SDU
University of Southern Denmark

Cryptographic Engineering

Timing Side Channels

Ruben Niederhagen
(based on content by Norman Lahr, Richard Petri, and Peter Schwabe)

Department of Mathematics and Computer Science (IMADA)

Department of Mathematics
and Computer Science (IMADA)

- Timing Attacks exploit the timing variations of implementations.
- Information leakage: The runtime of a cryptographic operation depends on processed data (e.g., a secret key).
- The reasons of the data dependency are conditional cases in the program, the caching of data, or even instruction latencies.
- The timing characteristics can potentially be exploited to reveal secrets.

Targets

- Mostly software implementations running:
  - on microcontrollers (e.g., smart cards) and
  - on general purpose processors (e.g., personal computers or server systems).
- Algorithm targets are:
  - public key ciphers, e.g., RSA or ECC-based ciphers,
  - and symmetric ciphers, e.g., AES,
  - other algorithms processing sensitive data (e.g., passwords).
- In some scenarios attacks can be mounted remotely!

Requirements

- For an attack we need reasonably precise timing measurements.
- Since there are a lot of noise sources in complex computer systems, a huge amount of measurements is required to get a proper signal-to-noise ratio.
- Examples of noise sources are:
  - interrupts,
  - parallel computations,
  - network delay,
  - caching. . .
- Timing analysis requires knowledge or accurate assumptions on the implementation.

Brute Force Attack

- Assume that the comparison function is utilized for a password prompt.
- The prompt is never locked.
- Idea: Brute force all combinations.
- Alphabet $\Sigma = \{a - z, A - Z, 0 - 9, !..\#\}$
- $\Sigma^n = (2 \cdot 26 + 10 + 30)^n \in \mathcal{O}(92^n)$
- Example:
  - $n = 12 \Rightarrow 92^{12} \approx 3.67 \cdot 10^{23} \approx 2^{78}$ possible passwords.
  - $1\mu s$ per attempt $\Rightarrow$ ca. 9.6 billion years (max.)!

Naiive Compare Function

Byte Comparison

```
function compare(s1, s2, len)
begin
  for i = 0 to len-1
    if s1[i] != s2[i]
      return false;
  return true;
end
```

Timing attack on the naiive approach:

- Let us exploit the timing!
- Reveal the password with the following approach:
  - Test all values of Σ sequentially.
  - Acquire the time per run.
  - If the character is correct, the runtime increases.
  - Repeat until the complete password is revealed.

GNU libc and OpenBSD libc Byte Comparison

memcmp.c

```
static int
memcmp_bytes (op_t a, op_t b)
{
  long int srcp1 = (long int) &a;
  long int srcp2 = (long int) &b;
  op_t a0, b0;

  do
    {
      a0 = ((byte *) srcp1)[0];
      b0 = ((byte *) srcp2)[0];
      srcp1 += 1;
      srcp2 += 1;
    }
  while (a0 == b0);
  return a0 - b0;
}
```

timingsafe_bcmp.c

```
int
timingsafe_bcmp(const void *b1, const void *b2, size_t n)
{
        const unsigned char *p1 = b1, *p2 = b2;
        int ret = 0;

        for (; n > 0; n--)
                ret |= *p1++ ^ *p2++;
        return (ret != 0);
}
```

RSA (Reminder)

Encrypt

$$c = m^e \mod n$$

Decrypt

$$e = c^d \mod n$$

Sign

$$s = h(m)^d \mod n$$

Verify

$$v = s^e \mod n$$

Square & Multiply (Reminder)

- Many schemes involve some form of exponentiation in a group.
- An exponentiation can be expressed using only squaring and multiplying:

$$a \cdot \underbrace{a \cdot a \cdot a} \cdot \underbrace{a \cdot a \cdot a} = a \cdot (a \cdot \underbrace{a \cdot a})^2 = a \cdot (a \cdot (a)^2)^2$$

- The binary representation of the exponent is used to determine the pattern:

$$a^{75} = a^{1001011_{\text{bin}}} = ((((((1 \cdot a)^2)^2)^2 a)^2)^2 a)^2 a$$

```c
void exp(bigint* m,
         bigint* c,
         bigint* d)
{
  bigint_set(m, 1);
  int v = bigint_size(d);
  for (int i = v-1;
       i >= 0; i--) {
    bigint_square(m);
    if (bigint_bit(d, i) == 1) {
      bigint_mul(m, c);
    }
  }
}
```

- Original attack by Kocher from 1996.
- Target is a software implementation of RSA.
- The exponentiation algorithm is the Square&Mult. (left-to-right) approach.
- An attacker wants to reveal the secret key d.
- Leakage: The timing difference of the data dependent conditional case.

Example Timings

FIGURE 1: RSAREF Modular Multiplication Times

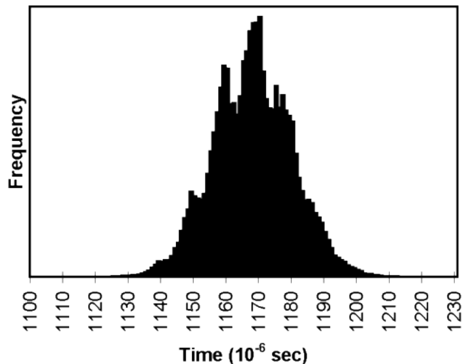
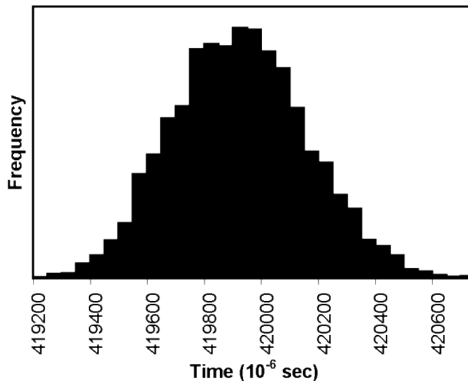


FIGURE 2: RSAREF Modular Exponentiation Times



Department of Mathematics
and Computer Science (IMADA)

The attacker is able to:

- communicate with the target device (remote or local),
- send infinitely many ciphertexts $c_1, c_2, \ldots, c_N$ to the device, and
- measure the time $\mathrm{time}(c^d \mod n)$ of the decryption, and
- determine $\mathrm{time}(a \cdot b \mod n)$ for $a, b \in \{0, 1, ..., n-1\}$ (e.g., by simulation).

Attack Idea (rough)

1. Measurement: Measure the decryption timing of many different ciphertexts.
2. Attack: Guess private key, bit by bit, starting at the top (first bit is always 1).
   2.1 For each ciphertext, determine how long the timing of all known bits takes and subtract it from the measured timing.
   2.2 For each ciphertext, determine how long the rest of the calculation should take, if the next bit would be 1 (square *and* multiply).
   2.3 For each ciphertext, determine how long the rest of the calculation should take, if the next bit would be 0 (*only* square).
   2.4 Compare the two timing series and decide which scenario fits better.

Attack Idea (1/2)

The attacker sequentially guesses the bits of the exponent d:

- $d = \{d_{v-1}, d_{v-2}, ..., d_0\}$, with d_{v-1} as the most significant bit (which is always 1).

Assumptions:

- The attacker measured the timing of the decryption of each ciphertext, i.e. $t_1 = \mathrm{time}(c_1^d \mod n), ..., t_N = \mathrm{time}(c_N^d \mod n)$.
- $d_{v-1}, d_{v-2}, ..., d_{k+1}$ are already correctly guessed.

Department of Mathematics
and Computer Science (IMADA)

Guess next bit d_k:

- Determine the intermediate result m'_j from already guessed $d_{v-1}, ..., d_{k+1}$ for all $j \leq N$.
- Estimate the remaining time for the unknown bits:

$$\begin{aligned} t_{\text{rem},j} =& t_j - \Sigma_{i=k+1}^{v-1}(1-d_i)u_i + d_i(v_i + x_i), \text{ with} \\ & u_i \text{ :time for squaring, if } d_i = 0, \\ & v_i \text{ :time for multiplication, if } d_i = 1, \text{ and} \\ & x_i \text{ :time for squaring, if } d_i = 1. \end{aligned}$$

- Apply the decision strategy d_k:

$$d_k = \begin{cases} 0, & \text{if} \quad \text{Var}(t_{rem,1} - u_1, ..., t_{rem,N} - u_N) < \\ & \qquad \text{Var}(t_{rem,1} - (v_1 + x_1), ..., t_{rem,N} - (v_N + x_N)) \\ 1, & \qquad \text{otherwise} \end{cases}$$

SDU

Square & Multiply


```
void exp(bigint* m,
         bigint* c,
         bigint* d)
{

  bigint_set(m, 1);
  int v = bigint_size(d);
  for (int i = v-1; i >= 0; i–) {
    bigint_square(m);
    if (bigint_bit(d, i) == 1) {
      bigint_mul(m, c);
    }
  }
}
```

Insert dummy operation?


```c
void exp(bigint* m,
         bigint* c,
         bigint* d)
{
  bigint dummy;
  bigint_set(m, 1);
  int v = bigint_size(d);
  for (int i = v-1; i >= 0; i--) {
    bigint_square(m);
    if (bigint_bit(d, i) == 1) {
      bigint_mul(m, c);
    } else {
      bigint_mul(&dummy, c);
    }
  }
}
```

Still not guaranteed to be constant time:

- 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.
- Even if A and B seem to take the same amount of cycles this is generally not true!
- Reasons: Branch prediction, instruction-caches, ...
- **Never use secret-data-dependent branch conditions!**

Department of Mathematics
and Computer Science (IMADA)

- 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 than branching.

Eliminating Branches


```
void exp(bigint* m,
         bigint* c,
         bigint* d)
{
  bigint mmul, tmp;
  bigint_set(m, 1);
  int v = bigint_size(d);
  for (int i = v-1; i >= 0; i--) {
    bigint_square(m);
    bigint_set(mmul, m);                       bigint_mul(mmul, c);
    bigint_set(tmp,   bigint_bit(d, i));       bigint_mul(mmul, tmp);
    bigint_set(tmp, 1-bigint_bit(d, i));       bigint_mul(m,    tmp);
    bigint_add(m, mmul);
  }
}
```

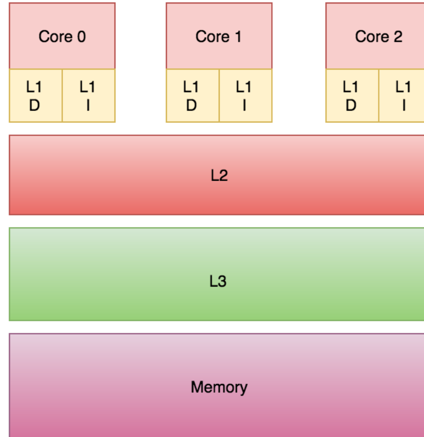
Exponent blinding:

- Use $d' = d + r\varphi(n)$, with r is a random number.
- $c^{d+r\varphi(n)} \equiv c^d \mod n$

Basis blinding:

$$
\begin{aligned}
c &= m^e \mod n \\
c' &= c \cdot r^e \mod n, \ 0 < r < N \\
c' &= (m \cdot r)^e \mod n \\
\\
m' &= (c')^d \mod n \\
m' &= (m \cdot r)^{ed} \mod n \\
m' &= m \cdot r \mod n \\
m &= m' \cdot r^{-1} \mod n
\end{aligned}
$$

Cache Mechanism



Cache Timing Leakage

Department of Mathematics
and Computer Science (IMADA)

- Cache hits and misses cause a timing variance during execution.
- Occurs, e.g., when accessing an array or a lookup table.
- Latencies on current processors:
  - L1 cache: 4 cycles
  - L2 cache: 12 cycles
  - L3 cache: 26-31 cycle
  - DRAM memory > 120 cycles

Cache Timing Attacks

$T[0] \ldots T[15]$
$T[16] \ldots T[31]$
???
???
$T[64] \ldots T[79]$
$T[80] \ldots T[95]$
???
$T[112] \ldots T[127]$
???
???
$T[160] \ldots T[175]$
$T[176] \ldots T[191]$
$T[192] \ldots T[207]$
$T[208] \ldots T[223]$
???
???

- Consider lookup table of 32-bit integers.
- Cache lines have 64 bytes.
- Crypto and the attacker's program run on the same CPU and thus share the cache.
- 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).

AES T-Box-based Implementation

- Round transformation: combine round function (except AddRoundKey) to a single set of table lookups.
- T-boxes: 4 tables, each 256 entries, 32 bit per entry.
- Recommended in the initial standard for fast software-based implementations.

Example Table (mbedTLS)


```
/*
 * Forward tables
 */
#define FT \
\
    V(A5,63,63,C6), V(84,7C,7C,F8), V(99,77,77,EE), V(8D,7B,7B,F6), \
    V(0D,F2,F2,FF), V(BD,6B,6B,D6), V(B1,6F,6F,DE), V(54,C5,C5,91), \
    V(50,30,30,60), V(03,01,01,02), V(A9,67,67,CE), V(7D,2B,2B,56), \
    V(19,FE,FE,E7), V(62,D7,D7,B5), V(E6,AB,AB,4D), V(9A,76,76,EC), \
    V(45,CA,CA,8F), V(9D,82,82,1F), V(40,C9,C9,89), V(87,7D,7D,FA), \
    V(15,FA,FA,EF), V(EB,59,59,B2), V(C9,47,47,8E), V(0B,F0,F0,FB), \
    V(EC,AD,AD,41), V(67,D4,D4,B3), V(FD,A2,A2,5F), V(EA,AF,AF,45), \
    V(BF,9C,9C,23), V(F7,A4,A4,53), V(96,72,72,E4), V(5B,C0,C0,9B), \
    V(C2,B7,B7,75), V(1C,FD,FD,E1), V(AE,93,93,3D), V(6A,26,26,4C), \
    ...
```

Example Round (mbedTLS)


```
#define AES_FROUND(X0,X1,X2,X3,Y0,Y1,Y2,Y3)                         \
    do                                                              \
    {                                                               \
        (X0) = *RK++ ^ AES_FT0( ( (Y0)       ) & 0xFF ) ^          \
                       AES_FT1( ( (Y1) »  8 ) & 0xFF ) ^           \
                       AES_FT2( ( (Y2) » 16 ) & 0xFF ) ^           \
                       AES_FT3( ( (Y3) » 24 ) & 0xFF );            \
                                                                    \
        (X1) = *RK++ ^ AES_FT0( ( (Y1)       ) & 0xFF ) ^          \
                       AES_FT1( ( (Y2) »  8 ) & 0xFF ) ^           \
                       AES_FT2( ( (Y3) » 16 ) & 0xFF ) ^           \
                       AES_FT3( ( (Y0) » 24 ) & 0xFF );            \
    [...]
```

Bernstein's Timing Attacks on AES using T-Tables

- Assumption: The access time to a table is similar for the same index.
- The access pattern for first accesses:
  $T_0[p \oplus k]$, with p is a plaintext byte and k is a key byte.
- The attacker can vary p.
- Template attack:
  - Training phase: Collect times t_{p_i} for $T_0[p_i \oplus 0] = T_0[p_i]$ for all p_i (constant zero key).
  - Attack phase: Gather times t'_{p_i} for $T_0[p_i \oplus k]$ for all p_i (unknown key).
  - Find the maximum t_{p_m} of the t_{p_i} and $t_{p_{m'}}$ of the t'_{p_i}.
  - Compute the key value as $k = p_m \oplus p_{m'}$.
- Details: `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`

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

- DIV, IDIV, FDIV on pretty much all Intel/AMD CPUs.
- Various math instructions on Intel/AMD CPUs (FSIN, FCOS, . . .).
- MUL, MULHW, MULHWU on many PowerPC CPUs.
- UMULL, SMULL, UMLAL, and SMLAL on ARM Cortex-M3.

Solution

- Avoid these instructions.
- Make sure that inputs to the instructions do not leak timing information.
- Requires assembler implementation or special compilers!

Department of Mathematics
and Computer Science (IMADA)

Timing-leakage sources:

- conditional branches,
- memory access,
  and
- instrucion latencies

depending on secret data.

Countermeasures:

- arithmically eliminate branches,
- avoid table lookups,
  iterate over *all* data, and
- avoid variable-latency instructions

depending on secret data.