NTRU-HRSS-KEM

- Multiply in the ring $\mathcal{R} = \mathbb{Z}_{8192}[X]/(X^{701} - 1)$
- Use Toom-Cook to split into 7 quarter-size, then 2 levels of Karatsuba
- Obtain 63 multiplications of 44-coefficient polynomials
- $11\,722$ Haswell cycles for multiplication in $\mathcal{R}$

We can do better: NTTs

- Many LWE/MLWE systems use very specific parameters:
    - Work in polynomial ring $\mathcal{R} = \mathbb{Z}_q[X]/(X^n + 1)$
    - Choose n a power of 2
    - Choose q prime, s.t. $2n$ divides $(q - 1)$

We can do better: NTTs

- Many LWE/MLWE systems use very specific parameters:
    - Work in polynomial ring $\mathcal{R} = \mathbb{Z}_q[X]/(X^n + 1)$
    - Choose n a power of 2
    - Choose q prime, s.t. $2n$ divides $(q - 1)$
- Examples: NewHope ($n = 1024, q = 12289$), Kyber ($n = 256, q = 7681$)

We can do better: NTTs

- Many LWE/MLWE systems use very specific parameters:
  - Work in polynomial ring $\mathcal{R} = \mathbb{Z}_q[X]/(X^n + 1)$
  - Choose n a power of 2
  - Choose q prime, s.t. $2n$ divides $(q - 1)$
- Examples: NewHope ($n = 1024, q = 12289$), Kyber ($n = 256, q = 7681$)
- Big advantage: fast *negacyclic number-theoretic transform*
- Given $g \in \mathcal{R}$, n-th primitive root of unity ω and $\psi = \sqrt{\omega}$, compute

$$\mathsf{NTT}(g) = \hat{g} = \sum_{i=0}^{n-1} \hat{g}_i X^i, \text{ with}$$

$$\hat{g}_i = \sum_{j=0}^{n-1} \psi^j g_j \omega^{ij},$$

We can do better: NTTs

- Many LWE/MLWE systems use very specific parameters:
  - Work in polynomial ring $\mathcal{R} = \mathbb{Z}_q[X]/(X^n + 1)$
  - Choose n a power of 2
  - Choose q prime, s.t. $2n$ divides $(q - 1)$
- Examples: NewHope ($n = 1024, q = 12289$), Kyber ($n = 256, q = 7681$)
- Big advantage: fast *negacyclic number-theoretic transform*
- Given $g \in \mathcal{R}$, n-th primitive root of unity ω and $\psi = \sqrt{\omega}$, compute

$$\mathsf{NTT}(g) = \hat{g} = \sum_{i=0}^{n-1} \hat{g}_i X^i, \text{ with}$$

$$\hat{g}_i = \sum_{j=0}^{n-1} \psi^j g_j \omega^{ij},$$

- Compute $f \cdot g$ as $\mathsf{NTT}^{-1}(\mathsf{NTT}(f) \circ \mathsf{NTT}(g))$

We can do better: NTTs

- Many LWE/MLWE systems use very specific parameters:
  - Work in polynomial ring $\mathcal{R} = \mathbb{Z}_q[X]/(X^n + 1)$
  - Choose n a power of 2
  - Choose q prime, s.t. $2n$ divides $(q-1)$
- Examples: NewHope ($n = 1024, q = 12289$), Kyber ($n = 256, q = 7681$)
- Big advantage: fast *negacyclic number-theoretic transform*
- Given $g \in \mathcal{R}$, n-th primitive root of unity ω and $\psi = \sqrt{\omega}$, compute

$$\mathsf{NTT}(g) = \hat{g} = \sum_{i=0}^{n-1} \hat{g}_i X^i, \text{ with}$$

$$\hat{g}_i = \sum_{j=0}^{n-1} \psi^j g_j \omega^{ij},$$

- Compute $f \cdot g$ as $\mathsf{NTT}^{-1}(\mathsf{NTT}(f) \circ \mathsf{NTT}(g))$
- NTT^{-1} is essentially the same computation as NTT

Zooming into the NTT

- FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1X + \cdots + f_{n-1}X^{n-1}$ at all n-th roots of unity
- Divide-and-conquer approach
  - Write polynomial f as $f_0(X^2) + Xf_1(X^2)$

Zooming into the NTT

- FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1X + \cdots + f_{n-1}X^{n-1}$ 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

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

Zooming into the NTT

- FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1 X + \cdots + f_{n-1} X^{n-1}$ 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

Zooming into the NTT

- FFT in a finite field
- Evaluate polynomial $f = f_0 + f_1X + \cdots + f_{n-1}X^{n-1}$ 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
- Apply recursively through $\log n$ levels

Vectorizing the NTT

- First thing to do: replace recursion by iteration
- Loop over $\log n$ levels with $n/2$ "butterflies" each
- Butterfly on level k:
  - Pick up f_i and f_{i+2^k}
  - Multiply f_{i+2^k} by a power of ω to obtain t
  - Compute $f_{i+2^k} \leftarrow a_i - t$
  - Compute $f_i \leftarrow a_i + t$
- All $n/2$ butterflies on one level are independent
- Vectorize across those butterflies

Vectorized NTT results

- Güneysu, Oder, Pöppelmann, Schwabe, 2013:
    - 4480 Sandy Bridge cycles ($n = 512$, 23-bit q)
    - Use double-precision floats to represent coefficients

Vectorized NTT results

- Güneysu, Oder, Pöppelmann, Schwabe, 2013:
  - 4480 Sandy Bridge cycles ($n = 512$, 23-bit q)
  - Use double-precision floats to represent coefficients
- Alkim, Ducas, Pöppelmann, Schwabe, 2016:
  - 8448 Haswell cycles ($n = 1024$, 14-bit q)
  - Still use doubles

Vectorized NTT results

- Güneysu, Oder, Pöppelmann, Schwabe, 2013:
  - 4480 Sandy Bridge cycles ($n = 512$, 23-bit q)
  - Use double-precision floats to represent coefficients
- Alkim, Ducas, Pöppelmann, Schwabe, 2016:
  - 8448 Haswell cycles ($n = 1024$, 14-bit q)
  - Still use doubles
- Longa, Naehrig, 2016:
  - 9100 Haswell cycles ($n = 1024$, 14-bit q)
  - Uses vectorized integer arithmetic

Vectorized NTT results

- Güneysu, Oder, Pöppelmann, Schwabe, 2013:
  - 4480 Sandy Bridge cycles ($n = 512$, 23-bit q)
  - Use double-precision floats to represent coefficients
- Alkim, Ducas, Pöppelmann, Schwabe, 2016:
  - 8448 Haswell cycles ($n = 1024$, 14-bit q)
  - Still use doubles
- Longa, Naehrig, 2016:
  - 9100 Haswell cycles ($n = 1024$, 14-bit q)
  - Uses vectorized integer arithmetic
- Seiler, 2018:
  - 2784 Haswell cycles ($n = 1024$, 14-bit q)
  - 460 Haswell cycles ($n = 256$, 13-bit q)
  - Uses vectorized integer arithmetic

How about hashing?

- NTT-based multiplication is **fast**
- Consequence: “symmetric” parts in lattice-based crypto becomes significant overhead!
- Most important: hashes and XOFs

How about hashing?

- NTT-based multiplication is **fast**
- Consequence: "symmetric" parts in lattice-based crypto becomes significant overhead!
- Most important: hashes and XOFs
- Typical hash construction:
  - Process message in blocks
  - Each block modifies an internal state
  - Cannot vectorize across blocks

How about hashing?

- NTT-based multiplication is **fast**
- Consequence: “symmetric” parts in lattice-based crypto becomes significant overhead!
- Most important: hashes and XOFs
- Typical hash construction:
    - Process message in blocks
    - Each block modifies an internal state
    - Cannot vectorize across blocks
- Idea: Vectorize internal processing (permutation or compression function)
- Two problems:
    - Often strong dependencies between instructions
    - Need limited instruction-level parallelism for pipelining

How about hashing?

- NTT-based multiplication is **fast**
- Consequence: “symmetric” parts in lattice-based crypto becomes significant overhead!
- Most important: hashes and XOFs
- Typical hash construction:
  - Process message in blocks
  - Each block modifies an internal state
  - Cannot vectorize across blocks
- Idea: Vectorize internal processing (permutation or compression function)
- Two problems:
  - Often strong dependencies between instructions
  - Need limited instruction-level parallelism for pipelining
- Consequence: consider designing with parallel hash/XOF calls!

PQCRYPTO $\neq$ Lattices

- So far we've looked at lattices, how about other PQCRYPTO?
- Code-based crypto (and some $\mathcal{MQ}$-based crypto) need binary-field arithmetic
- Typical: operations in $\mathbb{F}_{2^k}$ for $k \in 1, \ldots, 20$

PQCRYPTO $\neq$ Lattices

- So far we've looked at lattices, how about other PQCRYPTO?
- Code-based crypto (and some $\mathcal{MQ}$-based crypto) need binary-field arithmetic
- Typical: operations in $\mathbb{F}_{2^k}$ for $k \in 1, \ldots, 20$
- Most architectures don't support this efficiently
- Traditional approach: use lookups (log tables)

PQCRYPTO $\neq$ Lattices

- So far we've looked at lattices, how about other PQCRYPTO?
- Code-based crypto (and some $\mathcal{MQ}$-based crypto) need binary-field arithmetic
- Typical: operations in $\mathbb{F}_{2^k}$ for $k \in 1, \ldots, 20$
- Most architectures don't support this efficiently
- Traditional approach: use lookups (log tables)
- Obvious question: can vector operations help?

Bitslicing

- So far: vectors of bytes, 32-bit words, floats,. . .
- Consider now vectors of bits

Bitslicing

- So far: vectors of bytes, 32-bit words, floats,...
- Consider now vectors of bits
- Perform arithmetic on those vectors using `XOR`, `AND`, `OR`
- "Simulate hardware implemenations in software"

Bitslicing

- So far: vectors of bytes, 32-bit words, floats,...
- Consider now vectors of bits
- Perform arithmetic on those vectors using `XOR`, `AND`, `OR`
- "Simulate hardware implemenations in software"
- Technique was introduced by Biham in 1997 for DES
- Bitslicing works for every algorithm
- *Efficient* bitslicing needs a huge amount of data-level parallelism

Bitslicing binary polynomials

4-coefficient binary polynomials

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

4-coefficient bitsliced binary polynomials

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

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

Bitsliced binary-polynomial multiplication

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

void poly4x64_mul(poly7x64 r, const poly4x64 f, const poly4x64 g)
{
  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]);
}
```

McBits (revisited)

- Bernstein, Chou, Schwabe, 2013: High-speed code-based crypto
- Low-level: bitsliced arithmetic in $\mathbb{F}_{2^k}$, $k \in \{11, \ldots, 16\}$

McBits (revisited)

- Bernstein, Chou, Schwabe, 2013: High-speed code-based crypto
- Low-level: bitsliced arithmetic in $\mathbb{F}_{2^k}$, $k \in \{11, \ldots, 16\}$
- Higher level:
    - Additive FFT for efficient root finding
    - Transposed FFT for syndrome computation
    - Batcher sort for random permutations

McBits (revisited)

- Bernstein, Chou, Schwabe, 2013: High-speed code-based crypto
- Low-level: bitsliced arithmetic in $\mathbb{F}_{2^k}$, $k \in \{11, \ldots, 16\}$
- Higher level:
  - Additive FFT for efficient root finding
  - Transposed FFT for syndrome computation
  - Batcher sort for random permutations
- Results:
  - $75\,935\,744$ Ivy Bridge cycles for 256 decodings at ≈ 256-bit pre-quantum security
  - **Not** $75\,935\,744/256 = 296\,624$ cycles for one decoding
  - Reason: Need 256 independent decodings for parallelism

McBits (revisited)

- Bernstein, Chou, Schwabe, 2013: High-speed code-based crypto
- Low-level: bitsliced arithmetic in $\mathbb{F}_{2^k}$, $k \in \{11, \ldots, 16\}$
- Higher level:
  - Additive FFT for efficient root finding
  - Transposed FFT for syndrome computation
  - Batcher sort for random permutations
- Results:
  - $75\,935\,744$ Ivy Bridge cycles for 256 decodings at ≈ 256-bit pre-quantum security
  - **Not** $75\,935\,744/256 = 296\,624$ cycles for one decoding
  - Reason: Need 256 independent decodings for parallelism
- Chou, CHES 2017: use *internal* parallelism
  - Target even higher security (297 bits pre-quantum)
  - Does *not* require independent decryptions
  - Even faster, even when considering throughput

How about $\mathcal{MQ}$?

- Most important operation: evaluate system of quadratic equations
- Massively parallel, efficiently vectorizable

How about $\mathcal{MQ}$?

- Most important operation: evaluate system of quadratic equations
- Massively parallel, efficiently vectorizable
- Distinguish 3 (or 4) different cases, depending on the field
- $\mathbb{F}_{31}$: 16-bit-word vector elements, use integer arithmetic

How about $\mathcal{MQ}$?

- Most important operation: evaluate system of quadratic equations
- Massively parallel, efficiently vectorizable
- Distinguish 3 (or 4) different cases, depending on the field
- $\mathbb{F}_{31}$: 16-bit-word vector elements, use integer arithmetic
- $\mathbb{F}_2/\mathbb{F}_4$: Use bitslicing

How about $\mathcal{MQ}$?

- Most important operation: evaluate system of quadratic equations
- Massively parallel, efficiently vectorizable
- Distinguish 3 (or 4) different cases, depending on the field
- $\mathbb{F}_{31}$: 16-bit-word vector elements, use integer arithmetic
- $\mathbb{F}_2/\mathbb{F}_4$: Use bitslicing
- $\mathbb{F}_{16}/\mathbb{F}_{256}$: Use vector-permute instructions for table lookups
- For $\mathbb{F}_{256}$ use tower-field arithmetic on top of $\mathbb{F}_{16}$

Recent $\mathcal{MQ}$ results

- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe, 2016: 64 eqns in 64 vars over $\mathbb{F}_{31}$: 6616 Haswell cycles

Recent $\mathcal{MQ}$ results

- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe, 2016:
  64 eqns in 64 vars over $\mathbb{F}_{31}$: 6616 Haswell cycles
- Chen, Li, Peng, Yang, Cheng, 2017:
  - 256 eqns in 256 vars over $\mathbb{F}_2$: 92800 Haswell cycles
  - 128 eqns in 128 vars over $\mathbb{F}_4$: 32300 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{16}$: 9600 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{31}$: 8700 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{256}$: 16200 Haswell cycles
  - In particular for $\mathbb{F}_2$ speedups for public inputs

Recent $\mathcal{MQ}$ results

- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe, 2016:
  64 eqns in 64 vars over $\mathbb{F}_{31}$: 6616 Haswell cycles
- Chen, Li, Peng, Yang, Cheng, 2017:
  - 256 eqns in 256 vars over $\mathbb{F}_2$: 92800 Haswell cycles
  - 128 eqns in 128 vars over $\mathbb{F}_4$: 32300 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{16}$: 9600 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{31}$: 8700 Haswell cycles
  - 64 eqns in 64 vars over $\mathbb{F}_{256}$: 16200 Haswell cycles
  - In particular for $\mathbb{F}_2$ speedups for public inputs
- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe, 2017:
  128 eqns in 128 vars over $\mathbb{F}_4$: $17\,558$ Haswell cycles (batched)

Vectorizing hash-based signatures

- I said earlier that hashes are hard to vectorize
- How about hash-based signatures?

Vectorizing hash-based signatures

- I said earlier that hashes are hard to vectorize
- How about hash-based signatures?
- Most speed-critical operation is Winternitz public-key computation
- Compute 67 independent hash chains of length 16 each
- All hashes have the same (short) input length
- This is trivially vectorizable!

Vectorizing hash-based signatures

- I said earlier that hashes are hard to vectorize
- How about hash-based signatures?
- Most speed-critical operation is Winternitz public-key computation
- Compute 67 independent hash chains of length 16 each
- All hashes have the same (short) input length
- This is trivially vectorizable!
- Examples:
  - Oliveira, López, Cabral, 2017: Optimize LMS and XMSS
  - ≈ 10ms for XMSS signing ($h = 20$) on Skylake

Vectorizing hash-based signatures

- I said earlier that hashes are hard to vectorize
- How about hash-based signatures?
- Most speed-critical operation is Winternitz public-key computation
- Compute 67 independent hash chains of length 16 each
- All hashes have the same (short) input length
- This is trivially vectorizable!
- Examples:
  - Oliveira, López, Cabral, 2017: Optimize LMS and XMSS
  - ≈ 10ms for XMSS signing ($h = 20$) on Skylake
  - Bernstein, Hopwood, Hülsing, Lange, Niederhagen, Papachristodoulou, Schneider, Schwabe, Wilcox-O'Hearn, 2015: Optimize SPHINCS
  - Vectorize also Merkle-tree hashes inside HORST computation
  - ≈ 52 Mio cycles for signing on Haswell

Additional benefits

Two things very inefficient to vectorize

1. Variably indexed lookups:

$$v \leftarrow (m[i], m[j], m[k], m[\ell])$$

Additional benefits

Two things very inefficient to vectorize

1. Variably indexed lookups:

$$v \leftarrow (m[i], m[j], m[k], m[\ell])$$

2. Branches

$$v \leftarrow (c[0]?a : b, c[1]?c : d, c[2]?e : f, c[3]?g : h)$$

Additional benefits

Two things very inefficient to vectorize

1. Variably indexed lookups:

$$v \leftarrow (m[i], m[j], m[k], m[\ell])$$

2. Branches

$$v \leftarrow (c[0]?a : b, c[1]?c : d, c[2]?e : f, c[3]?g : h)$$

Rethink algorithms

- Consequence: rethink algorithms without those constructs
- Different approach to thinking algorithms: a lot of fun!

Additional benefits

Two things very inefficient to vectorize

1. Variably indexed lookups:

$$v \leftarrow (m[i], m[j], m[k], m[\ell])$$

2. Branches

$$v \leftarrow (c[0]?a : b, c[1]?c : d, c[2]?e : f, c[3]?g : h)$$

Rethink algorithms

- Consequence: rethink algorithms without those constructs
- Different approach to thinking algorithms: a lot of fun!
- More importantly: eliminates most notorious timing side channels!
- Efficient vectorized implementations are often also "constant-time"

References

- Alkim, Bindel, Buchmann, Dagdelen, Schwabe: *TESLA: Tightly-Secure Efficient Signatures from Standard Lattices.* `https://cryptojedi.org/papers/#tesla` (superseded by `https://eprint.iacr.org/2015/755`)
- Bernstein, Chuengsatiansup, Lange, van Vredendaal: *NTRU Prime: reducing attack surface at low cost.* `http://cr.yp.to/papers.html#ntruprime`
- Hülsing, Rijneveld, Schanck, Schwabe: *High-speed key encapsulation from NTRU.* `https://cryptojedi.org/papers/#ntrukem`

References

- Güneysu, Oder, Pöppelmann, Schwabe: *Software speed records for lattice-based signatures.* `https://cryptojedi.org/papers/#lattisigns`
- Alkim, Ducas, Pöppelmann, Schwabe: *Post-quantum key exchange – a new hope.* `https://cryptojedi.org/papers/#newhope`
- Longa, Naehrig: *Speeding up the Number Theoretic Transform for Faster Ideal Lattice-Based Cryptography.* `https://eprint.iacr.org/2016/504`
- Seiler: *Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography* `https://eprint.iacr.org/2018/039`

References

- Bernstein, Chou, Schwabe: *McBits: fast constant-time code-based cryptography.* `https://cryptojedi.org/papers/#mcbits`
- Chou: *McBits revisited.* `https://eprint.iacr.org/2017/793`

References

- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe: *From 5-pass MQ-based identification to MQ-based signatures.* `https://cryptojedi.org/papers/#mqdss`
- Chen, Li, Peng, Yang, Cheng: *Implementing 128-bit Secure MPKC Signatures.* `https://eprint.iacr.org/2017/636`
- Chen, Hülsing, Rijneveld, Samardjiska, Schwabe: *SOFIA: MQ-based signatures in the QROM.* `https://cryptojedi.org/papers/#sofia`

References

- Oliveira, López, Cabral: *High Performance of Hash-based Signature Schemes* `http://thesai.org/Publications/ViewPaper?Volume=8&Issue=3&Code=IJACSA&SerialNo=58`
- Bernstein, Hopwood, Hülsing, Lange, Niederhagen, Papachristodoulou, Schneider, Schwabe, Wilcox-O'Hearn: *SPHINCS: practical stateless hash-based signatures.* `https://cryptojedi.org/papers/#sphincs`