MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

Kyber – Implementation aspects

August 1, 2024

Three properties

1. Efficiency
2. Security
3. Correctness

1. Efficiency

What I mean by efficiency

"Oh, you mean numbers?!"

—Giulio Malavolta, September 2022

What I mean by efficiency

"Oh, you mean numbers?!"

—Giulio Malavolta, September 2022

- We typically care about 10% speedup
- We may care about every byte of RAM

What I mean by efficiency

"Oh, you mean numbers?!"

—Giulio Malavolta, September 2022

- We typically care about 10% speedup
- We may care about every byte of RAM
- 0.05% of CPU cycles in Meta's data centers are spent on X25519

What I mean by efficiency

"Oh, you mean numbers?!"

—Giulio Malavolta, September 2022

- We typically care about 10% speedup
- We may care about every byte of RAM
- 0.05% of CPU cycles in Meta's data centers are spent on X25519
- Saving a hash of 1KB may save an Internet giant some million USD per year

What I mean by efficiency

"Oh, you mean numbers?!"

—Giulio Malavolta, September 2022

- We typically care about 10% speedup
- We may care about every byte of RAM
- 0.05% of CPU cycles in Meta's data centers are spent on X25519
- Saving a hash of 1KB may save an Internet giant some million USD per year
- Consequence:
  - Crypto is commonly hand-optimized on ASM level
  - Interaction between design and low-level implementation

Parallel computation

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

—Maurice Herlihy: The Multicore Revolution, 2007

Parallel computation

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

—Maurice Herlihy: The Multicore Revolution, 2007

Why multicore does not matter for crypto

- **Crypto is fast**
- > 30000 X25519 shared-key computations on a 3 GHz Skylake
- > 50000 Kyber-768 encapsulations
- ≈ 70000 Kyber-768 decapsulations

Parallel computation

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

—Maurice Herlihy: The Multicore Revolution, 2007

Why multicore does not matter for crypto

- **Crypto is fast**
- > 30000 X25519 shared-key computations on a 3 GHz Skylake
- > 50000 Kyber-768 encapsulations
- ≈ 70000 Kyber-768 decapsulations
- **If you perform only one crypto operation, you don't care**

Parallel computation

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

—Maurice Herlihy: The Multicore Revolution, 2007

Why multicore does not matter for crypto

- **Crypto is fast**
- > 30000 X25519 shared-key computations on a 3 GHz Skylake
- > 50000 Kyber-768 encapsulations
- ≈ 70000 Kyber-768 decapsulations
- **If you perform only one crypto operation, you don't care**
- **Many crypto operations are trivially parallel on multiple cores**

Vector computations

Scalar computation

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

Vectorized computation

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

Vector computations

Scalar computation

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

Vectorized computation

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

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

Vector computations

Scalar computation

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

Vectorized computation

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

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

Vector computations

Scalar computation

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

Vectorized computation

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

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

Performance of vector arithmetic

- Consider the Intel Skylake processor with AVX2

Performance of vector arithmetic

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

Performance of vector arithmetic

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

Performance of vector arithmetic

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

Performance of vector arithmetic

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

The NTT

Core operation in LPR encryption: multiply in $\mathcal{R}_q$

- Schoolbook approach: $\Theta(n^2)$

The NTT

Core operation in LPR encryption: multiply in $\mathcal{R}_q$

- Schoolbook approach: $\Theta(n^2)$
- Karatsuba: $\Theta(n^{\log_2 3})$

The NTT

Core operation in LPR encryption: multiply in $\mathcal{R}_q$

- Schoolbook approach: $\Theta(n^2)$
- Karatsuba: $\Theta(n^{\log_2 3})$
- Approach for multiplication in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $n = 2^m$:

$$\mathbf{a} \cdot \mathbf{b} = \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$$

The NTT

Core operation in LPR encryption: multiply in $\mathcal{R}_q$

- Schoolbook approach: $\Theta(n^2)$
- Karatsuba: $\Theta(n^{\log_2 3})$
- Approach for multiplication in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $n = 2^m$:

$$\mathbf{a} \cdot \mathbf{b} = \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$$

- Number Theoretic Transform (NTT) is discrete FFT
- Complexity $\Theta(n \log n)$
- $\circ$ is “pointwise” multiplication

The NTT

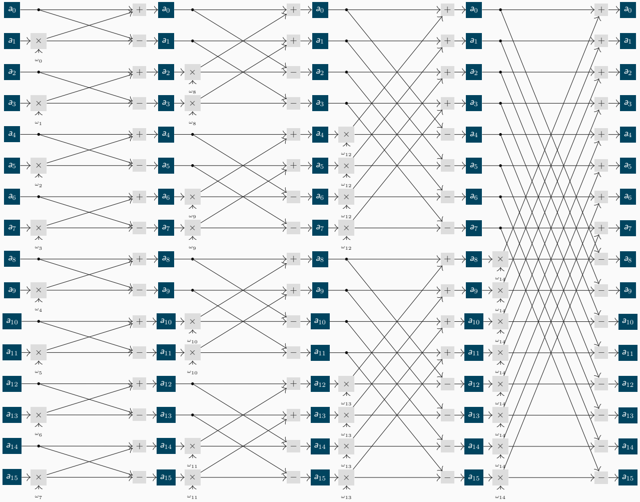
Core operation in LPR encryption: multiply in $\mathcal{R}_q$

- Schoolbook approach: $\Theta(n^2)$
- Karatsuba: $\Theta(n^{\log_2 3})$
- Approach for multiplication in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $n = 2^m$:

$$\mathbf{a} \cdot \mathbf{b} = \mathsf{NTT}^{-1}(\mathsf{NTT}(\mathbf{a}) \circ \mathsf{NTT}(\mathbf{b}))$$

- Number Theoretic Transform (NTT) is discrete FFT
- Complexity $\Theta(n \log n)$
- $\circ$ is "pointwise" multiplication
- Requires that $2n$ divides $q - 1$
- Split $(X^n + 1)$, perform multiplication modulo factors

Structure of (INV)NTT



Picture credit: Matthias Kannwischer

- $\log n$ layers of "butterfly operations"
- Each layer has $n/2$ butterflies
- On most layers straight-forwardly vectorizable
- Some layers need vector-permutation instructions

Generation of $\mathbf{A}$

- Kyber needs to generate $\mathbf{A} \leftarrow \text{Parse}(\text{XOF}(\textit{seed}))$
- For Kyber-1024, generate 16 polynomials in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^{256} + 1)$

Generation of $\mathbf{A}$

- Kyber needs to generate $\mathbf{A} \leftarrow \text{Parse}(\text{XOF}(\textit{seed}))$
- For Kyber-1024, generate 16 polynomials in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^{256} + 1)$
- XOF generates uniform-looking bytes in blocks of size 168
- Take 12 bits from XOF output, check if < 3329

Generation of $\mathbf{A}$

- Kyber needs to generate $\mathbf{A} \leftarrow \text{Parse}(\text{XOF}(\textit{seed}))$
- For Kyber-1024, generate 16 polynomials in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^{256}+1)$
- XOF generates uniform-looking bytes in blocks of size 168
- Take 12 bits from XOF output, check if < 3329
- Two options:
  1. Run XOF **once**, use for all 16 polynomials
  2. Run XOF $16\times$, once per polynomial

Generation of $\mathbf{A}$

- Kyber needs to generate $\mathbf{A} \leftarrow \text{Parse}(\text{XOF}(\textit{seed}))$
- For Kyber-1024, generate 16 polynomials in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^{256} + 1)$
- XOF generates uniform-looking bytes in blocks of size 168
- Take 12 bits from XOF output, check if < 3329
- Two options:
  1. Run XOF **once**, use for all 16 polynomials
  2. Run XOF $16\times$, once per polynomial
- Option 1 needs less computation (on average)
- Option 2 lets us use vectorization across XOF invocations

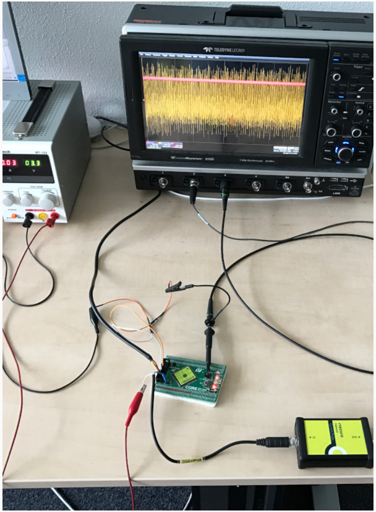
Generation of $\mathbf{A}$

- Kyber needs to generate $\mathbf{A} \leftarrow \text{Parse}(\text{XOF}(\textit{seed}))$
- For Kyber-1024, generate 16 polynomials in $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^{256}+1)$
- XOF generates uniform-looking bytes in blocks of size 168
- Take 12 bits from XOF output, check if < 3329
- Two options:
  1. Run XOF **once**, use for all 16 polynomials
  2. Run XOF $16\times$, once per polynomial
- Option 1 needs less computation (on average)
- Option 2 lets us use vectorization across XOF invocations
- Kyber uses Option 2

2. Security

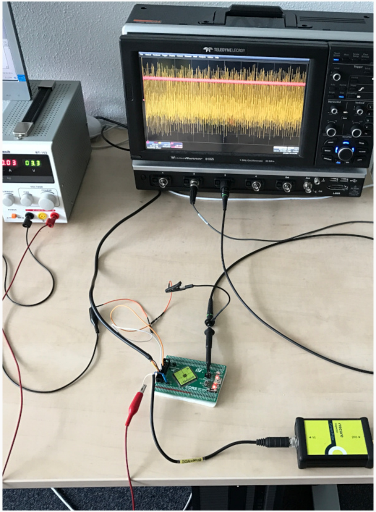
Side-channel attacks

Side-channel attacks



- Attackers see more than input/output:
  - Power consumption
  - Electromagnetic radiation
  - Timing
- Side-channel attacks:
  - Measure information
  - Use to obtain secret data

Side-channel attacks



- Attackers see more than input/output:
  - Power consumption
  - Electromagnetic radiation
  - Timing
- Side-channel attacks:
  - Measure information
  - Use to obtain secret data
- **Timing** attacks
  - Software visible
  - Can be performed **remotely**

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**

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if

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

Timing leakage part I

- Consider the following piece of code:

  if s **then**

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if

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

Timing leakage part I

- Consider the following piece of code:

  if s **then**
  $r \leftarrow A$
  else
  $r \leftarrow B$
  end if
- General structure of any conditional branch
- A and B can be large computations, r can be a large state
- This code takes different amount of time, depending on s
- Obvious timing leak if s is secret
- Even if A and B take the same amount of cycles this is *generally not* constant time!
- Reasons: Branch prediction, instruction-caches
- **Never use secret-data-dependent branch conditions**

Eliminating branches

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

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

Eliminating branches

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

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

- Replace by

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

Eliminating branches

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

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

- Replace by

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

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

Eliminating branches

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

  if s **then**

  $r \leftarrow A$

  else

  $r \leftarrow B$

  end if

- Replace by

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

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

Fun with optimizing compilers

```
void poly_frommsg(poly *r, const uint8_t msg[KYBER_INDCPA_MSGBYTES])
{
  unsigned int i,j;
  int16_t mask;

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

Fun with optimizing compilers

```
void poly_frommsg(poly *r, const uint8_t msg[KYBER_INDCPA_MSGBYTES])
{
  unsigned int i,j;
  int16_t mask;

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

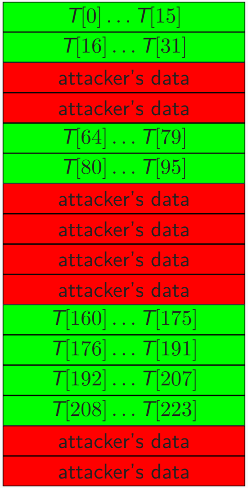
- LLVM from version 15 optimizes this to a branch with some flags
- Pointed out by Antoon Purnal, May 2024
- Different options to fix, all amount to “fighting the compiler”

Timing leakage part II

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

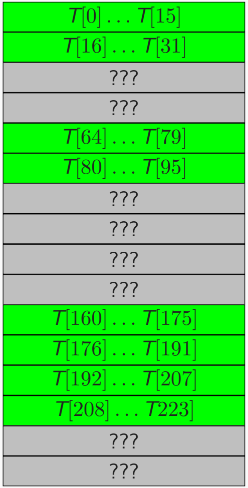
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache

Timing leakage part II



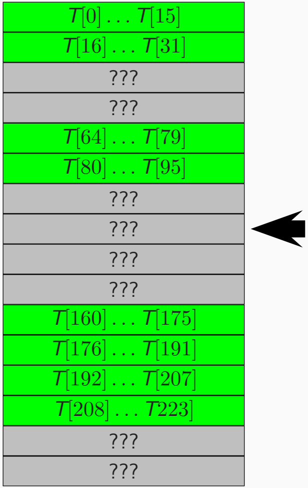
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines

Timing leakage part II



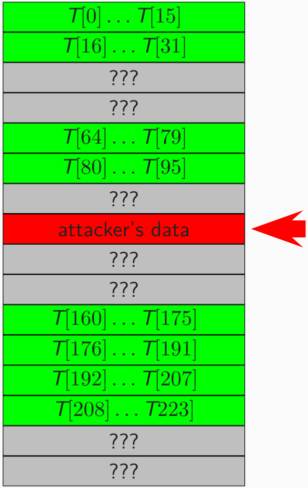
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again

Timing leakage part II



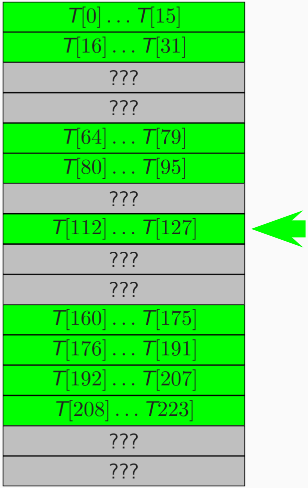
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads his data:

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, e.g.:

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

Is that all? (Timing leakage part III)

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can *always* be done; cost highly depends on the algorithm

Is that all? (Timing leakage part III)

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can *always* be done; cost highly depends on the algorithm
- Test this with `valgrind` and *uninitialized secret data* (see `https://www.post-apocalyptic-crypto.org/timecop/`)

Is that all? (Timing leakage part III)

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can *always* be done; cost highly depends on the algorithm
- Test this with `valgrind` and *uninitialized secret data* (see `https://www.post-apocalyptic-crypto.org/timecop/`)

"In order for a function to be constant time, the branches taken and memory addresses accessed must be independent of any secret inputs. (That's assuming that the fundamental processor instructions are constant time, but that's true for all sane CPUs.)"

—Langley, Apr. 2010

Is that all? (Timing leakage part III)

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can *always* be done; cost highly depends on the algorithm
- Test this with `valgrind` and *uninitialized secret data* (see https://www.post-apocalyptic-crypto.org/timecop/)

"In order for a function to be constant time, the branches taken and memory addresses accessed must be independent of any secret inputs. (That's assuming that the fundamental processor instructions are constant time, but that's true for all sane CPUs.)"

—Langley, Apr. 2010

"So the argument to the `DIV` instruction was smaller and `DIV`, on Intel, takes a variable amount of time depending on its arguments!"

—Langley, Feb. 2013

More fun with optimizing compilers

```
void poly_tomsg(uint8_t msg[KYBER_INDCPA_MSGBYTES], const poly *a)
{
  unsigned int i,j;
  uint16_t t;

  for(i=0;i<KYBER_N/8;i++) {
    msg[i] = 0;
    for(j=0;j<8;j++) {
      t  = a->coeffs[8*i+j];
      t += ((int16_t)t >> 15) & KYBER_Q;
      t  = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
      msg[i] |= t << j;
    }
  }
}
```

More fun with optimizing compilers

- Division **by a constant** typically uses multiplication
  Division by Invariant Integers using Multiplication, Granlund, Montgomery, PLDI 1994

More fun with optimizing compilers

- Division **by a constant** typically uses multiplication
  Division by Invariant Integers using Multiplication, Granlund, Montgomery, PLDI 1994
- With some flags, gcc and LLVM will actually use a `DIV` instruction
- See “KyberSlash” paper by Bernstein, Bhargavan, Bhasin, Chattopadhyay, Kiah Chia, Kannwischer, Kiefer, Paiva, Ravi, Tamvada. https://eprint.iacr.org/2024/1049

More fun with optimizing compilers

- Division **by a constant** typically uses multiplication
  Division by Invariant Integers using Multiplication, Granlund, Montgomery, PLDI 1994
- With some flags, gcc and LLVM will actually use a `DIV` instruction
- See “KyberSlash” paper by Bernstein, Bhargavan, Bhasin, Chattopadhyay, Kiah Chia, Kannwischer, Kiefer, Paiva, Ravi, Tamvada. https://eprint.iacr.org/2024/1049
- Rewrite division, but still no *guarantee* that compilers won’t use `DIV`

Imagine

Problem: There is no concept of secret data in LLVM or GCC!

Imagine

Problem: There is no concept of secret data in LLVM or GCC!

Imagine a language+compiler compiler that...

- distinguishes between public and secret data
- gives programmers all power to optimize (crypto) code

Imagine

Problem: There is no concept of secret data in LLVM or GCC!

Imagine a language+compiler compiler that...

- distinguishes between public and secret data
- gives programmers all power to optimize (crypto) code
- is formally proven to preserve semantics through compilation

Imagine

Problem: There is no concept of <u>secret</u> data in LLVM or GCC!

Imagine a language+compiler compiler that. . .

- distinguishes between public and secret data
- gives programmers all power to optimize (crypto) code
- is formally proven to preserve semantics through compilation
- is formally proven to preserve “constant-time” through compilation

Imagine

Problem: There is no concept of secret data in LLVM or GCC!

Imagine a language+compiler compiler that...

- distinguishes between public and secret data
- gives programmers all power to optimize (crypto) code
- is formally proven to preserve semantics through compilation
- is formally proven to preserve "constant-time" through compilation
- can check safety properties at compile time

Imagine

Problem: There is no concept of secret data in LLVM or GCC!

Imagine a language+compiler compiler that...

- distinguishes between public and secret data
- gives programmers all power to optimize (crypto) code
- is formally proven to preserve semantics through compilation
- is formally proven to preserve "constant-time" through compilation
- can check safety properties at compile time
- can automatically zeroize sensitive data at well-defined spots

Problem: There is no concept of <u>secret</u> data in LLVM or GCC!

Imagine a language+compiler compiler that...

- distinguishes between public and secret data
- gives programmers all power to optimize (crypto) code
- is formally proven to preserve semantics through compilation
- is formally proven to preserve "constant-time" through compilation
- can check safety properties at compile time
- can automatically zeroize sensitive data at well-defined spots
- interfaces to interactive theorem provers to verify functional correctness

Jasmin – assembly in your head

Almeida, Barbosa, Barthe, Blot, Grégoire, Laporte, Oliveira, Pacheco, Schmidt, Strub. *Jasmin: High-Assurance and High-Speed Cryptography.* ACM CCS 2017

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

Jasmin – assembly in your head

Almeida, Barbosa, Barthe, Blot, Grégoire, Laporte, Oliveira, Pacheco, Schmidt, Strub. *Jasmin: High-Assurance and High-Speed Cryptography.* ACM CCS 2017

- Language with "C-like" syntax
- Programming in Jasmin is much closer to assembly:
  - Generally: 1 line in Jasmin → 1 line in assembly
  - A few exceptions, but highly predictable
  - Compiler does not schedule code
  - Compiler does not spill registers
- Many new features since 2017 paper
- Big credit also to Santiago Arranz Olmos and Jean-Christophe Léchenet!
- See Ph.D. thesis by Oliveira:
  High-speed and High-assurance Cryptographic Software

Constant-time code in Jasmin

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

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

Constant-time code in Jasmin

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

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

Constant-time code in Jasmin

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

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

Constant-time code in Jasmin

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

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

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

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

Constant-time code in Jasmin

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

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

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

Spectre v1 (“Speculative bounds-check bypass”)

```
stack u8[10] public;
stack u8[32] secret;
reg u8 t;
reg u64 r, i;

i = 0;
while(i < 10) {
  t = public[(int) i] ;
  r = leak(t);
  ...
}
```

It's more subtle than this

```
fn aes_rounds (stack u128[11] rkeys, reg u128 in) -> reg u128 {
  reg u64 rkoffset;
  state = in;

  state ^= rkeys[0];
  rkoffset = 0;
  while(rkoffset < 9*16) {
    rk = rkeys.[(int)rkoffset];
    state = #AESENC(state, rk);
    rkoffset += 16;
  }
  rk = rkeys[10];
  #declassify state = #AESENCLAST(state, rk);
  return state;
}
```

It's more subtle than this

Spectre declassified

- Caller is free to leak (declassified) state
- Very common in crypto: ciphertext is actually **sent**!
- `state` is not "out of bounds" data, it's "early data"
- Must not speculatively `#declassify` early!

Ammanaghatta Shivakumar, Barnes, Barthe, Cauligi, Chuengsatiansup, Genkin, O'Connell, Schwabe, Sim, and Yarom: *Spectre Declassified: Reading from the Right Place at the Wrong Time*. IEEE S&P 2023. https://eprint.iacr.org/2022/426

Countermeasures

Fencing

- Can prevent speculation through **barriers** (LFENCE)
- Protecting *all* branches is possible but costly

Countermeasures

Fencing

- Can prevent speculation through **barriers** (LFENCE)
- Protecting *all* branches is possible but costly

Speculative Load Hardening

- Idea: maintain misprediction predicate `ms` (in a register)
- At every branch use arithmetic to update predicate
- Option 1: Mask every loaded value with `ms`
- Option 2: Mask every address with `ms`
- Effect: during misspeculation "leak" constant value

Countermeasures

Fencing

- Can prevent speculation through **barriers** (LFENCE)
- Protecting *all* branches is possible but costly

Speculative Load Hardening

- Idea: maintain misprediction predicate `ms` (in a register)
- At every branch use arithmetic to update predicate
- Option 1: Mask every loaded value with `ms`
- Option 2: Mask every address with `ms`
- Effect: during misspeculation "leak" constant value
- Implemented in LLVM since version 8
  - Still noticable performance overhead
  - No formal guarantees of security

Selective SLH

Do we need to mask/protect all loads?

Selective SLH

Do we need to mask/protect all loads?

- No need to mask loads into registers that never enter leaking instructions

Selective SLH

Do we need to mask/protect all loads?

- No need to mask loads into registers that never enter leaking instructions
- `secret` registers never enter leaking instructions!
- Obvious idea: mask only loads into `public` registers

Extending the type system

- Type system gets three security levels:
  - `secret`: secret
  - `public`: public, also during misspeculation
  - `transient`: public, but possibly secret during misspeculation

Extending the type system

- Type system gets three security levels:
  - `secret`: secret
  - `public`: public, also during misspeculation
  - `transient`: public, but possibly secret during misspeculation
- Maintain misspeculation flag `ms`:
  - `ms = #init_msf()`: Translate to `LFENCE`, set `ms` to zero
  - `ms = #set_msf(b, ms)`: Set `ms` according to branch condition `b`
  - Branches invalidate `ms`

Extending the type system

- Type system gets three security levels:
  - `secret`: secret
  - `public`: public, also during misspeculation
  - `transient`: public, but possibly secret during misspeculation
- Maintain misspeculation flag `ms`:
  - `ms = #init_msf()`: Translate to `LFENCE`, set `ms` to zero
  - `ms = #set_msf(b, ms)`: Set `ms` according to branch condition `b`
  - Branches invalidate `ms`
- Two operations to lower level:
  - `x = #protect(x, ms)`: Go from `transient` to `public`
  - `#protect` translates to mask by `ms`

Extending the type system

- Type system gets three security levels:
  - `secret`: secret
  - `public`: public, also during misspeculation
  - `transient`: public, but possibly secret during misspeculation
- Maintain misspeculation flag `ms`:
  - `ms = #init_msf()`: Translate to `LFENCE`, set `ms` to zero
  - `ms = #set_msf(b, ms)`: Set `ms` according to branch condition `b`
  - Branches invalidate `ms`
- Two operations to lower level:
  - `x = #protect(x, ms)`: Go from `transient` to `public`
  - `#protect` translates to mask by `ms`
  - `#declassify r`: Go from `secret` to `transient`
  - `#declassify` requires cryptographic proof/argument

Extending the type system

- Type system gets three security levels:
  - `secret`: secret
  - `public`: public, also during misspeculation
  - `transient`: public, but possibly secret during misspeculation
- Maintain misspeculation flag `ms`:
  - `ms = #init_msf()`: Translate to `LFENCE`, set `ms` to zero
  - `ms = #set_msf(b, ms)`: Set `ms` according to branch condition `b`
  - Branches invalidate `ms`
- Two operations to lower level:
  - `x = #protect(x, ms)`: Go from `transient` to `public`
  - `#protect` translates to mask by `ms`
  - `#declassify r`: Go from `secret` to `transient`
  - `#declassify` requires cryptographic proof/argument
- Still: allow branches and indexing only for `public`

The special case of crypto

- We know what inputs **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need `protect`!

The special case of crypto

- We know what inputs **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need `protect`!
- Even better: mark additional inputs as **secret**
- No cost if those inputs don't flow into leaking instructions

The special case of crypto

- We know what inputs **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need `protect`!
- Even better: mark additional inputs as **secret**
- No cost if those inputs don't flow into leaking instructions
- Even better: Spills don't need `protect` if there is no branch between store and load

The special case of crypto

- We know what inputs **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need `protect`!
- Even better: mark additional inputs as **secret**
- No cost if those inputs don't flow into leaking instructions
- Even better: Spills don't need `protect` if there is no branch between store and load
- Even better: "Spill" public data to MMX registers instead of stack

The special case of crypto

- We know what inputs **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need `protect`!
- Even better: mark additional inputs as **secret**
- No cost if those inputs don't flow into leaking instructions
- Even better: Spills don't need `protect` if there is no branch between store and load
- Even better: "Spill" public data to MMX registers instead of stack

Type system supports programmer in writing efficient Spectre-v1-protected code!

Performance impact (Comet Lake cyles)

Primitive	Impl.	Op.	CT	SCT	overhead [%]
ChaCha20	avx2	32 B	314	352	12.10
	avx2	32 B xor	314	352	12.10
	avx2	128 B	330	370	12.12
	avx2	128 B xor	338	374	10.65
	avx2	1 KiB	1190	1234	3.70
	avx2	1 KiB xor	1198	1242	3.67
	avx2	1 KiB	18872	18912	0.21
	avx2	16 KiB xor	18970	18994	0.13

Performance impact (Comet Lake cyles)

Primitive	Impl.	Op.	CT	SCT	overhead [%]
X25519	mulx	smult	98352	98256	-0.098
	mulx	base	98354	98262	-0.094
Kyber512	avx2	keypair	25694	25912	0.848
	avx2	enc	35186	35464	0.790
	avx2	dec	27684	27976	1.055
Kyber768	avx2	keypair	42768	42888	0.281
	avx2	enc	54518	54818	0.550
	avx2	dec	43824	44152	0.748

References

Ammanaghatta Shivakumar, Barthe, Grégoire, Laporte, Oliveira, Priya, Schwabe, and Tabary-Maujean: *Typing High-Speed Cryptography against Spectre v1*. IEEE S&P 2023. https://eprint.iacr.org/2022/1270

Arranz Olmos, Barthe, Blatter, Grégoire, and Laporte: *Preservation of Speculative Constant-time by Compilation*. https://eprint.iacr.org/2024/1203

FORMOSA CRYPTO

- Goal: Formally verified post-quantum crypto
- Software written in Jasmin
- Implementation security through Jasmin language features
- Proofs of functional correctness using EasyCrypt
- Security proofs in EasyCrypt

University of Bristol
The University of Melbourne
CASA — Cyber Security in the Age of Large-Scale Adversaries
Max Planck Institute for Security and Privacy
institute IMDEA software
U.PORTO
Radboud University
INESCTEC
Ruhr Universität Bochum RUB
Inria — Inventeurs du monde numérique
TU/e Eindhoven University of Technology

Formosan black bear

文A 24 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

The **Formosan black bear** (臺灣黑熊, ***Ursus thibetanus formosanus***), also known as the **Taiwanese black bear** or **white-throated bear**, is a subspecies of the Asiatic black bear. It was first described by Robert Swinhoe in 1864. Formosan black bears are endemic to Taiwan. They are also the largest land animals and the only native bears (*Ursidae*) in Taiwan. They are seen to represent the Taiwanese nation.

Because of severe exploitation and habitat degradation in recent decades, populations of wild Formosan black bears have been declining. This species was listed as "endangered" under Taiwan's Wildlife Conservation Act (Traditional Chinese: 野生動物保育法) in 1989. Their geographic distribution is restricted to remote, rugged areas at elevations of 1,000–3,500 metres (3,300–11,500 ft). The estimated number of individuals is 200 to 600.[3]

Formosan black bear

Conservation status

Extinct Threatened Least Concern

EX EW CR EN VU NT LC

Vulnerable (IUCN 3.1)[1]

Physical characteristics [edit]

The V-shaped white mark on a bear's chest

The Formosan black bear is sturdily built and has a round head, short neck, small eyes, and long snout. Its head measures 26–35 cm (10–14 in) in length and 40–60 cm (16–24 in) in circumference. Its ears are 8–12 cm (3.1–4.7 in) long. Its snout resembles a dog's, hence its nickname is "dog bear". Its tail is inconspicuous and short—usually less than 10 cm (3.9 in) long. Its body is well covered with rough, glossy, black hair, which can grow over 10 cm long around the neck. The tip of its chin is white. On the chest, there is a

https://en.wikipedia.org/wiki/Formosan_black_bear

Learn more

Kyber website:	https://pq-crystals.org/kyber/
NIST PQC:	https://csrc.nist.gov/projects/post-quantum-cryptography
pqc-forum:	https://groups.google.com/a/list.nist.gov/g/pqc-forum
Formosa Crypto:	https://formosa-crypto.org