Something with implementations

Peter Schwabe

June 23, 2016

PQCRYPTO Summer School on Post-Quantum Cryptography 2017

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**

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if

Eliminating branches

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

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

  $\quad r \leftarrow A$

  else

  $\quad r \leftarrow B$

  end if
- Replace by
$$r \leftarrow sA + (1 - s)B$$
- Can expand s to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication
- For very fast A and B this can even be faster

Problem No. 2

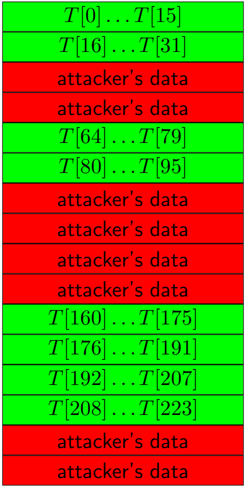
```
table[secret]
```

Timing leakage part II

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

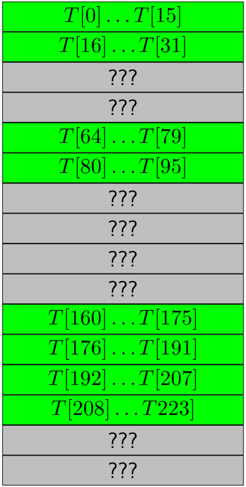
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache

Timing leakage part II



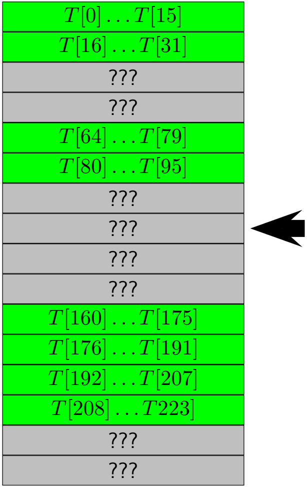
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines

Timing leakage part II



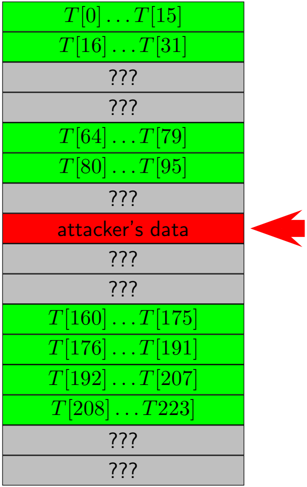
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again

Timing leakage part II



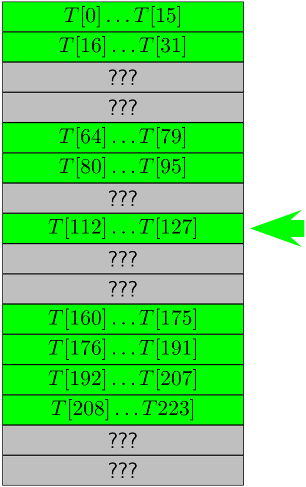
- Consider lookup table of 32-bit integers
- *Cache lines* have 64 bytes
- Crypto and the attacker's program run on the same CPU
- Tables are in cache
- The attacker's program replaces some cache lines
- Crypto continues, loads from table again
- Attacker loads his data:

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)

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

“The multicore revolution”

- Until early years 2000 each new processor generation had higher clock speeds
- Nowadays: increase performance by number of cores:
  - My laptop has 2 phyiscal (and 4 virtual) cores
  - Smartphones typically have 2 or 4 cores
  - Servers have 4, 8, 16,... cores
  - Special-purpose hardware (e.g., GPUs) often comes with many more cores
- Consequence: “The free lunch is over” (Herb Sutter, 2005)

“The multicore revolution”

- Until early years 2000 each new processor generation had higher clock speeds
- Nowadays: increase performance by number of cores:
  - My laptop has 2 phyiscal (and 4 virtual) cores
  - Smartphones typically have 2 or 4 cores
  - Servers have 4, 8, 16,. . . cores
  - Special-purpose hardware (e.g., GPUs) often comes with many more cores
- Consequence: “The free lunch is over” (Herb Sutter, 2005)

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

—Maurice Herlihy: The Multicore Revolution, 2007

Why multicore doesn't matter. . .

. . . for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

Why multicore doesn't matter...

... for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

Post-quantum crypto is fast

- > 3900 "lattisigns512" signatures per second
- > 45000 "lattisigns512" verifications per second
- > 38000 rainbow5640 signatures per second
- > 57000 rainbow5640 verifications per second

Why multicore doesn't matter. . .

. . . for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

Post-quantum crypto is fast

- > 3900 "lattisigns512" signatures per second
- > 45000 "lattisigns512" verifications per second
- > 38000 rainbow5640 signatures per second
- > 57000 rainbow5640 verifications per second

- **If you perform only one crypto operation, you don't care**

Why multicore doesn't matter...
... for algorithm design in crypto

Crypto is fast (single core of Intel Core i3-2310M)

- > 50 RSA-4096 signatures per second
- > 8000 RSA-4096 signature verifications per second
- > 28000 Ed25519 signatures per second
- > 9000 Ed25519 signature verifications per second

Post-quantum crypto is fast

- > 3900 "lattisigns512" signatures per second
- > 45000 "lattisigns512" verifications per second
- > 38000 rainbow5640 signatures per second
- > 57000 rainbow5640 verifications per second

- **If you perform only one crypto operation, you don't care**
- **Many crypto operations are trivially parallel on multiple cores**

Pipelined and multiscalar processors

- Almost all CPUs 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

Pipelined and multiscalar processors

- Almost all CPUs 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
- Pipelined execution: overlap processing of *independent* instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)

Pipelined and multiscalar processors

- Almost all CPUs 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
- Pipelined execution: overlap processing of *independent* instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)
- Superscalar execution: duplicate units and process multiple instructions in the same stage

Pipelined and multiscalar processors

- Almost all CPUs 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
- Pipelined execution: overlap processing of *independent* instructions (e.g., while one instruction is in step 2, the next one can do step 1 etc.)
- Superscalar execution: duplicate units and process multiple instructions in the same stage
- Crucial to make use of these concepts: *instruction-level parallelism*
- To some extent, compilers will help here

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 would you care?

- Consider the Intel Nehalem processor

Why would you care?

- 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

Why would you care?

- 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

Why would you care?

- 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**

Why would you care?

- 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

Why would you care? (Part II)

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

Why would you care? (Part II)

- 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

Why would you care? (Part II)

- 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!

Example: butterflies

- Recall the NTT in NewHope
- Polynomials are represented as `uint32_t aa[1024]`
- Inside NTT load into vectors of 4 double-precision floats
- Perform 4 parallel butterflies on `vx0` and `vx1`:

```
vx0 = _mm256_cvtepi32_pd (*(__m128i*) aa);
vx1 = _mm256_cvtepi32_pd (*(__m128i*) (aa+offset));

vt  = _mm256_add_pd(vx0, vx1);
vx1 = _mm256_sub_pd(vx1, vx0);
vx1 = _mm256_mul_pd(vx1, vomega);

// reduce
vc = _mm256_mul_pd(vx1, vqinv);
vc = _mm256_round_pd(vc,0x09);
vc = _mm256_mul_pd(vc, vq);
vx1 = _mm256_sub_pd(vx1, vc);

sv = _mm256_cvtpd_epi32(vx0);
_mm_store_si128((__m128i *)aa,sv);

sv = _mm256_cvtpd_epi32(vt)
_mm_store_si128((__m128i *)(aa+4),sv);
```

Take-home message

- Never branch on secret data
- Never access memory at secret addresses
- Vectorize, vectorize, vectorize!

Exercise

- Download https://cryptojedi.org/mvmul.tar.bz2
- Unpack and cd: `tar xjvf mvmul.tar.bz2 && cd mvmul`
- Implement fast version of matrix-vector multiplication (`mvmul_fast`)

Exercise

- Download https://cryptojedi.org/mvmul.tar.bz2
- Unpack and cd: `tar xjvf mvmul.tar.bz2 && cd mvmul`
- Implement fast version of matrix-vector multiplication (`mvmul_fast`)
- Program will test against (slow) reference implementation
- Program will then benchmark both functions.

Exercise

- Download https://cryptojedi.org/mvmul.tar.bz2
- Unpack and cd: `tar xjvf mvmul.tar.bz2 && cd mvmul`
- Implement fast version of matrix-vector multiplication (`mvmul_fast`)
- Program will test against (slow) reference implementation
- Program will then benchmark both functions.
- Possibly helpful:
  - https://software.intel.com/sites/landingpage/IntrinsicsGuide/
  - http://agner.org/optimize/instruction_tables.pdf