This work is distributed under the CC BY-SA license, unless stated otherwise.

Hardware Security

6. Timing and cache attacks

Ján Jančár

Summer 2026

Today

- Timing attacks
- Cache attacks
- Defences
  - Constant-time programming
  - Compilers, Tools

Attacks
Defences

Big O

$$\Omega(n \log n)$$

$$O(n)$$

$$\Theta(n^2)$$

- Good for algorithm design
- Not good for attacks
  - lower bound, upper bound, tight bound
  - asymptotic, not concrete
  - abstract, not specific

Execution duration

- The **duration** of some operation
- What units? How do we measure it?
- Depends on what we care about
  - Computational steps,
  - Clock cycles,
  - Seconds

Example

Example

```
def bubble(l: list):
    n = len(l)
    while True:
        swapped = False

        for i in range(1, n):
            if l[i - 1] > l[i]:
                l[i], l[i - 1] = l[i - 1], l[i]
                swapped = True

        if not swapped:
            return
```

- Worst case: n^2
- Average case: n^2
- Best case: n

- In timing attacks we (usually) care about exactly one line in this code, which?
  - `if l[i - 1] > l[i]`
  - Data-dependent runtime
  - Length-dependent runtime usually not considered/does not lead to problems

*More specifically we care about **secret-dependent** runtime.*

Timing Attacks

When the attacker has a stopwatch

Timing Attacks

Side-channel attack where the execution duration of an operation is the side-channel.

- Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems (Paul Kocher - 1996)
- **Remote** / **Local**
- **Synchronous** / **Asynchronous**
- Use multiple measurements
- Measurements have
  - Some **signal**: Information that correlates with what the attacker wants
  - Some **noise**: Information that masks what the attacker wants
- **Cache attacks** are timing attacks

Attacker observes (a part of) an operation and infers information on the (secret) data it processes from its duration, because it was data-dependent.

Timing Attacks

Remote / Local

- **Remote**: There is separation between attacker and target
  - Over the network
  - Or, some other channel
  - No shared resources between attacker and target
  - Usually: More noise
  - Attacker sends a HTTP request, gets a response
  - Questionable: JavaScript in a browser tab
    - Remote or local?
    - Cache attacks possible
- **Local**: There is only limited separation
  - Some shared resources
  - Attacker observes cache accesses of a separate process
  - Attacker sends a command to a TPM, gets a response

...culminating in the first browser-based side-channel attack which is constructed entirely from Cascading Style Sheets (CSS) and HTML, and works even when script execution is completely blocked.

Timing Attacks

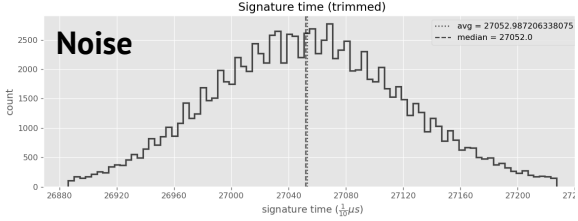
Synchronous / Asynchronous

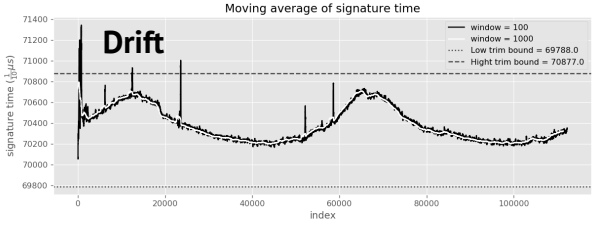
- **Synchronous**: Attacker triggers the operation
  - Thus, attacker can directly measure it (start ↔ end)
  - Attacker sends a command to a smartcard, gets a response
- **Asynchronous**: Attacker observes the operation
  - Thus, attacker needs to find its start and end
  - Attacker observes cache accesses of a TLS server, without interacting with it

Timing Attacks

Noise

- Timing measurements are **noisy**
- There can be measurement **noise**
  - Measurement error
  - Other leaks the attacker is not interested in
  - Data-dependency on other data can mask secret leakage





- There can be measurement **drift**
  - An overall trend in the measurements
  - ? What do you think caused the drift in the plot on the right?
    - It was temperature, data collection went over two days, sunlight heated the card and the host machine

Timing Attacks

Example 1

Example (From Lecture 1)

```
def checkPassword(password):
    correct = "hunter2"
    if len(correct) != len(password):
        return False
    for i in range(len(correct)):
        if password[i] != correct[i]:
            return False
    return True
```

- Slightly modified example from lecture 1
- **Attack**
  - Find length: a, aa, aaa, ...
  - Find password: aaaaaaa, baaaaaa, caaaaaa, ...

Timing Attacks

Example 2

Example (From Embedded Cryptography 1 - Chapter 1 - Example 1.3)

```
def montgomery_mul(a, b, N):
    r = a * b
    r = (r + ((r * w) mod rho) * N) // rho  # Pseudocode
    # r < 2N
    if r >= N:
        r = r - N
    return r

def RSA_decrypt(N, d, c):
    c = montgomery_init(c, N)
    r = montgomery_init(1, N)
    for i in range(d.bit_length(), 0, -1):
        r = montgomery_mul(r, r, N)
        if d & (1 << i):
            r = montgomery_mul(r, c, N)
    return montgomery_deinit(r)
```

- Left-to-right (msb-to-lsb) square and multiply
- Using Montgomery multiplication
  - Fast, avoids modular reduction by N
- What is the obvious leak?

Timing Attacks

Example 2

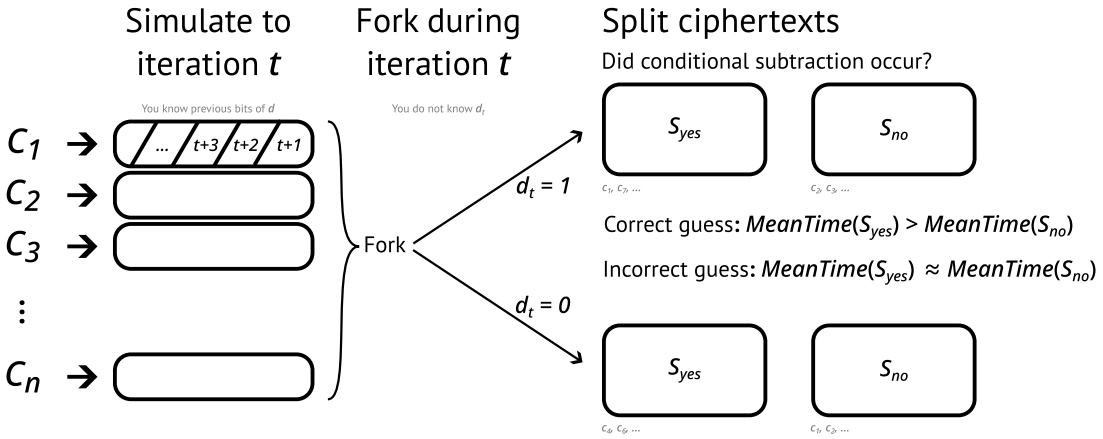
- What about the **conditional** subtraction?
  - Depends on the key but also **data**

```
if r >= N:
    r = r - N
```

- **Attack**
  - Plan: Iteratively recover the next bit d_t if you know previous bits $d_{top}, ..., d_{t+1}$
    - Most significant bit is always 1
  - Collect measurements of random ciphertexts c
  - For each, simulate (partial) decryption until iteration t
  - Then, fork simulation into two cases
    - $d_t = 0$
    - $d_t = 1$
  - For each case, simulate iteration t and mark whether subtraction occurred
    - Split ciphertexts using this mark into S_{yes} and S_{no}
  - **Correct d_t guess will have** $MeanTime(S_{yes}) > MeanTime(S_{no})$
  - **Incorrect d_t guess will have** $MeanTime(S_{yes}) \approx MeanTime(S_{no})$

Timing Attacks

Example 2



Timing Attacks

Example 3

- **Kyber** (or ML-KEM)
  - Post-quantum secure key encapsulation mechanism
  - Based on lattices
- Implementations used division on secret data
  - Division instruction is often variable-time (dividend)
- **KyberSlash**
  - Key recovery from variable-time division instruction

Timing Attacks

Example 3

Example (From KyberSlash)

```
void poly_tomsg(uint8_t msg[32], 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;
            /* Division by Kyber Prime */
            t = (((t << 1) + KYBER_Q/2)/KYBER_Q) & 1;
            msg[i] |= t << j;
        }
    }
}
```

Cache Attacks

When the attacker lives in your machine (kind of)

Cache Attacks

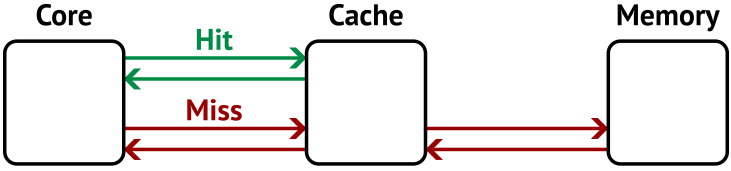
- Timing attacks where state of the cache is involved/probed by the attacker
- Cache attacks are microarchitectural attacks (μArch attacks)
  - μArch attacks: Utilize μArch state
    - Architectural state: registers, program counter, memory (what the ISA defines)
    - μArch state: caches, branch predictors, out-of-order execution (hardware implementation details)
  - Cache state is μArch state

Cache Attacks

Caches

Focusing on x86_64

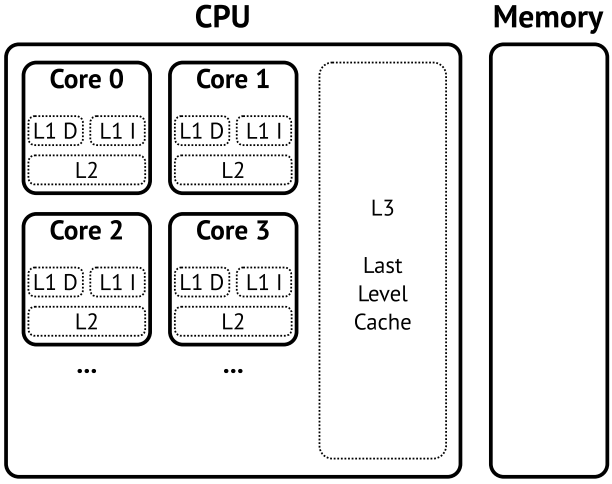
- Small banks of fast memory that store recently accessed memory values
  - Locality: It will likely be accessed again
- **Cache hit**: Value was read from cache, *fast*
- **Cache miss**: Value was not in cache, main memory needed, *slow*



Cache Attacks

Caches

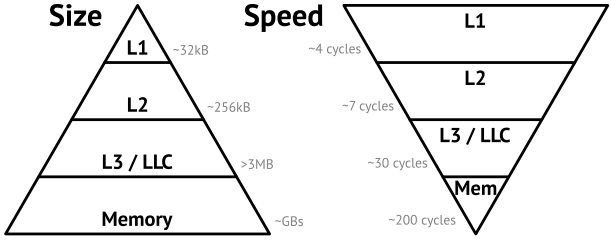
- Several levels (Cache hierarchy)
  - **L1**
    - L1-D for data
    - L1-I for instructions
    - Per-core
    - ~ 32KB
  - **L2**
    - Both instructions and data
    - Per-core
    - ~ 256KB
  - **L3/LLC**: Last Level Cache
    - Both instructions and data
    - Shared among cores
    - > 3MB



Cache Attacks

Caches

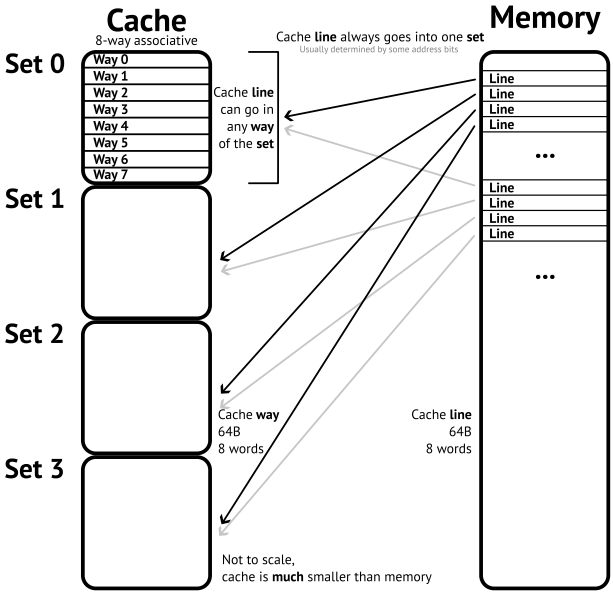
- Several levels (Cache hierarchy)
  - **L1**
    - L1-D for data
    - L1-I for instructions
    - Per-core
    - ~ 32KB
  - **L2**
    - Both instructions and data
    - Per-core
    - ~ 256KB
  - **L3/LLC**: Last Level Cache
    - Both instructions and data
    - Shared among cores
    - > 3MB



Cache Attacks

Cache structure and behavior

- **Cache line**: Block of memory that can be cached
- **Cache set**: Part of cache that stores cache **lines** in cache **ways**
- **Cache way**: Block of cache in a cache **set** that can contain a **line**
- **Cache eviction**: When a cache **set** is full and another line needs to be cached
  - The cache evicts the Least Recently Used (LRU) cache **line** from the **way**
- **Associativity**: Cache **line** can be in any way of a **set**
- **Inclusivity**: Higher level cache includes all contents of the lower levels
  - Eviction at higher level (LLC) causes eviction at lower levels
- **Exclusivity**: Higher level cache does not contain contents of the lower levels



Cache Attacks

General principle

1. Attacker executes for a bit: Prepares the cache state
2. Then victim executes (on the same core or not)
3. Attacker executes again: Observes the cache state

Attacker repeats this as often as they can

- **Spatial resolution**: What is the granularity of victim's memory accesses the attacker gets?
- **Temporal resolution**: What is the timing precision of the attack? How frequently can the attacker prepare and observe the state?

Cache Attacks

Prime+Probe

- Cache attacks and countermeasures: The case of AES
- Cache Missing for Fun and Profit

1. **Prime** step:
   - Attacker fills a cache set with their own data
2. Victim executes
3. **Probe** step:
   - Attacker measures time it takes to access data previously used to fill the cache set

- Victim access to cache set → Attacker data evicted → Slow probe
- No victim access to cache set → Attacker data present → Fast probe

---

- **Spatial resolution**: Cache set
- **Temporal resolution**: Limited (cache filling and probing takes time)
- Same-core
- Targets: L1-D, L1-I, LLC

Cache Attacks

Evict+Time

- Cache attacks and countermeasures: The case of AES

1. Victim executes operation
   - Cache now contains its working set
2. **Evict** step:
   - Attacker evicts a cache set
3. **Time** step:
   - Victim executes same operation again
   - Attacker measures victim runtime

- Victim access to cache set → Cache miss → Slow time
- No victim access to cache set → Cache hits → Fast time

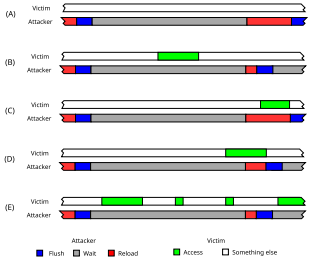
- **Spatial resolution**: Cache set
- **Temporal resolution**: Very limited (whole operation, synchronous)
- Same-core
- Targets: L1-D, (L1-I), LLC

Cache Attacks

Flush+Reload

- Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack
- Requires shared memory
  - Either shared libraries or memory deduplication

1. **Flush** step:
   - Attacker flushes a memory location from cache (`clflush`)
2. Victim executes
3. **Reload** step:
   - Attacker accesses a memory location previously flushed



- Victim access to cache line → Cache hit → Fast reload
- No victim access to cache line → Cache miss → Slow reload

- **Spatial resolution**: Cache line
- **Temporal resolution**: High (flush is fast, single address reload also fast)
- Cross-core
- Targets: LLC

Cache Attacks

Much more

- Flush+Flush
- Evict+Reload
- Prime+Abort
- Prime+Scope
- ...

Defences

How to not leak via timing/caches

Constant-time Programming

What to do/avoid

Assumes attacker gets

- Direction of every branch taken (cache attack: instructions)
- Address of every memory access (cache attack: data)
- Information on input to non-constant-time instructions (timing attack: instructions)

Constant-time programming

- Technique avoiding leaks in the above model
- **Constant-time code**
  - Avoids secret-dependent branches
  - Avoids secret-dependent memory accesses
  - Avoids secret inputs to certain variable-time instructions
    - Generally: Divide, sometimes shift, rotate, and multiply

Constant-time Programming

How

- Secret-dependent branch
  - Execute both, then select result in constant-time
- Secret-dependent memory access
  - Access all possible addresses, then select result in constant-time
- Secret inputs to certain instructions
  - Avoid the instructions

Constant-time Programming

Compiler woes

- Constant-timeness needs to hold **during execution**
  - Thus, on **binary level**, not only on source level
- **Compiler**
  - Modern compilers are incredibly powerful
  - They will *optimize* your constant-timeness away!
  - `ct_select`? Not on my watch!
  - What you get is what you C: Controlling side effects in mainstream C compilers
  - Breaking Bad: How Compilers Break Constant-Time Implementations
- **Interpreter**
  - All bets are off, interpreter may do what it wants
  - In practice, interpreted code *can* be constant-time
    - But it is hard to check
    - Interpreter needs to execute it in constant-time

Constant-time Programming

DOIT/DIT/Zkt

- Which instructions are actually constant-time?
  - Previously: Folklore
  - Now: Vendors provide a HW-SW contract
- **Intel DOIT**: Data Operand Independent Timing mode (DOITM)
  - Mode that has to be enabled
  - Let's DOIT: Using Intel's Extended HW/SW Contract for Secure Compilation of Crypto Code
- **Arm DIT**: Data Independent Timing (FEAT_DIT)
  - Mode that has to be enabled
  - Also on Apple silicon
- **RISC-V Zkt/Zvkt**: Lists of scalar (Zkt) and vector (Zvkt) instructions
  - Always-on when CPU advertises it

Constant-time Programming

Tools

How do you know your code is constant-time?

- Look at it and hope
- Or, use **tools**
- **Analysis levels**: Source, Intermediate Representation (IR), Binary
  - Lower level: More guarantees, less compiler issues
- **Techniques**: Dynamic analysis, Static analysis, Statistical
- **Simple idea**: Use Valgrind's memcheck (Timecop)
  - Mark secrets via `VALGRIND_MAKE_MEM_UNDEFINED(addr, len)`
  - Run Valgrind's memcheck
  - It will warn on
    - conditional jumps based on uninitialized data
    - memory accesses based on uninitialized data
  - Watch out for variable-time instructions

Takeaways

- How timing and cache attacks work
- How to defend against them
- How to verify the absence of timing leaks

Recommended reading

- For **this** lecture
  - Read Embedded Cryptography 1 - Chapter 1 - Timing Attacks - Section 1.2. Example attacks
    - Nice examples of timing attacks on cryptography
- For the **next** lecture
  - Read A Systematic Evaluation of Transient Execution Attacks and Defenses
    - A great overview of transient execution attacks

References

- https://timing.attacks.cr.yp.to/index.html
- Let's DOIT: Using Intel's Extended HW/SW Contract for Secure Compilation of Crypto Code
- Embedded Cryptography 1 - Chapter 1 - Timing attacks
- Embedded Cryptography 1 - Chapter 2 - Microarchitectural attacks
- A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware
- https://crocs-muni.github.io/ct-tools/