Microarchitectural Side-Channel Attacks

From the Basics to Transient Execution Attacks

Claudio Canella

June 17, 2018

IAIK – Graz University of Technology
Claudio Canella
PhD student @ Graz University of Technology

@cc0x1f

claudio.canella@iaik.tugraz.at
Side-Channel Attacks
• Safe software infrastructure does not mean safe execution
Side-Channel Attacks

- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware
Side-Channel Attacks

- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware
- Exploit unintentional information leakage by side-effects
Side-Channel Attacks

- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware
- Exploit unintentional information leakage by side-effects

Power consumption
Execution time
CPU caches

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
Caches and Cache Attacks
• Cache-based keylogging
CPU Cache Attacks

- Cache-based keylogging
- Crypto key recovery

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
CPU Cache Attacks

- Cache-based keylogging
- Crypto key recovery
  - various implementations (AES, RSA, ECC, ...)
  - up to 97% key bits recovered after 1 encryption

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
Cache-based keylogging

Crypto key recovery

- various implementations (AES, RSA, ECC, ...)
- up to 97% key bits recovered after 1 encryption

Cross-VM, cross-core, even cross-CPU
CPU Cache Attacks

- Cache-based keylogging
- Crypto key recovery
  - various implementations (AES, RSA, ECC, ...)
  - up to 97% key bits recovered after 1 encryption
- Cross-VM, cross-core, even cross-CPU
- Any CPU vendor
• using the *inclusive* property
Cross-Core Attacks?

- using the *inclusive* property
- last-level cache is a superset of L1 and L2
Cross-Core Attacks?

- using the *inclusive* property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache → evicted from L1 and L2
Cross-Core Attacks?

- using the *inclusive* property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache → evicted from L1 and L2
- a core can evict lines in the private L1 of another core
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);

Cache miss
Request
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
CPU Cache

```
printf("%d", i);
printf("%d", i);
```
CPU Cache

```
printf("%d", i);
```

Cache miss

```
printf("%d", i);
```

Cache hit

DRAM access, slow

Request

Response

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
printf("%d", i);
Cache miss Request
Response
printf("%d", i);
Cache hit
No DRAM access, much faster
DRAM access, slow
Flush+Reload

ATTACKER

Shared Memory

VICTIM

flush
access

access
Flush+Reload

ATTACKER

flush

access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access
Cache Template Attacks
Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - Called “Cache Template”
Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. "events" (keystrokes, encryptions, ...)
  - Called “Cache Template”

Exploitation Phase

- Monitor exploitable addresses
Profiling Phase

Attacker address space

Shared 0x0

Cache

Victim address space

Shared 0x0

Cache is empty

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
Profiling Phase

Attacker triggers an event
Profiling Phase

Attacker checks one address for cache hits ("Reload")
Profiling Phase

Update cache hit ratio (per event and address)
Profiling Phase

Attacker address space

Cache

Shared 0x0

Victim address space

Attacker flushes shared memory

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
Profiling Phase

Attacker address space

Shared 0x0

Cache

Victim address space

Shared 0x0

Repeat for higher accuracy
Profiling Phase

Repeat for all events
Profiling Phase

Attacker address space

Cache

Victim address space

Repeat for all events

Shared 0x0

Shared 0x0

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
Profiling Phase

Attacker address space

Cache

Victim address space

Shared 0x40

Continue with next address

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
Attacker address space

Shared 0x80

Cache

Victim address space

Shared 0x80

Continue with next address
Profiling Phase: All Events, All Addresses

Key

0x7c680 0x7c6c0 0x7c700 0x7c740 0x7c780 0x7c7c0 0x7c800 0x7c840 0x7c880 0x7c8c0 0x7c900 0x7c940 0x7c980 0x7c9c0 0x7ca00 0x7cb80 0x7cc40 0x7cc80 0x7ccc0 0x7cd00

ADDRESS

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
Exploitation Phase

- Monitor addresses from Cache Template
Exploitation Phase

- Monitor addresses from Cache Template
- Report to log file / attacker
Exploitation Phase

- Monitor addresses from Cache Template
- Report to log file / attacker
- Manual analysis of log file
  - Find password in keypress log, etc.
AES uses T-Tables (precomputed from S-Boxes)
AES uses T-Tables (precomputed from S-Boxes)

- 4 T-Tables
AES uses T-Tables (precomputed from S-Boxes)

- 4 T-Tables
- 
  \[
  T_0 \left[ k_{\{0,4,8,12\}} \oplus p_{\{0,4,8,12\}} \right] \\
  T_1 \left[ k_{\{1,5,9,13\}} \oplus p_{\{1,5,9,13\}} \right]
  \]
AES uses T-Tables (precomputed from S-Boxes)

- 4 T-Tables
- 
  \[ T_0 \left[ k_{\{0,4,8,12\}} \oplus p_{\{0,4,8,12\}} \right] \]
  \[ T_1 \left[ k_{\{1,5,9,13\}} \oplus p_{\{1,5,9,13\}} \right] \]
  
  ...

- If we know which entry of \( T \) is accessed, we know the result of \( k_i \oplus p_i \).
AES uses T-Tables (precomputed from S-Boxes)

- 4 T-Tables
- 

\[
T_0 \left[ k_{\{0,4,8,12\}} \oplus p_{\{0,4,8,12\}} \right]
\]
\[
T_1 \left[ k_{\{1,5,9,13\}} \oplus p_{\{1,5,9,13\}} \right]
\]

... 

- If we know which entry of \( T \) is accessed, we know the result of \( k_i \oplus p_i \).
- Known-plaintext attack (\( p_i \) is known) \( \rightarrow k_i \) can be determined
AES T-Table implementation from OpenSSL 1.0.2
AES T-Table implementation from OpenSSL 1.0.2

- Most addresses in two groups:
  - Cache hit ratio 100% (always cache hits)
  - Cache hit ratio 0% (no cache hits)
AES T-Table implementation from OpenSSL 1.0.2

- Most addresses in two groups:
  - Cache hit ratio 100% (always cache hits)
  - Cache hit ratio 0% (no cache hits)
- One 4096 byte memory block:
  - Cache hit ratio of 92%
  - Cache hits depend on key value and plaintext value
  - The T-Tables
AES T-Table Template Attack

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
AES T-Table Template Attack

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event
AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event
- Exploitation phase:
  - Eliminate key candidates
AES T-Table Template Attack

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event
- Exploitation phase:
  - Eliminate key candidates
  - Reduction of key space in first-round attack:
    - 64 bits after 16–160 encryptions
AES T-Table Template Attack

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event
- Exploitation phase:
  - Eliminate key candidates
  - Reduction of key space in first-round attack:
    - 64 bits after 16–160 encryptions
  - State of the art: full key recovery after 30000 encryptions
AES T-Table Template

Value of $p_0$

ADDRESS

$0$

255

$0 = 0x00$

ADDRESS

$0$

255

$0 = 0x55$

(transposed)
Transient Execution Attacks
```c
int width = 10, height = 5;

float diagonal = sqrt(width * width
                      + height * height);
int area = width * height;

printf("Area %d x %d = %d\n", width, height, area);
```
int width = 10, height = 5;

float diagonal = sqrt(width * width + height * height);

int area = width * height;

printf("Area %d x %d = %d\n", width, height, area);
Instructions

- are executed out-of-order
Instructions

- are executed out-of-order
- wait until their dependencies are ready
Out-of-Order Execution

Instructions

- are executed out-of-order
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
Out-of-Order Execution

Instructions

- are executed **out-of-order**
- wait until their **dependencies are ready**
  - Later instructions might execute prior earlier instructions
- retire in-order
Out-of-Order Execution

Instructions

- are executed out-of-order
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- retire in-order
  - State becomes architecturally visible

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
Out-of-Order Execution

Instructions

- are executed out-of-order
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- retire in-order
  - State becomes architecturally visible
- Exceptions are checked during retirement
Instructions

- are executed **out-of-order**
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- retire **in-order**
  - State becomes architecturally visible
- **Exceptions** are checked during retirement
  - Flush pipeline and recover state
• CPU tries to predict the future (branch predictor), ...
Speculative Execution

- CPU tries to predict the future (branch predictor), ...
  - ... based on events learned in the past
Speculative Execution

- CPU tries to predict the future (branch predictor), ...
  - ... based on events learned in the past
- Speculative execution of instructions
Speculative Execution

- CPU tries to predict the future (branch predictor), ...
  - ... based on events learned in the past
- Speculative execution of instructions
- If the prediction was correct, ...
• CPU tries to predict the future (branch predictor), ...
  • ...based on events learned in the past
• Speculative execution of instructions
• If the prediction was correct, ...
  • ...very fast
Speculative Execution

• CPU tries to predict the future (branch predictor), . . .
  • . . . based on events learned in the past
• Speculative execution of instructions
• If the prediction was correct, . . .
  • . . . very fast
  • otherwise: Discard results
Speculative Execution

- CPU tries to predict the future (branch predictor), ...
  - ... based on events learned in the past
- Speculative execution of instructions
- If the prediction was correct, ...
  - ... very fast
  - otherwise: Discard results
- Measurable side-effects?
• Out-of-order/speculatively executed instructions leave microarchitectural traces
• Out-of-order/speculatively executed instructions leave microarchitectural traces
  • We can see them for example in the cache
Transient: What's that?

• Out-of-order/speculatively executed instructions leave microarchitectural traces
  • We can see them for example in the cache
• Give such instructions a name: transient instructions
Transient: What’s that?

- Out-of-order/speculatively executed instructions leave microarchitectural traces
  - We can see them for example in the cache
- Give such instructions a name: transient instructions
- We can indirectly observe the execution of transient instructions
MELTDOWN

SPECTRE
Transient Execution Attacks: Classification

- **Spectre-type**
  - Spectre-PHT
  - Spectre-BTB
  - Spectre-RSB
  - Spectre-STL

- **Meltdown-type**
  - Meltdown-NM
  - Meltdown-AC
  - Meltdown-DE
  - Meltdown-PF
  - Meltdown-UD
  - Meltdown-SS
  - Meltdown-BR
  - Meltdown-GP

- **Prediction fault**
  - Cross-address-space in-place (IP) vs., out-of-place (OP)
  - Same-address-space

- **Microarchitectural buffer**
  - PHT-CA-IP *
  - PHT-CA-OP *
  - PHT-SA-IP
  - PHT-SA-OP *
  - BTB-CA-IP
  - BTB-CA-OP
  - BTB-SA-IP *
  - BTB-SA-OP *
  - RSB-CA-IP
  - RSB-CA-OP
  - RSB-SA-IP
  - RSB-SA-OP

- **Transient cause?**
  - Cross-address-space
  - Same-address-space

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
Meltdown vs. Spectre

<table>
<thead>
<tr>
<th>Operation #n</th>
<th>Flush pipeline on wrong prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prediction</td>
<td></td>
</tr>
<tr>
<td>Operation #n+2</td>
<td></td>
</tr>
<tr>
<td>Possibly architectural</td>
<td>Transient execution</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation #n</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
</tr>
<tr>
<td>Meltdown</td>
<td></td>
</tr>
<tr>
<td>Operation #n+2</td>
<td></td>
</tr>
<tr>
<td>Possibly architectural</td>
<td>Transient execution</td>
</tr>
</tbody>
</table>
Attacker loads inaccessible data into register
• Attacker loads inaccessible data into register → page-fault
• Attacker loads inaccessible data into register → page-fault
• Transient instruction stream accesses cache line based on loaded data
• Attacker loads inaccessible data into register → page-fault
• Transient instruction stream accesses cache line based on loaded data
• Determine accessed cache line using Flush+Reload
- Attacker loads inaccessible data into register $\rightarrow$ page-fault
- Transient instruction stream accesses cache line based on loaded data
- Determine accessed cache line using Flush+Reload
- Exploit lazy-enforcement of bits in the page-table entry
- OS must save registers on context switch
- OS must save registers on context switch → lazy-switching
• OS must save registers on context switch → lazy-switching
• Next FPU instruction causes #NM exception
- OS must save registers on context switch → lazy-switching
- Next FPU instruction causes #NM exception
- Execution continues with previous data in register
- OS must save registers on context switch → lazy-switching
- Next FPU instruction causes #NM exception
- Execution continues with previous data in register
- Extract data using Flush+Reload
x86 provides dedicated instruction raising #BR exception if bound-range is exceeded
x86 provides dedicated instruction raising #BR exception if bound-range is exceeded

Subsequent data is again used in transient execution
• x86 provides dedicated instruction raising #BR exception if bound-range is exceeded
• Subsequent data is again used in transient execution
• Attacker determines accessed cache line using Flush+Reload
Meltdown-BR

• x86 provides dedicated instruction raising #BR exception if bound-range is exceeded
• Subsequent data is again used in transient execution
• Attacker determines accessed cache line using Flush+Reload
• First Meltdown-type attack on AMD
Spectre: Mistraining Strategies

**Victim**
- Out of place/same address space
  - Aliased branch
    - Address collision
- In place/same address space
  - Spectre-vulnerable branch

**Attacker**
- Out of place/cross address space
  - Aliased address
    - Address collision
- In place/cross address space
  - Same address

**Shared Branch Prediction State**
index = 0;

char* data = "textKEY";

if (index < 4)
 then
 LUT[data[index] * 4096]
 Prediction
 else
 0
index = 0;
char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
0
index = 0;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
Speculate
Prediction
0
index = 0;

char* data = "textKEY";

if (index < 4) {
    LUT[data[index] * 4096]
} else {
    0
}
index = 1;

char* data = "textKEY";

if (index < 4)
  then
    Prediction
  else
    LUT[data[index] * 4096]
    0
index = 1;
char* data = "textKEY";

if (index < 4)
    Prediction
else
    LUT[data[index] * 4096] = 0

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
index = 1;

char* data = "textKEY";

if (index < 4)
    Speculate
    then
    LUT[data[index] * 4096]
    Prediction
    else
    0
index = 1;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
Prediction
0
index = 2;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
Prediction
0
index = 2;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

Prediction

0
index = 2;

char* data = "textKEY";

if (index < 4)
   Speculate
   then
      Prediction
   else
      0

LUT[data[index] * 4096]
index = 2;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
0
index = 3;

char* data = "textKEY";

if (index < 4)
    then
    LUT[data[index] * 4096]
    else
    Prediction
    0
index = 3;

char* data = "textKEY";

if (index < 4)
    then
        Prediction
    else
        LUT[data[index] * 4096] 0

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
index = 3;

char* data = "textKEY";

if (index < 4)
then
Speculate
LUT[data[index] * 4096]
Prediction
else
0
index = 3;

char* data = "textKEY";

if (index < 4)
    Prediction
else
    0

LUT[data[index] * 4096]
index = 4;

char* data = "textKEY";

if (index < 4)
    then
        LUT[data[index] * 4096]
    else
        Prediction

0
index = 4;

char* data = "textKEY";

if (index < 4)
  then
  LUT[data[index] * 4096]
  Prediction
  0
  else

index = 4;

char* data = "textKEY";

if (index < 4)
    LUT[data[index] * 4096]
else
    0
index = 4;

char* data = "textKEY";

if (index < 4)
    then
        LUT[data[index] * 4096]
    else
        Prediction
        Execute
        0
index = 5;

char* data = "textKEY";

if (index < 4)
  Prediction
else
  LUT[data[index] * 4096] = 0

Claudio Canella (@cc0x1f) — IAIK – Graz University of Technology
index = 5;

char* data = "textKEY";

if (index < 4)
    then
        Prediction
        LUT[data[index] * 4096]
    else
        0
index = 5;

char* data = "textKEY";

if (index < 4) {
    LUT[data[index] * 4096];
} else {
    Prediction
    0
}
index = 5;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
Execute

Prediction

0
index = 6;

char* data = "textKEY";

if (index < 4)
    LUT[data[index] * 4096]
else
    0
index = 6;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
Prediction
0
index = 6;

char* data = "textKEY";

if (index < 4)
    Speculate
    then
    LUT[data[index] * 4096]
else
    Prediction
    0
index = 6;

char* data = "textKEY";

if (index < 4)
    then
        LUT[data[index] * 4096]
    else
        Prediction
    Execute

0
Animal* a = bird;

a->move();

fly()  ↓

swim()  ↓

swim()  ↓

Prediction

LUT[data[index] * 4096]  0
Animal* a = bird;

a->move();

fly()

LUT[data[index] * 4096]

swim()

Prediction

Speculate

0
Animal* a = bird;

a->move()

fly()  
swim()  
swim()

LUT[data[index] * 4096]  
0
Animal* a = bird;

a->move();
Animal* a = bird;

a->move()

fly()

fly()

swim()

Prediction

LUT[data[index] * 4096] = 0
Animal* a = bird;

a->move();
Animal* a = bird;

a->move();

fly()

fly()

Prediction

LUT[data[index] * 4096]

0

swim()
Animal* a = fish;

a->move();

fly()

fly()

swim()

LUT[data[index] * 4096]

Prediction

0
Animal* a = fish;

a->move();
Animal* a = fish;

a->move()

fly()

LUT[data[index] * 4096]

Prediction

0

fly()

swim()
Animal* a = fish;

a->move()
Animal* a = fish;

a->move();

LUT[data[index] * 4096] 0
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

LUT[data[index] * 4096]

ignore
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

consider

Prediction

ignore

Speculate

LUT[data[index] * 4096]
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

LUT[data[index] * 4096]
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";
index = 1;

index = index & 0x3;  // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]  
LUT[data[index] * 4096]
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

LUT[data[index] * 4096]
```c
index = 2;
index = index & 0x3; // sanitization

char* data = "textKEY";
```

```
```
index = 2;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]
index = 2;

index = index & 0x3; // sanitization

char* data = "textKEY";

consider Prediction

LUT[data[index] * 4096]

Speculate

ignore Prediction

LUT[data[index] * 4096]
index = 2;

index = index & 0x3;  // sanitization

char* data = "textKEY";
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

ignore

LUT[data[index] * 4096]
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

ignore

Speculate
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

ignore

Prediction

LUT[data[index] * 4096]
index = 4;

index = index & 0x3;  // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]
index = 4;

index = index & 0x3;  // sanitization

char* data = "textKEY";
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

LUT[data[index] * 4096]

Execute

index = 0

Prediction

ignore
index = 5;

index = index & 0x3;  // sanitization

char* data = "textKEY";
index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]
index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]
index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]  
LUT[data[index] * 4096]
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";

index = 6;

index = index & 0x3;  // sanitization

char* data = "textKEY";
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";
• Small hardware stack storing return addresses of recent calls
• Small hardware stack storing return addresses of recent calls
• Redirect an indirect branch (a return in this case)
- Small hardware stack storing return addresses of recent calls
- Redirect an indirect branch (a return in this case)
- Fill buffer with “wrong” values
• Transient execution attacks present a new class of attacks
Transient execution attacks present a new class of attacks
Optimizations often have security implications
• Transient execution attacks present a new class of attacks
• Optimizations often have security implications
• Many problems to solve around transient execution attacks
Transient execution attacks present a new class of attacks
- Optimizations often have security implications
- Many problems to solve around transient execution attacks → still no satisfying solution
Thank you!
Microarchitectural Side-Channel Attacks

From the Basics to Transient Execution Attacks

Claudio Canella
June 17, 2018

IAIK – Graz University of Technology
Meltdown-PF: Meltdown-P [Van+18; Wei+18]

CPU micro-architecture

1. vadr
   - PT walk?
     - fail: Page fault
     - ok: EPT walk?
       - fail: EPCM fail
       - ok: SGX?
         - fail: Abort page
         - ok: Allow