RAGE AGAINST THE MACHINE CLEAR

A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks

Hany Ragab

Enrico Barberis

Herbert Bos

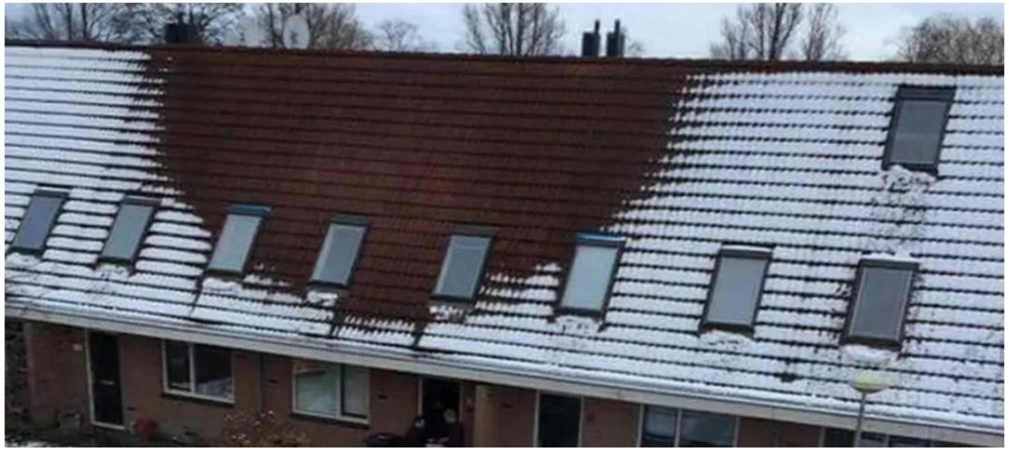
Cristiano Giuffrida

VUSec

Vrije Universiteit Amsterdam

Outline

Side Channels 101



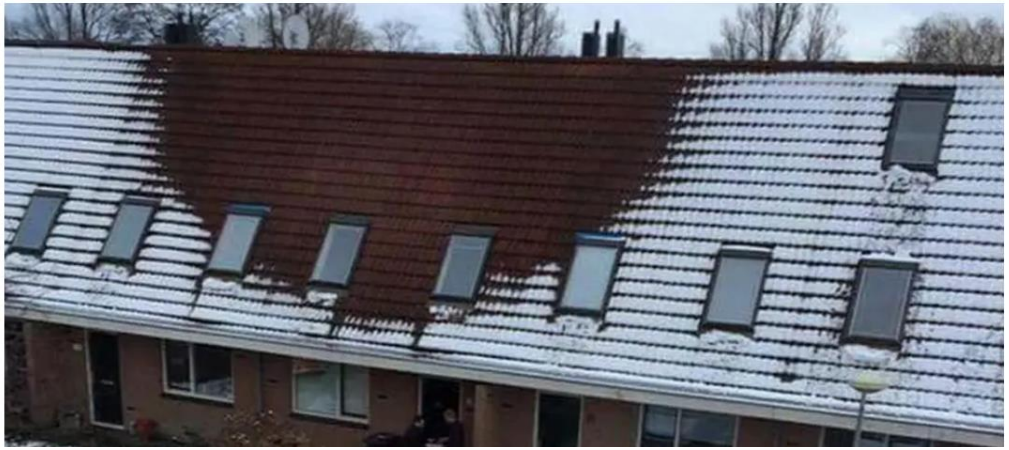
Side Channels 101

News > World > Europe

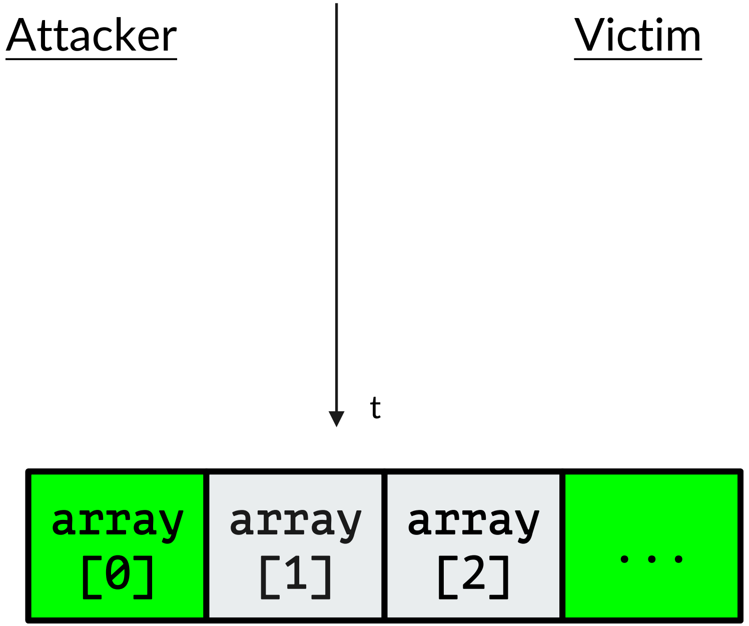
Melting snow being used by police to find cannabis farms in the Netherlands

Snow-free roofs can indicate the high temperatures needed to grow the drug

Lizzie Dearden | Tuesday 10 February 2015 13:31 | comments



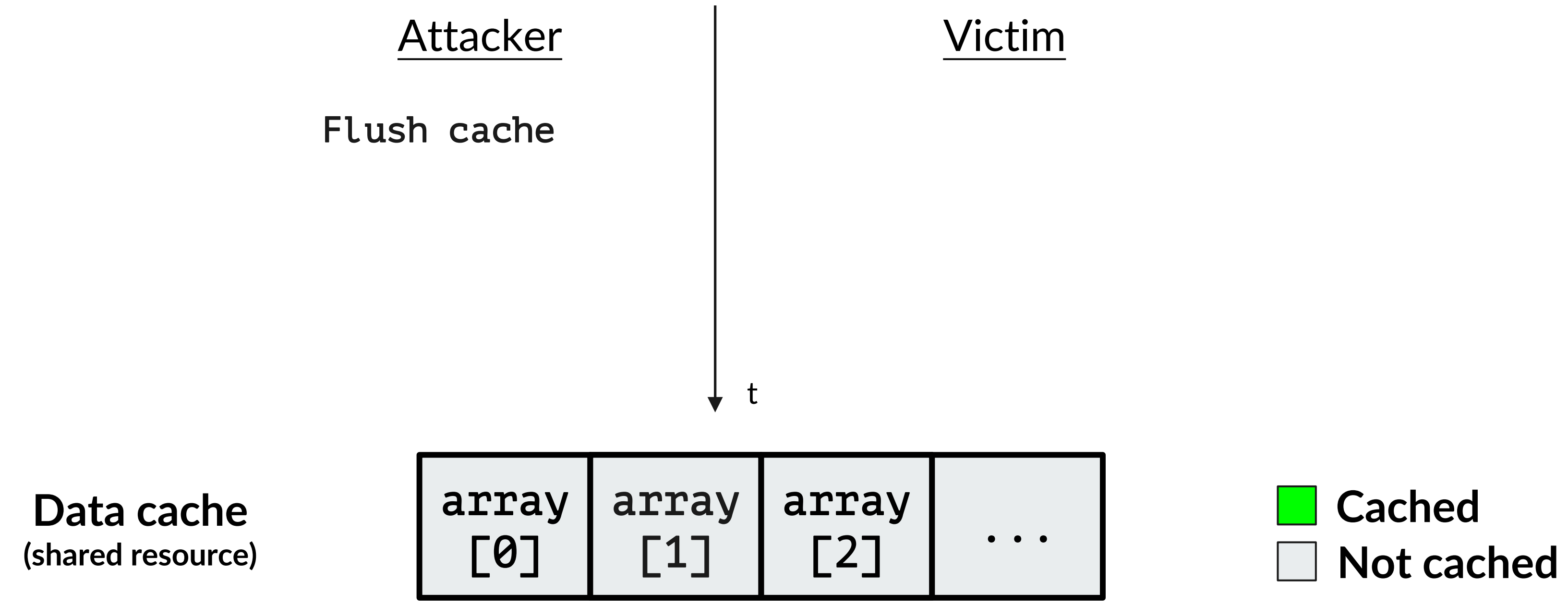
Flush+Reload Attack



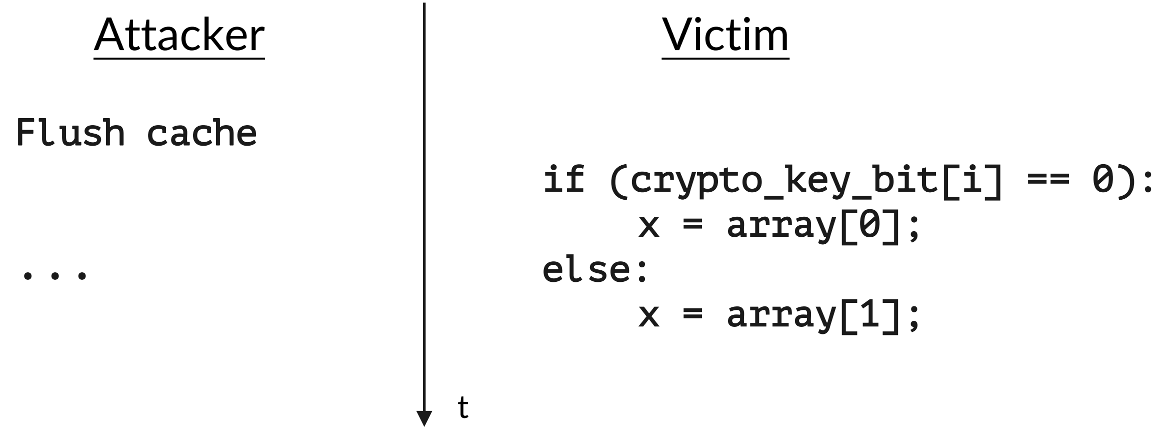
Data cache
(shared resource)

🟩 Cached
⬜ Not cached

Flush+Reload Attack



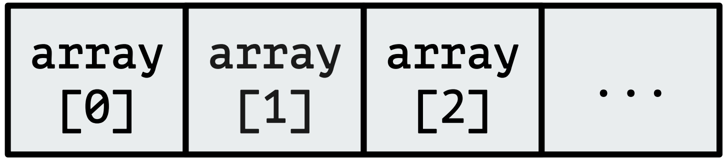
Flush+Reload Attack



```
Flush cache

...
```

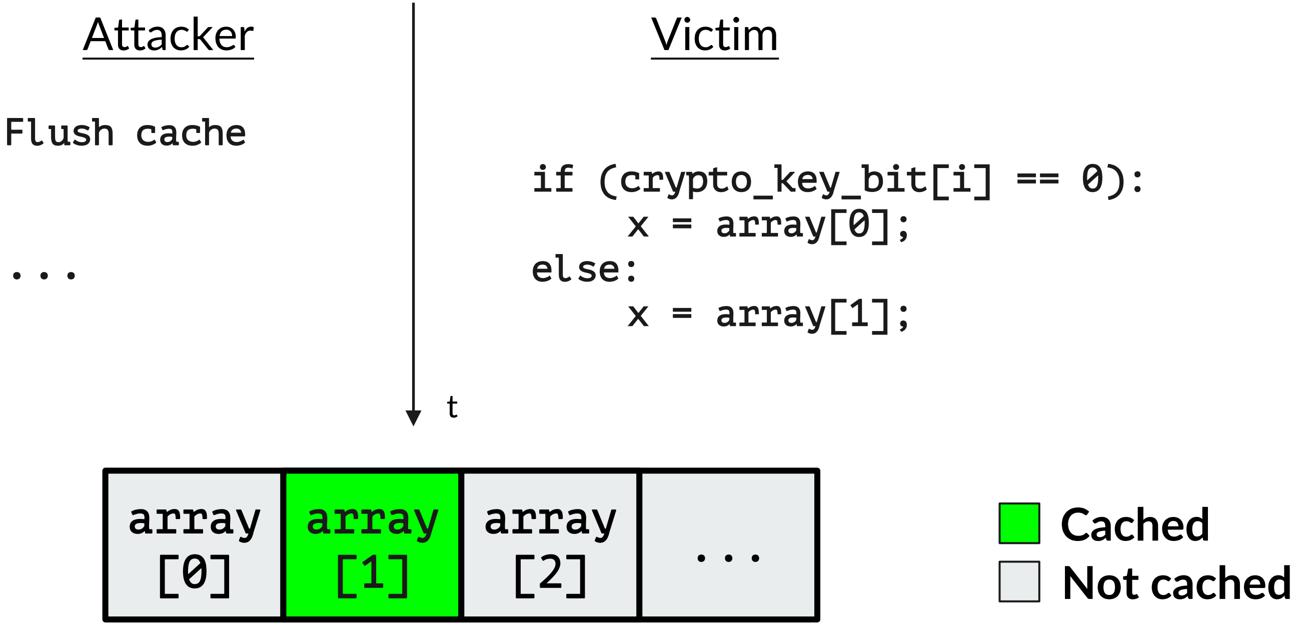
```
if (crypto_key_bit[i] == 0):
     x = array[0];
else:
     x = array[1];
```

Data cache
(shared resource)



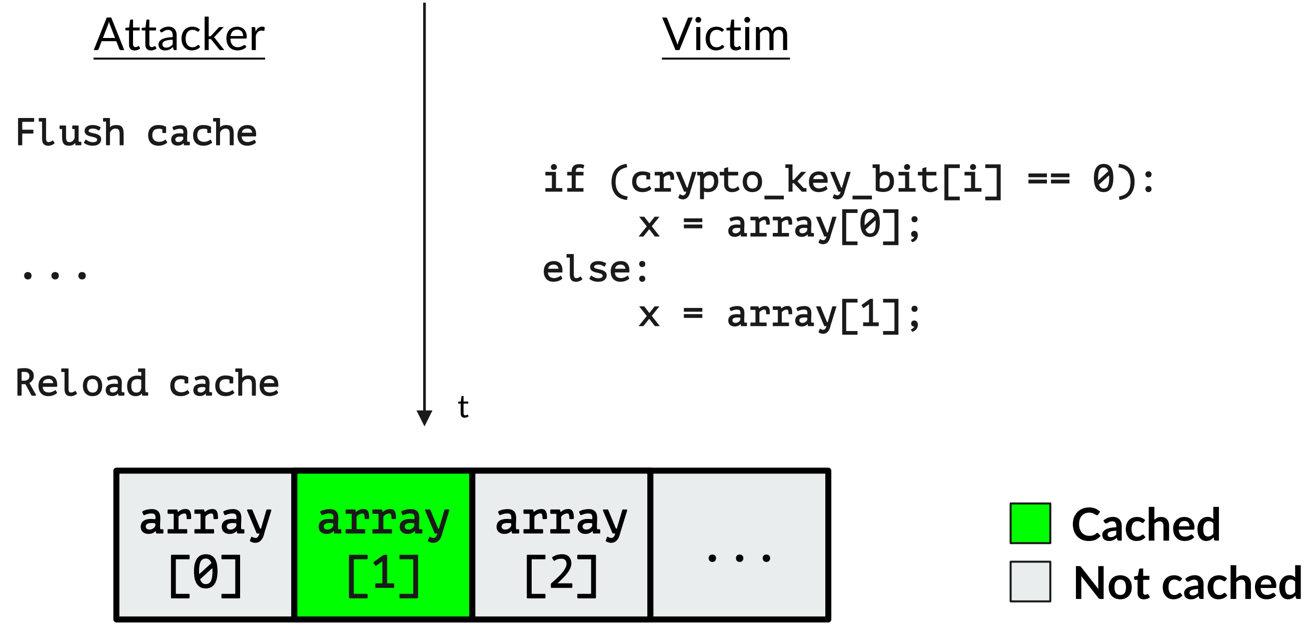
🟩 Cached
⬜ Not cached

Flush+Reload Attack



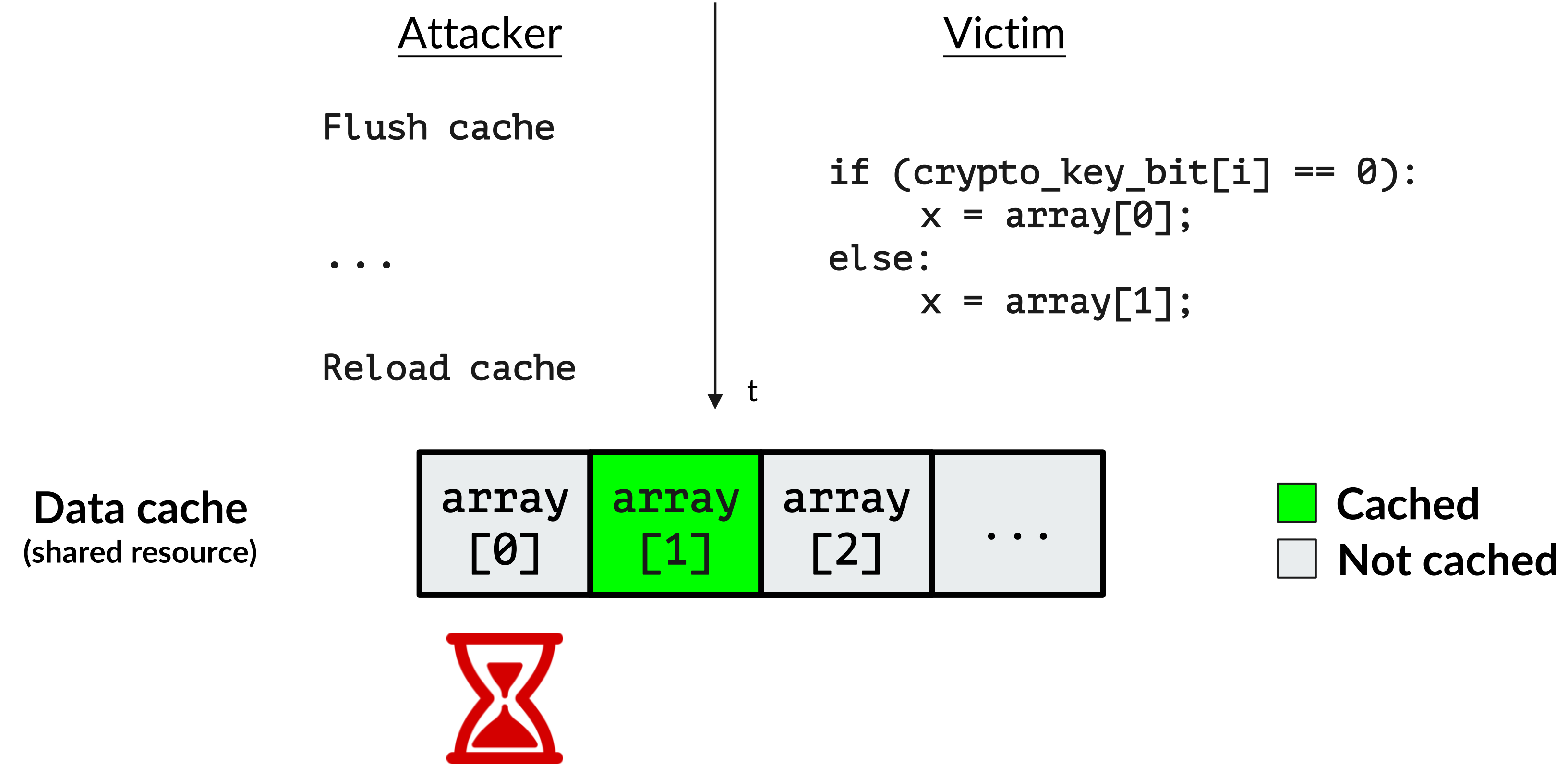
```
if (crypto_key_bit[i] == 0):
      x = array[0];
else:
      x = array[1];
```

Flush+Reload Attack



```
if (crypto_key_bit[i] == 0):
    x = array[0];
else:
    x = array[1];
```

Flush+Reload Attack



```
Flush cache

...

Reload cache
```

```
if (crypto_key_bit[i] == 0):
    x = array[0];
else:
    x = array[1];
```

Flush+Reload Attack

Attacker

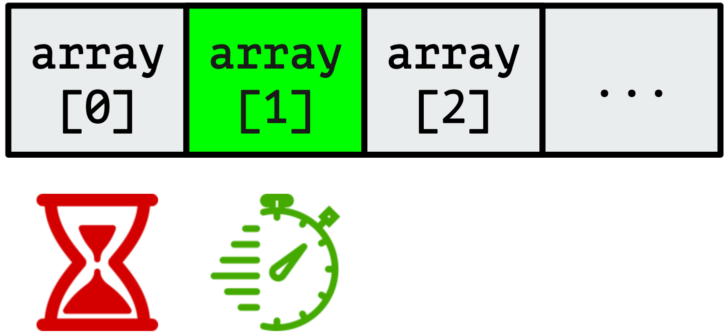
```
Flush cache

...

Reload cache
```

t

Victim

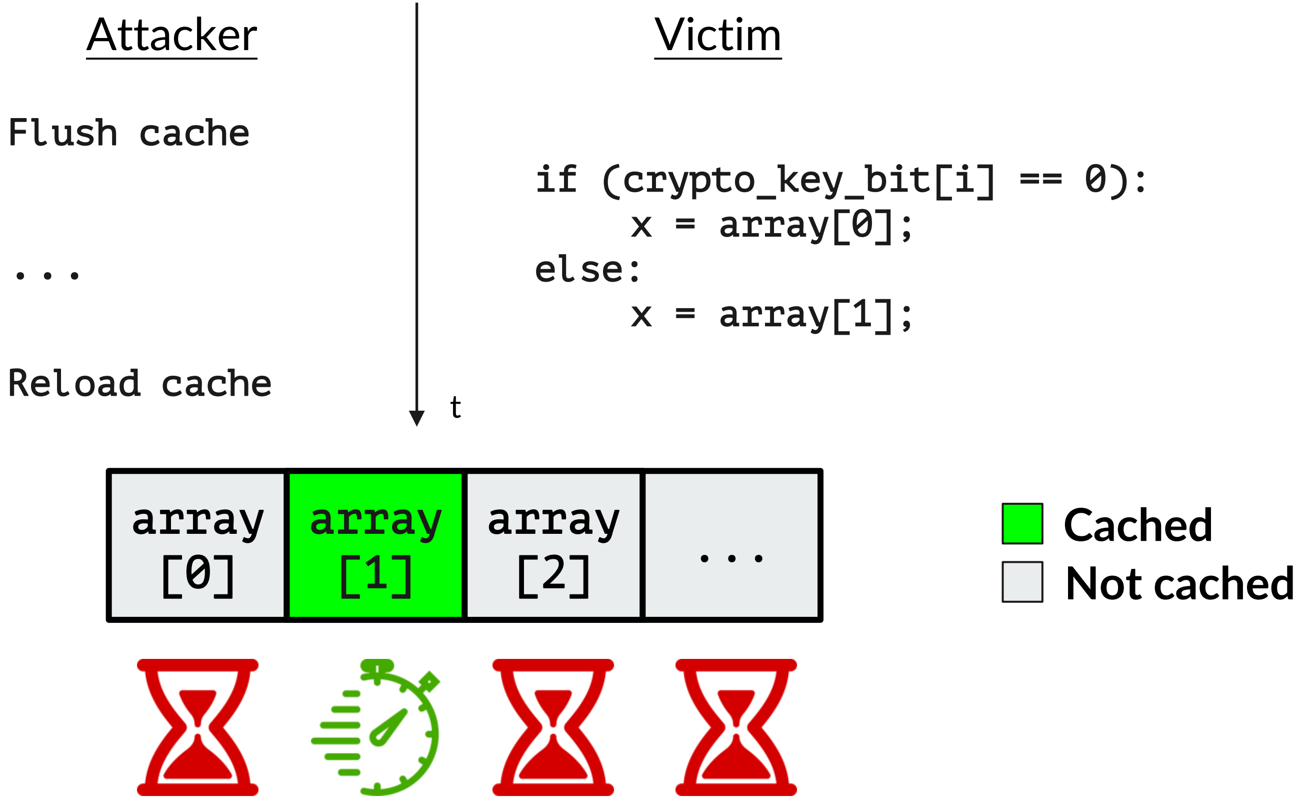
```
if (crypto_key_bit[i] == 0):
    x = array[0];
else:
    x = array[1];
```

Data cache
(shared resource)



🟩 Cached
⬜ Not cached

Flush+Reload Attack



Attacker:

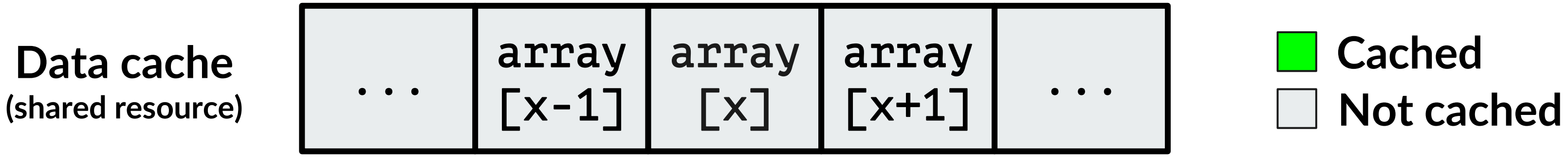
```
Flush cache

...

Reload cache
```

Victim:

```
if (crypto_key_bit[i] == 0):
    x = array[0];
else:
    x = array[1];
```

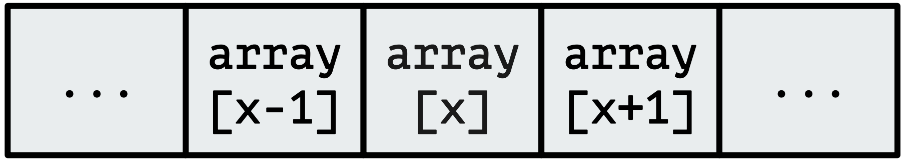
Transient Execution

```
if (x < array_size) {
    y = array[x]
}
```



Transient Execution

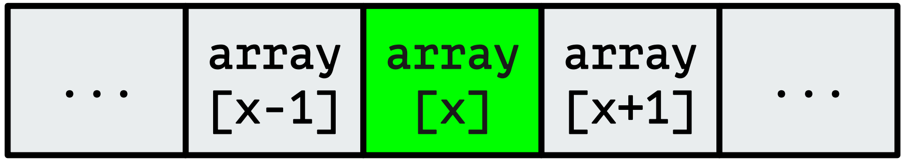
```
if (x < array_size) {
    y = array[x]
}
```



🟩 Cached
⬜ Not cached

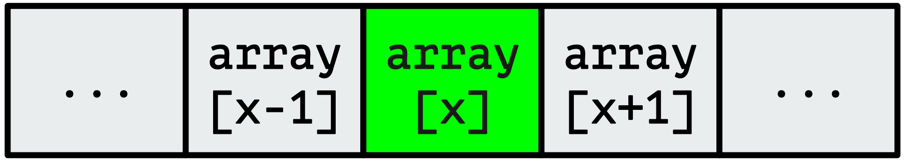
Transient Execution

```
if (x < array_size) {
    y = array[x]
}
```



Transient Execution

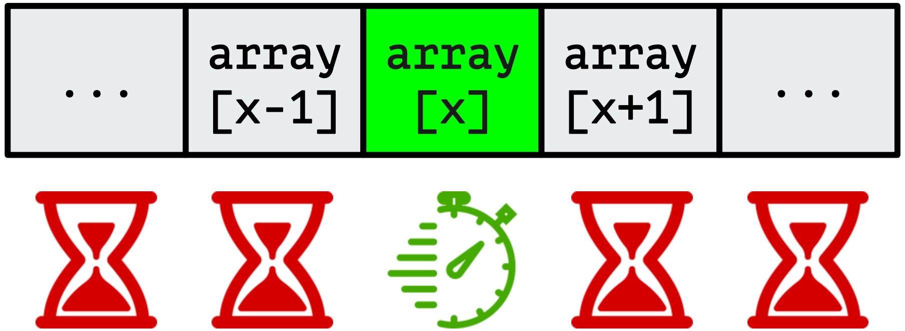
```
if (x < array_size) {
    ~~y = array[x]~~
}
```



🟩 Cached
⬜ Not cached

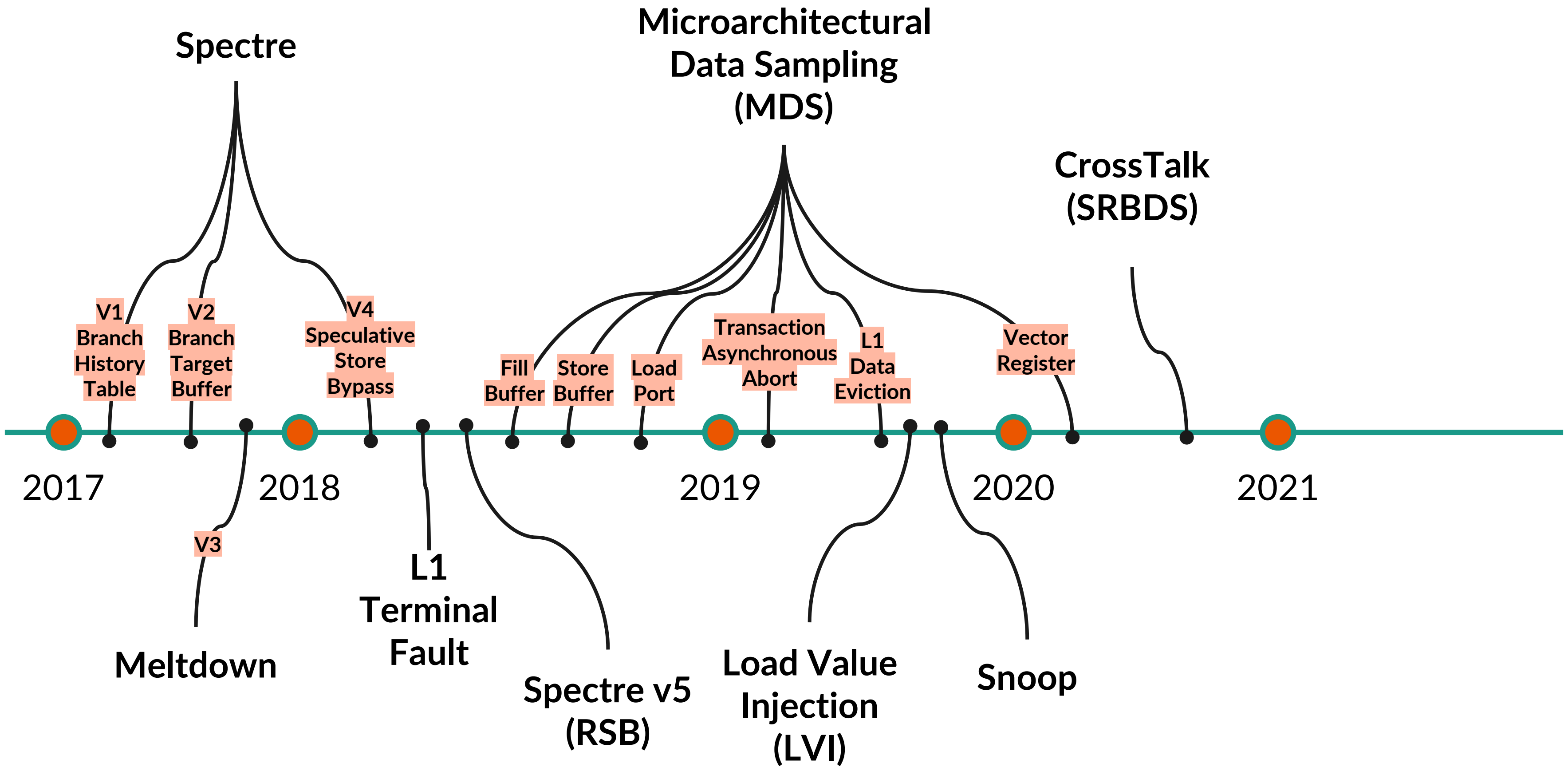
Transient Execution

```
if (x < array_size) {
    y = array[x]
}
```

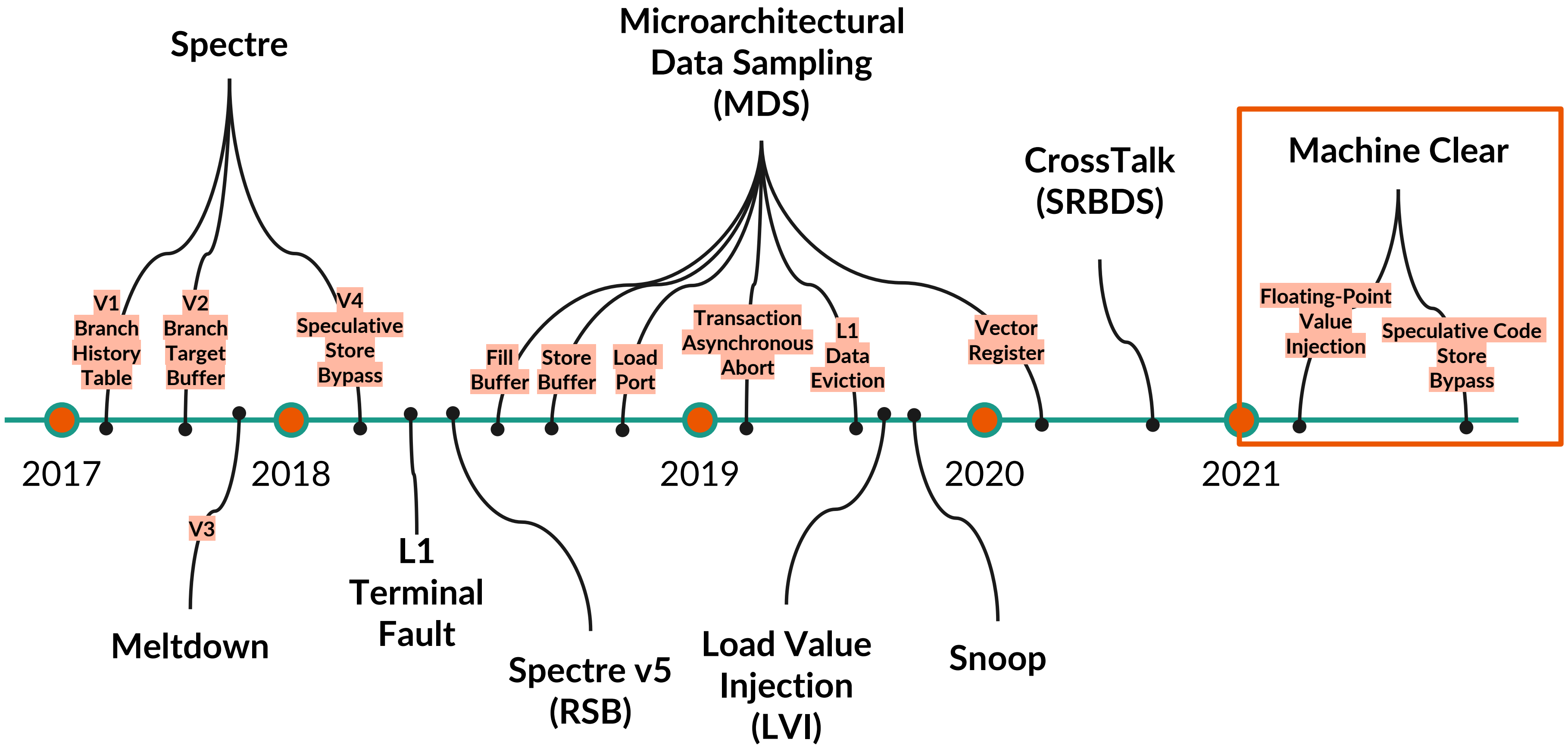


🟩 Cached
⬜ Not cached

Transient Execution Attacks



Transient Execution Attacks



Bad Speculation

The root cause of discarding issued µOps on x86 processors

Bad Speculation

The root cause of discarding issued µOps on x86 processors

Branch Misprediction

Bad Speculation

The root cause of discarding issued µOps on x86 processors

Branch Misprediction

Machine Clear

Bad Speculation

The root cause of discarding issued µOps on x86 processors

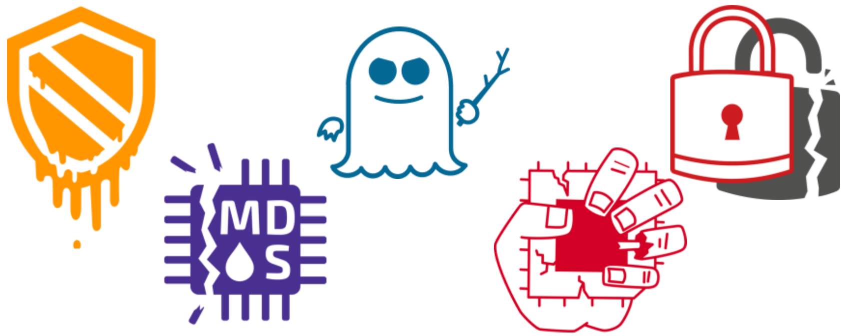
Branch Misprediction

Machine Clear

Bad Speculation

The root cause of discarding issued µOps on x86 processors

Branch Misprediction
& Faults & Intel TSX



Machine Clear

Bad Speculation

The root cause of discarding issued µOps on x86 processors

Branch Misprediction
& Faults & Intel TSX

Machine Clear

?

404 LOGO NOT FOUND

Rage Against The Machine Clear

Self-Modifying Code Machine Clear	Floating-Point Machine Clear
Memory Ordering Machine Clear	Memory Disambiguation Machine Clear

Rage Against The Machine Clear

Self-Modifying Code Machine Clear

Floating-Point Machine Clear

Rage Against The Machine Clear

Self-Modifying Code Machine Clear

↓

Speculative Code Store Bypass (SCSB)

Negligible mitigation overhead

Floating-Point Machine Clear

Rage Against The Machine Clear

Self-Modifying Code Machine Clear

↓

Speculative Code Store Bypass (SCSB)

Negligible mitigation overhead

Floating-Point Machine Clear

↓

Floating-Point Value Injection (FPVI)

53% Mitigation overhead

Rage Against The Machine Clear

Self-Modifying Code Machine Clear

Floating-Point Machine Clear

↓

End-to-end exploit leaking arbitrary memory in Firefox

With a leakage rate of **13 KB/s**

Security Analysis of Machine Clear

1. Architectural Invariant
2. Invariant Violation
3. Security Implications
4. Exploitation

SELF-MODIFYING CODE MACHINE CLEAR

Self-Modifying Code Machine Clear

Self-Modifying Code Machine Clear

Self-Modifying Code is a program storing instructions as data, modifying its own code as it is being executed

Self-Modifying Code Machine Clear

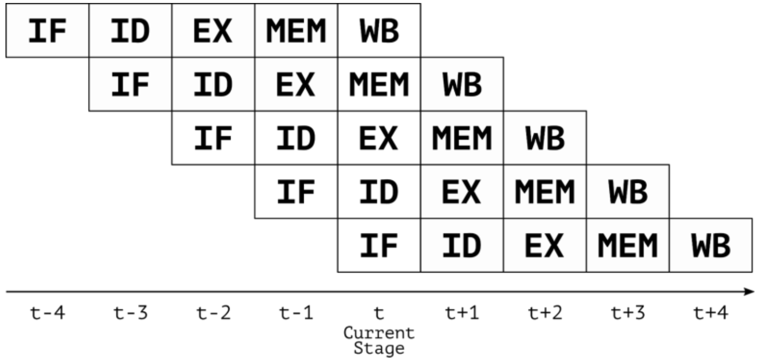
Self-Modifying Code is a program storing instructions as data, modifying its own code as it is being executed

```
i1: ...

i2: store nop @ i3

i3: load secret

i4: ...

i5: ...
```

Self-Modifying Code Machine Clear

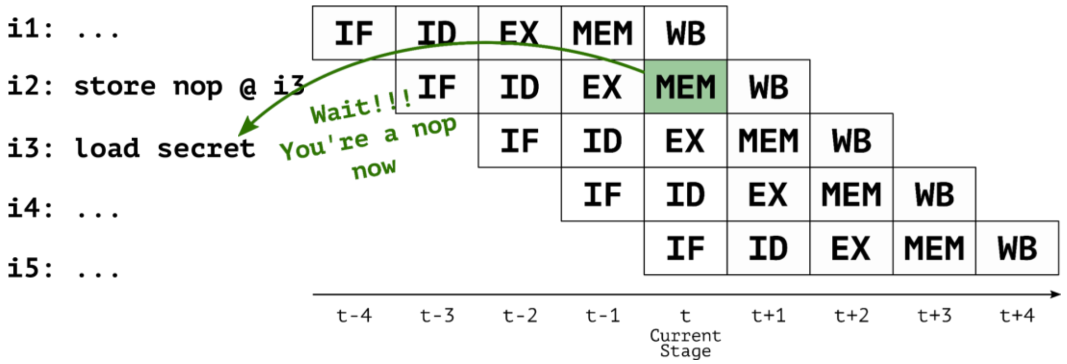
Self-Modifying Code is a program storing instructions as data, modifying its own code as it is being executed

```
i1: ...
i2: store nop @ i3
i3: load secret
i4: ...
i5: ...
```



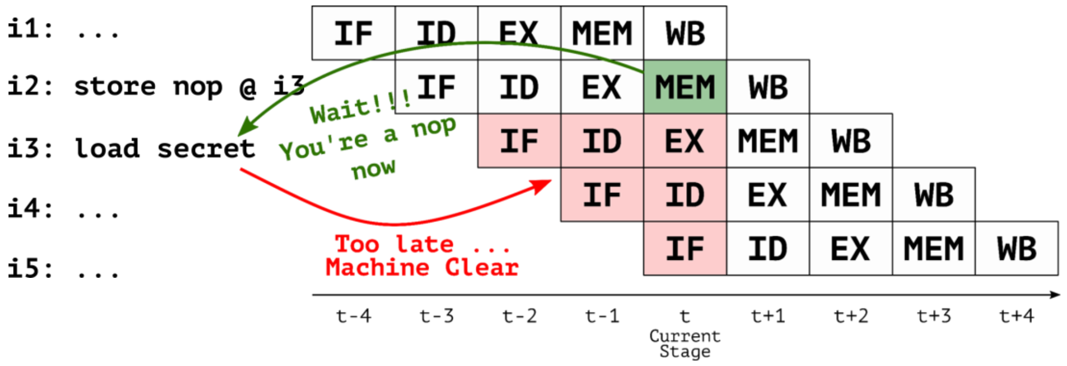
Self-Modifying Code Machine Clear

Self-Modifying Code is a program storing instructions as data, modifying its own code as it is being executed



Self-Modifying Code Machine Clear

Self-Modifying Code is a program storing instructions as data, modifying its own code as it is being executed



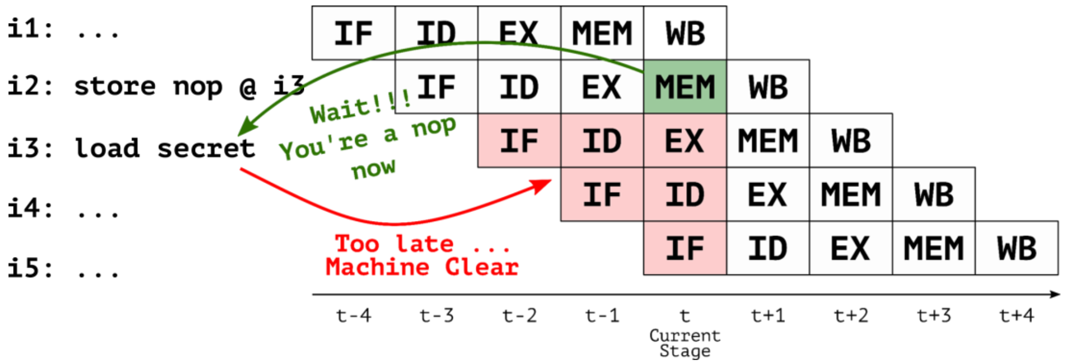
🟩 SMC Detection
🟥 Transiently Done

Self-Modifying Code Machine Clear

Self-Modifying Code is a program storing instructions as data, modifying its own code as it is being executed

Architectural Invariant

Stores always target data



🟩 SMC Detection
🟥 Transiently Done

Self-Modifying Code Machine Clear

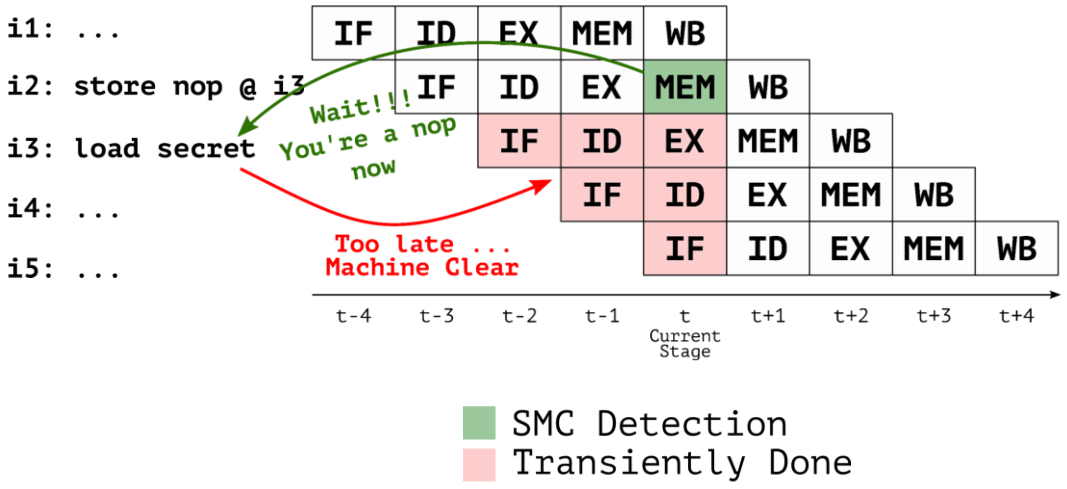
Self-Modifying Code is a program storing instructions as data, modifying its own code as it is being executed

Architectural Invariant

Stores always target data

Invariant Violation

Self-Modifying Code



Self-Modifying Code Machine Clear

Self-Modifying Code is a program storing instructions as data, modifying its own code as it is being executed

Architectural Invariant

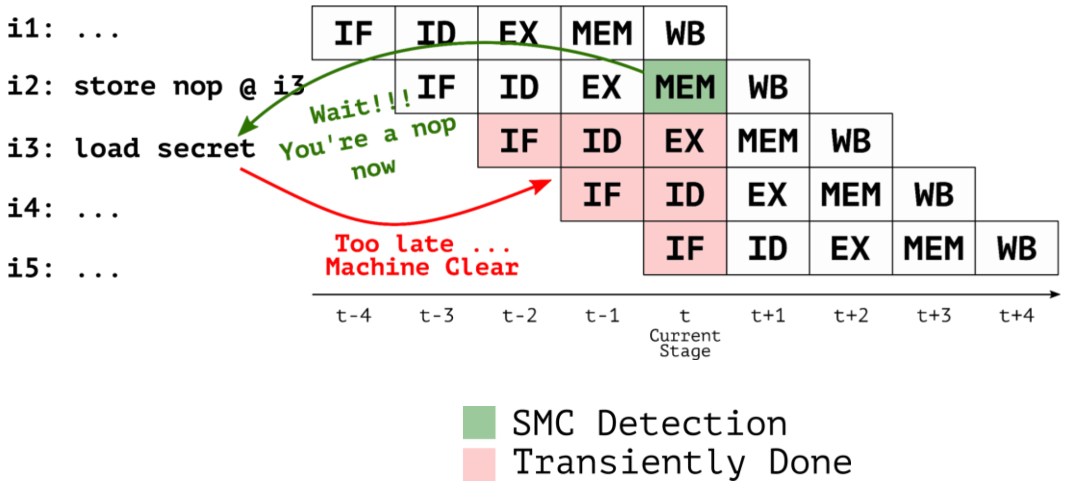
Stores always target data

Invariant Violation

Self-Modifying Code

Security Implications

Transiently execute stale code



Self-Modifying Code Machine Clear

> *Self-Modifying Code is a program storing instructions as data, modifying its own code as it is being executed*

Architectural Invariant

Stores always target data

Invariant Violation

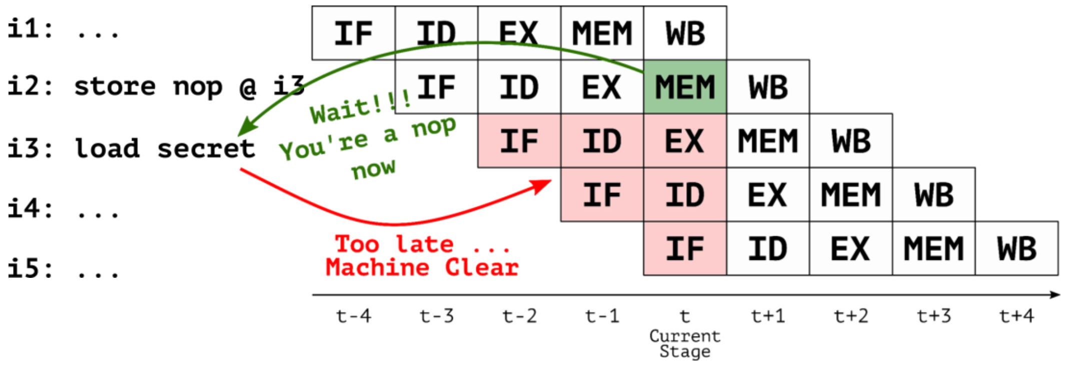
Self-Modifying Code

Security Implications

Transiently execute stale code

Exploitation

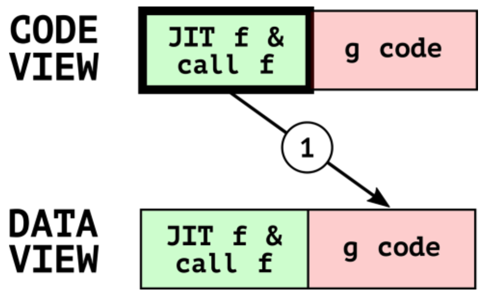
?



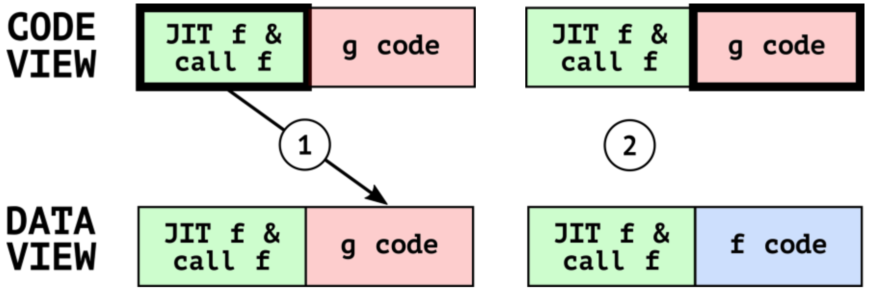
🟩 SMC Detection
🟥 Transiently Done

Speculative Code Store Bypass (SCSB)

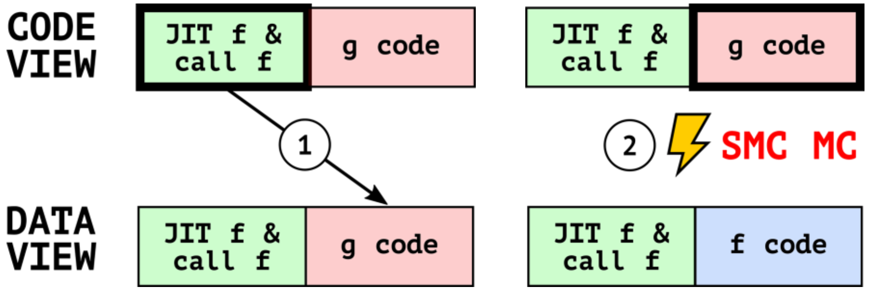
Speculative Code Store Bypass (SCSB)



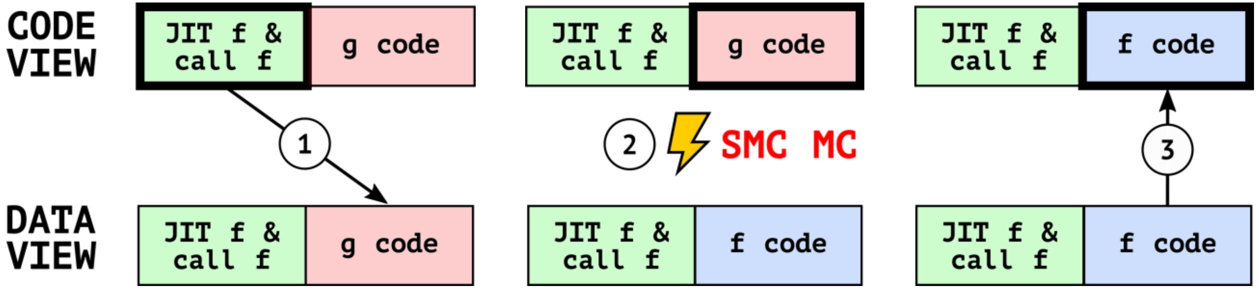
Speculative Code Store Bypass (SCSB)



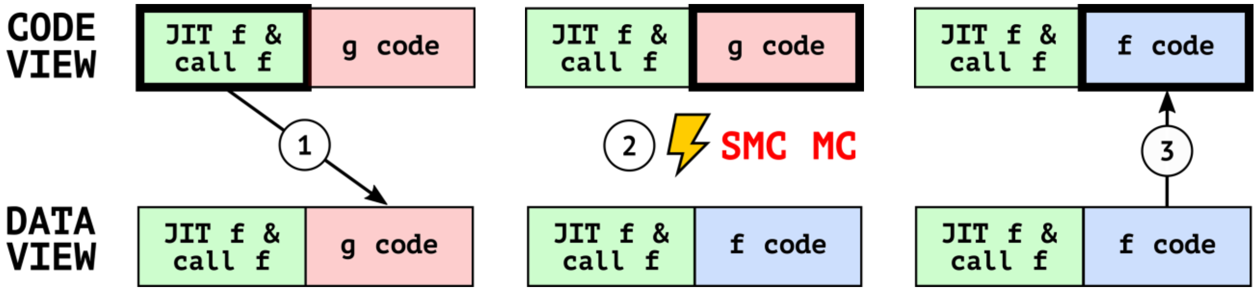
Speculative Code Store Bypass (SCSB)

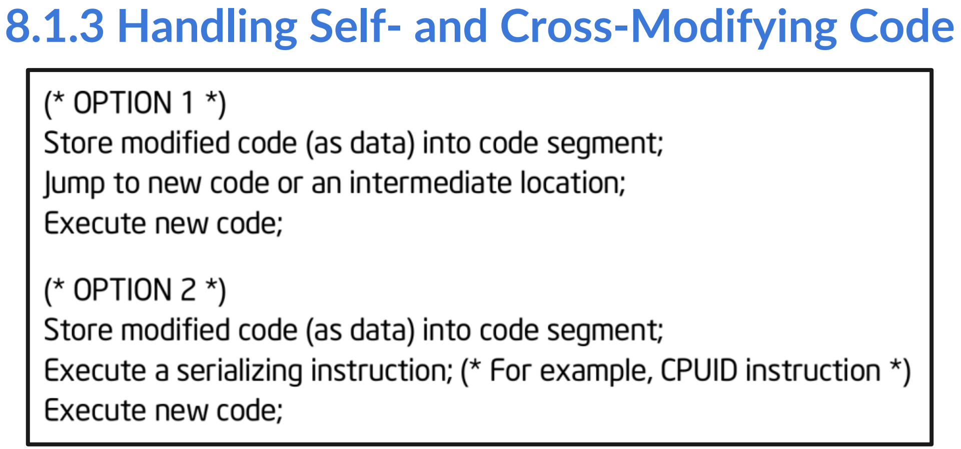


Speculative Code Store Bypass (SCSB)



Speculative Code Store Bypass (SCSB)

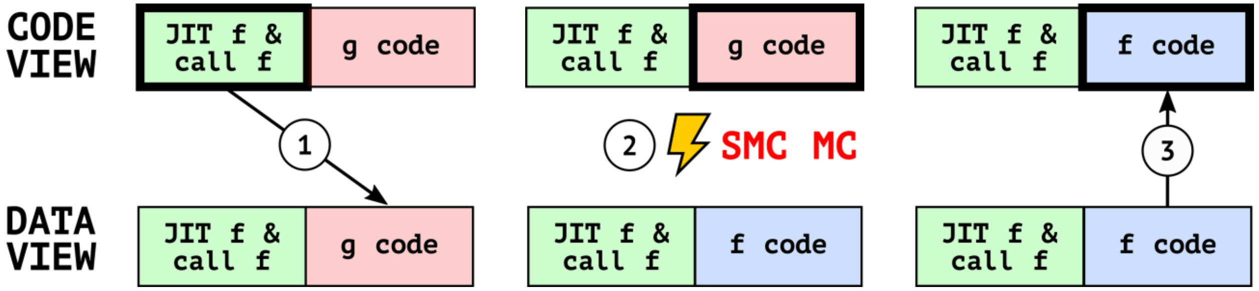


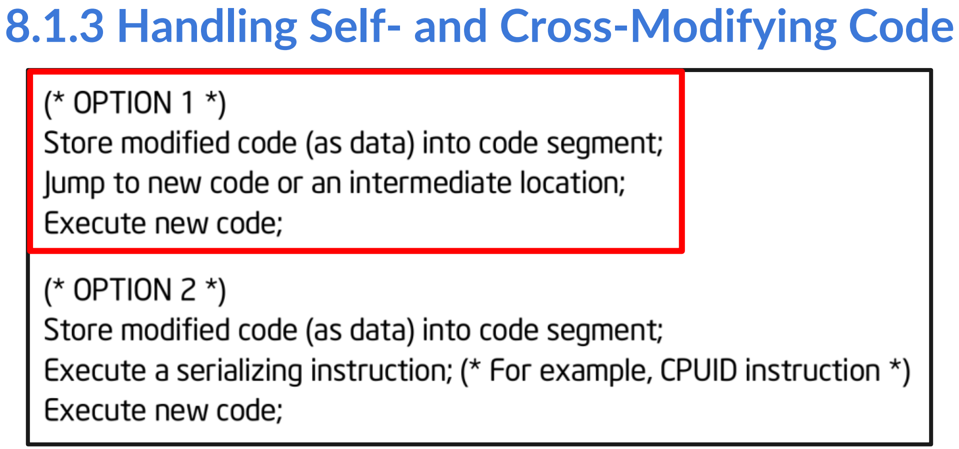

8.1.3 Handling Self- and Cross-Modifying Code

(* OPTION 1 *)
Store modified code (as data) into code segment;
Jump to new code or an intermediate location;
Execute new code;

(* OPTION 2 *)
Store modified code (as data) into code segment;
Execute a serializing instruction; (* For example, CPUID instruction *)
Execute new code;

Speculative Code Store Bypass (SCSB)

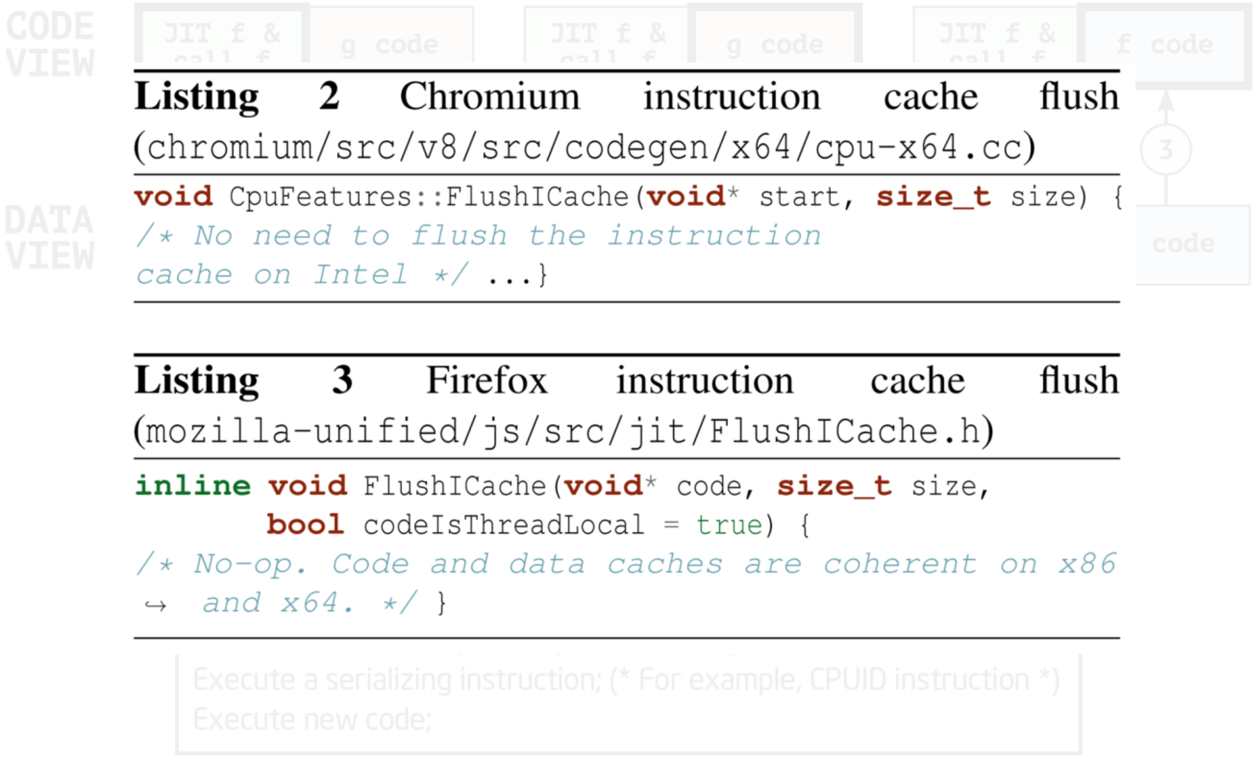




8.1.3 Handling Self- and Cross-Modifying Code

```
(* OPTION 1 *)
Store modified code (as data) into code segment;
Jump to new code or an intermediate location;
Execute new code;

(* OPTION 2 *)
Store modified code (as data) into code segment;
Execute a serializing instruction; (* For example, CPUID instruction *)
Execute new code;
```

Speculative Code Store Bypass (SCSB)



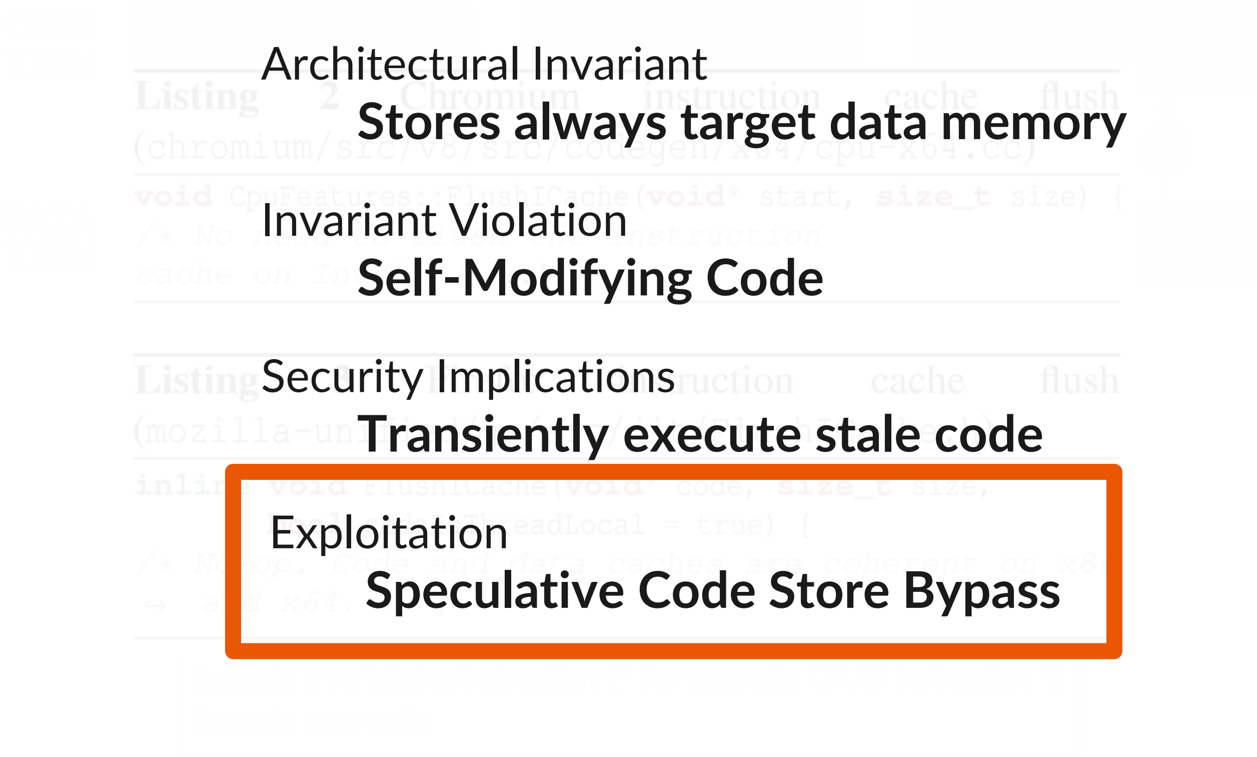
Listing 2 Chromium instruction cache flush (chromium/src/v8/src/codegen/x64/cpu-x64.cc)

```
void CpuFeatures::FlushICache(void* start, size_t size) {
/* No need to flush the instruction
cache on Intel */ ...}
```

Listing 3 Firefox instruction cache flush (mozilla-unified/js/src/jit/FlushICache.h)

```
inline void FlushICache(void* code, size_t size,
       bool codeIsThreadLocal = true) {
/* No-op. Code and data caches are coherent on x86
↪  and x64. */ }
```

Speculative Code Store Bypass (SCSB)



Architectural Invariant

Stores always target data memory

Invariant Violation

Self-Modifying Code

Security Implications

Transiently execute stale code

Exploitation

Speculative Code Store Bypass

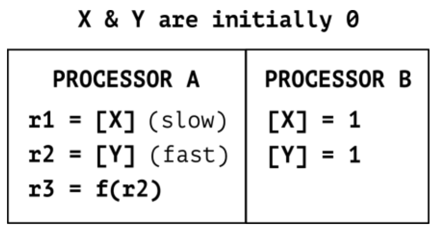
MEMORY ORDERING MACHINE CLEAR

Memory Ordering Machine Clear

A Total Store Order memory model guarantees that all CPU cores see all memory operations as the program order, except one case: A store instruction followed by a load instruction operating on different addresses may be reordered

Memory Ordering Machine Clear

A Total Store Order memory model guarantees that all CPU cores see all memory operations as the program order, except one case: A store instruction followed by a load instruction operating on different addresses may be reordered



X & Y are initially 0

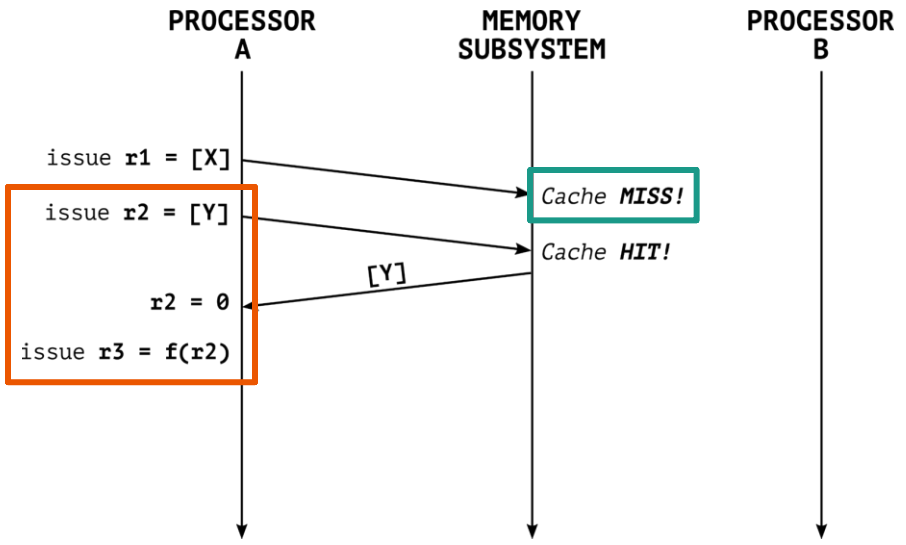
PROCESSOR A	PROCESSOR B
r1 = [X] (slow)	[X] = 1
r2 = [Y] (fast)	[Y] = 1
r3 = f(r2)	

Memory Ordering Machine Clear

A Total Store Order memory model guarantees that all CPU cores see all memory operations as the program order, except one case: A store instruction followed by a load instruction operating on different addresses may be reordered

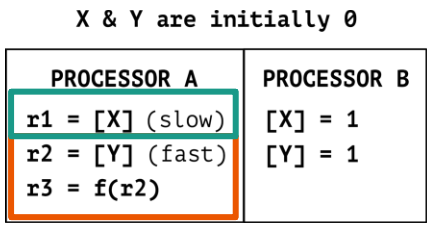
X & Y are initially 0

PROCESSOR A	PROCESSOR B
r1 = [X] (slow)	[X] = 1
r2 = [Y] (fast)	[Y] = 1
r3 = f(r2)	



Memory Ordering Machine Clear

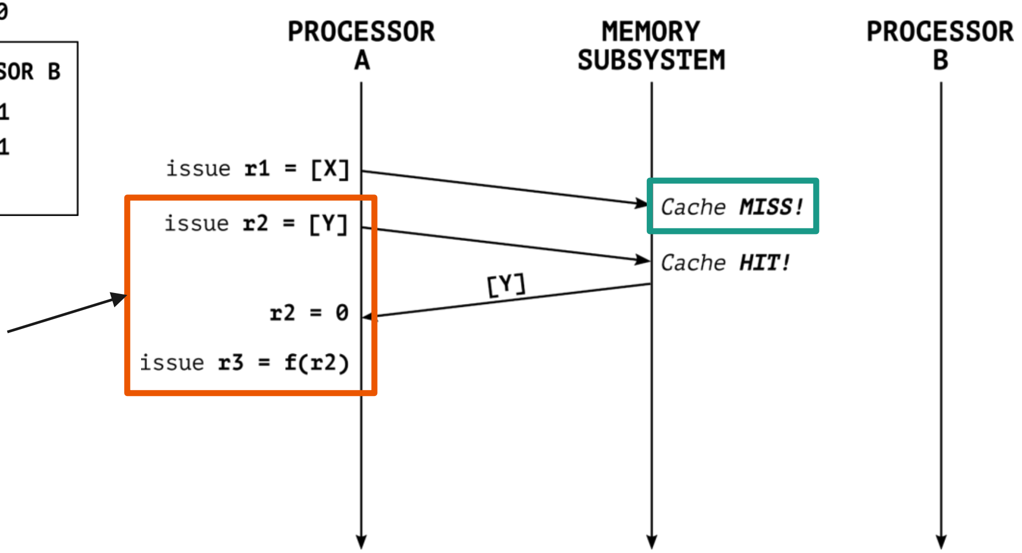
A Total Store Order memory model guarantees that all CPU cores see all memory operations as the program order, except one case: A store instruction followed by a load instruction operating on different addresses may be reordered



```
X & Y are initially 0
```

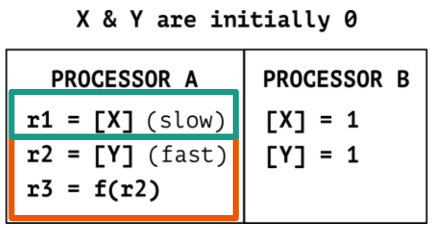
PROCESSOR A	PROCESSOR B
`r1 = [X] (slow)`	`[X] = 1`
`r2 = [Y] (fast)`	`[Y] = 1`
`r3 = f(r2)`	

Ready-to-commit
Waiting for r1=[x]
to reflect the TSO



Memory Ordering Machine Clear

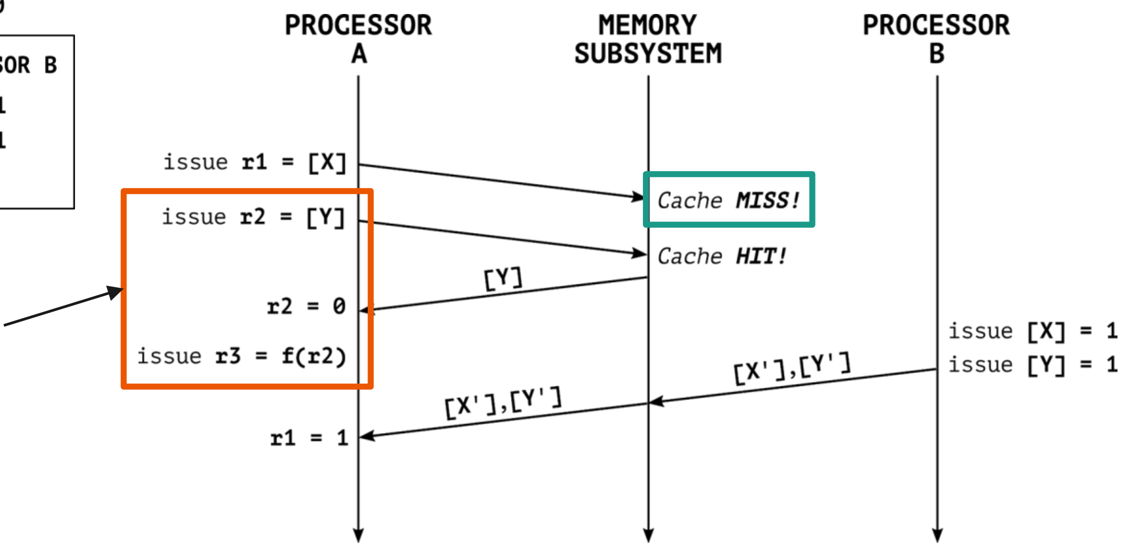
A Total Store Order memory model guarantees that all CPU cores see all memory operations as the program order, except one case: A store instruction followed by a load instruction operating on different addresses may be reordered



X & Y are initially 0

PROCESSOR A	PROCESSOR B
r1 = [X] (slow)	[X] = 1
r2 = [Y] (fast)	[Y] = 1
r3 = f(r2)	

Ready-to-commit
Waiting for r1=[x]
to reflect the TSO



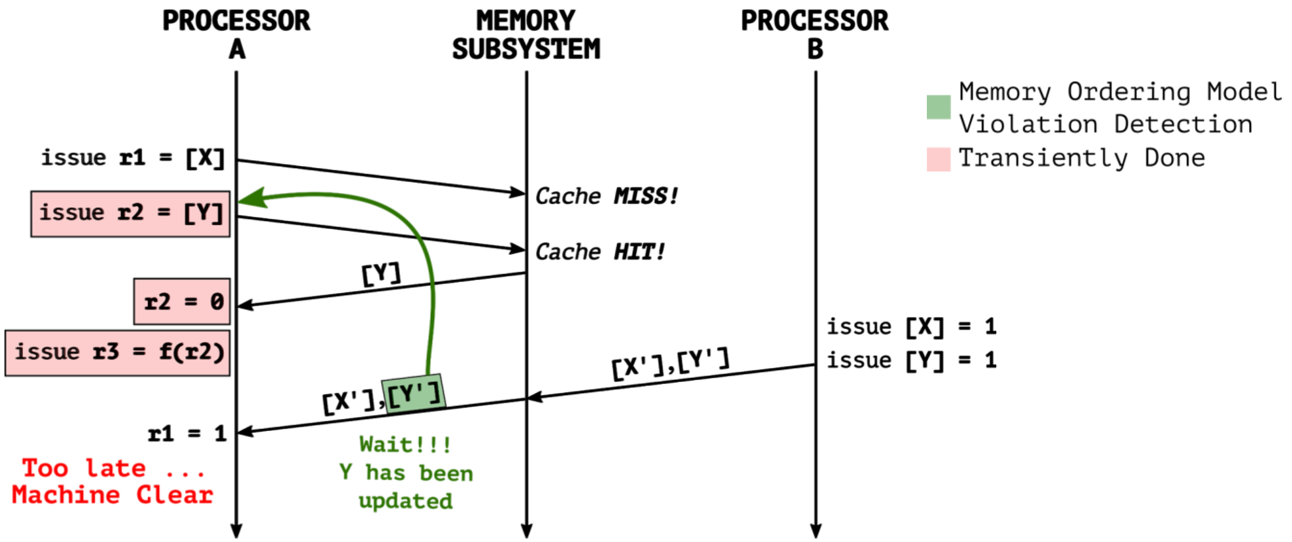
Memory Ordering Machine Clear

A Total Store Order memory model guarantees that all CPU cores see all memory operations as the program order, except one case: A store instruction followed by a load instruction operating on different addresses may be reordered

X & Y are initially 0

PROCESSOR A	PROCESSOR B

r1 = [X] (slow)
r2 = [Y] (fast)
r3 = f(r2)
	[X] = 1
	[Y] = 1



Memory Ordering Machine Clear

Architectural Invariant

OoO execution always complies with TSO

Invariant Violation

Memory ordering model violation

Security Implications

Transiently leak stale data

Exploitation

Non-trivial due to strict synchronization requirements

FLOATING-POINT MACHINE CLEAR

Floating-Point Machine Clear

Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. 2^-1022)

Floating-Point Machine Clear

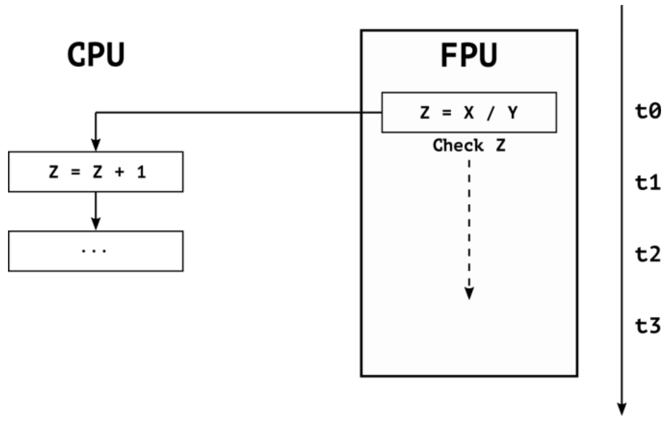
Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. 2^-1022)

```
i1: Z = X / Y
i2: Z = Z + 1
i3: ...
```

Floating-Point Machine Clear

Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. 2^-1022)

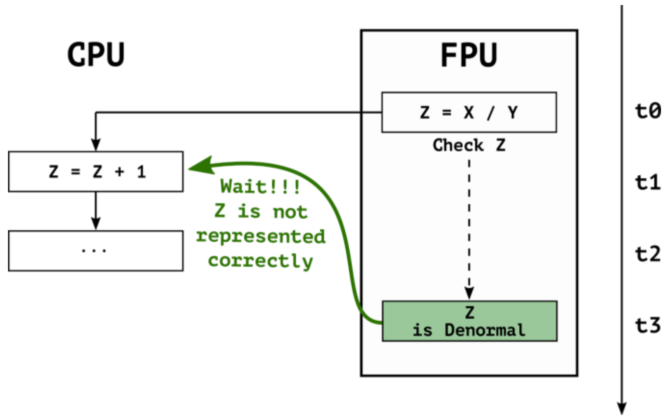
```
i1: Z = X / Y
i2: Z = Z + 1
i3: ...
```



Floating-Point Machine Clear

Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. 2^-1022)

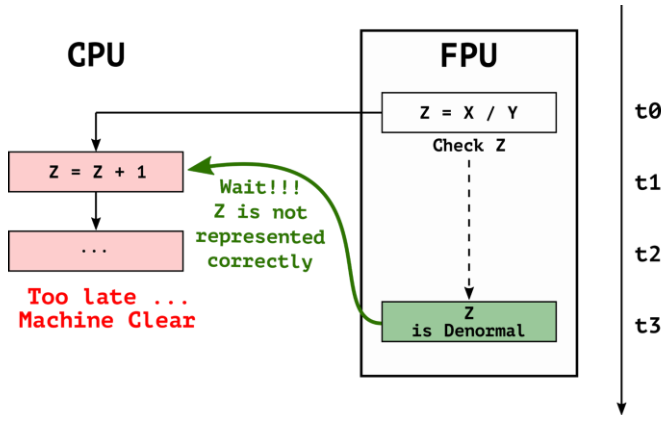
```
i1: Z = X / Y
i2: Z = Z + 1
i3: ...
```



Floating-Point Machine Clear

Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. 2^-1022)

```
i1: Z = X / Y
i2: Z = Z + 1
i3: ...
```



🟩 FP Denormal Detection
🟥 Transiently Done

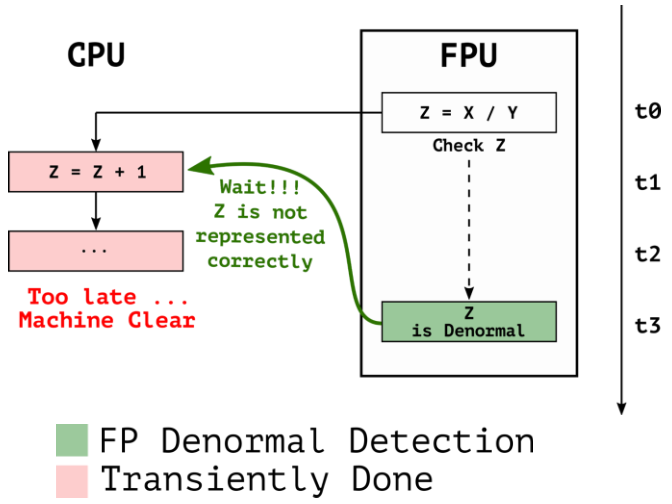
Floating-Point Machine Clear

Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. 2^-1022)

Architectual Invariant

FPU always operates on normal numbers

```
i1: Z = X / Y
i2: Z = Z + 1
i3: ...
```



Floating-Point Machine Clear

Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. 2^-1022)

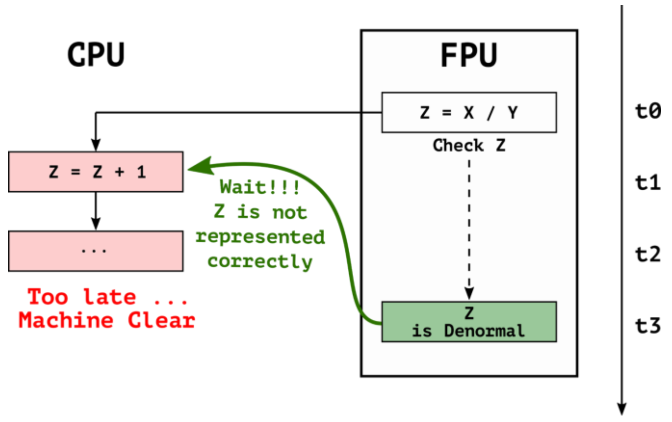
Architectural Invariant

FPU always operates on normal numbers

Invariant Violation

Subnormal FP operations

```
i1: Z = X / Y
i2: Z = Z + 1
i3: ...
```


CPU
FPU
Z = X / Y
Check Z
Z = Z + 1
...
Wait!!! Z is not represented correctly
Too late ... Machine Clear
Z is Denormal
t0
t1
t2
t3

🟩 FP Denormal Detection
🟥 Transiently Done

Floating-Point Machine Clear

Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. 2^-1022)

Architectural Invariant

FPU always operates on normal numbers

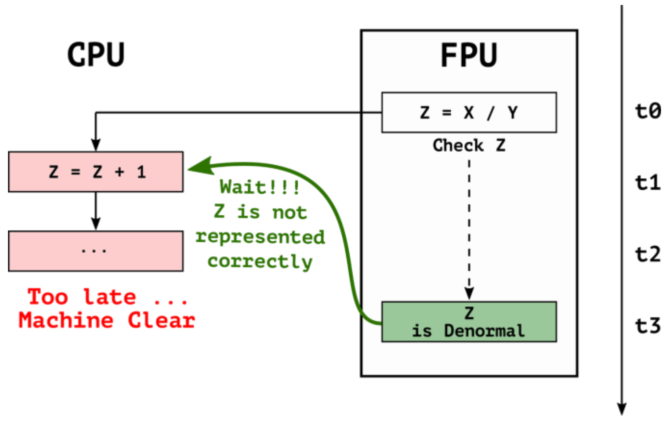
Invariant Violation

Subnormal FP operations

Security Implications

Transiently inject arbitrary FP values

```
i1: Z = X / Y
i2: Z = Z + 1
i3: ...
```



🟩 FP Denormal Detection
🟥 Transiently Done

Floating-Point Machine Clear

Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. 2^-1022)

Architectural Invariant

FPU always operates on normal numbers

Invariant Violation

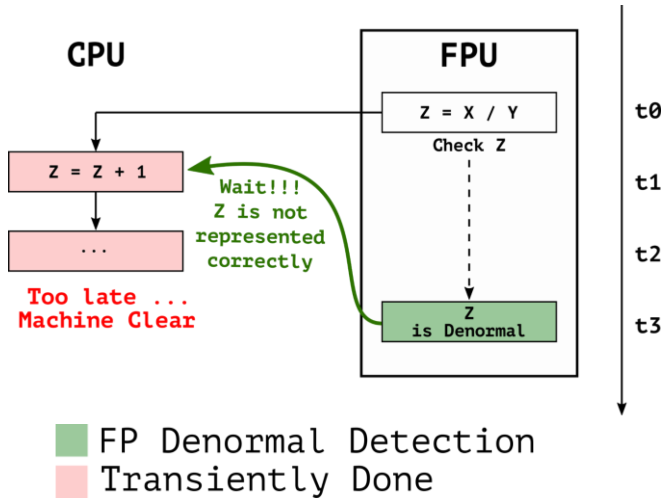
Subnormal FP operations

Security Implications

Transiently inject arbitrary FP values

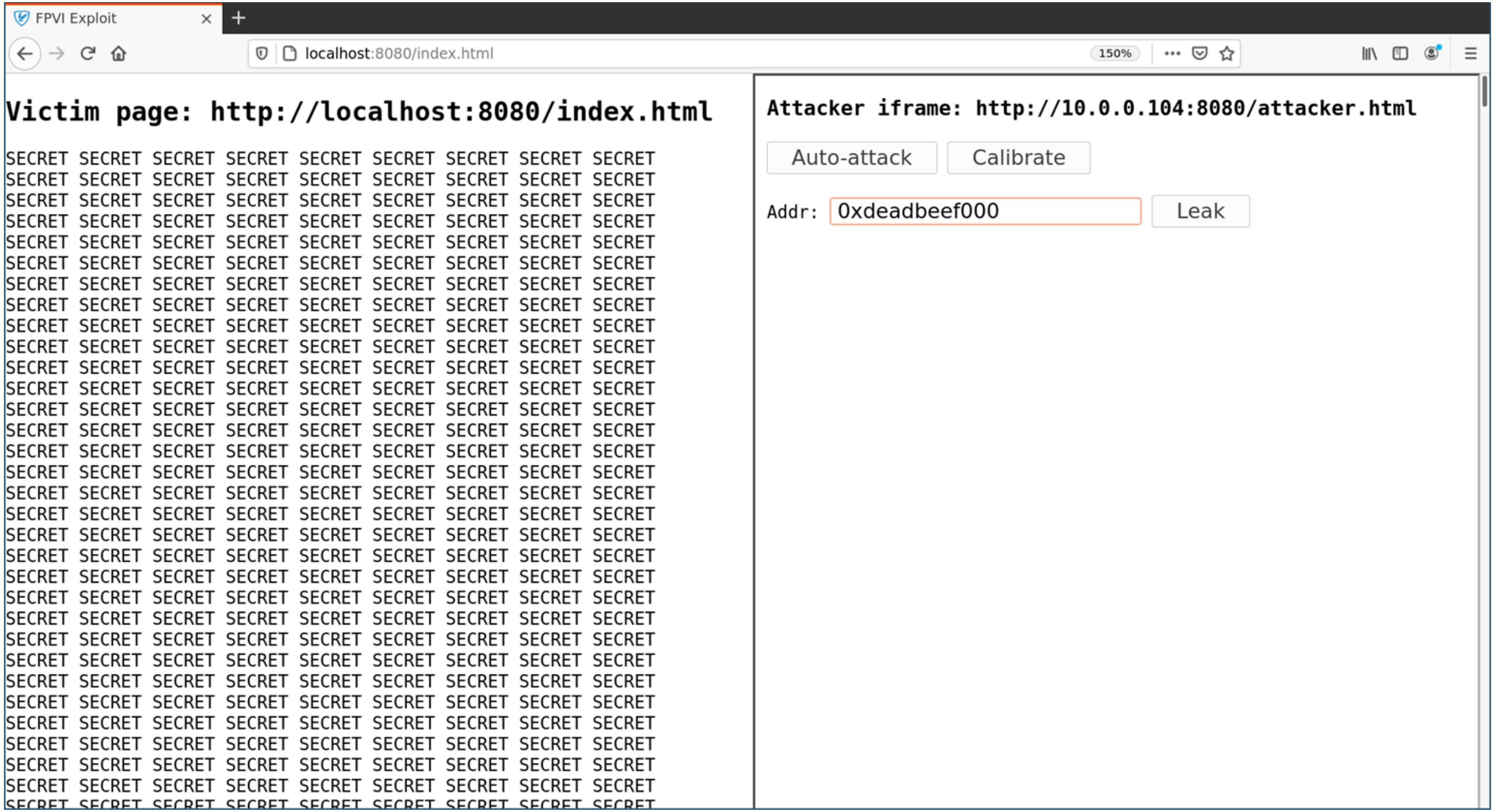
Exploitation

```
i1: Z = X / Y
i2: Z = Z + 1
i3: ...
```



FPVI EXPLOIT

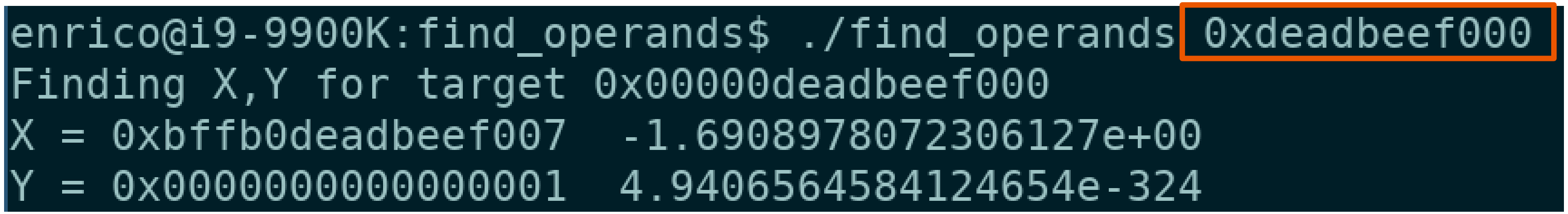
1. Attack Setup



```
Victim page: http://localhost:8080/index.html

SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET SECRET
```

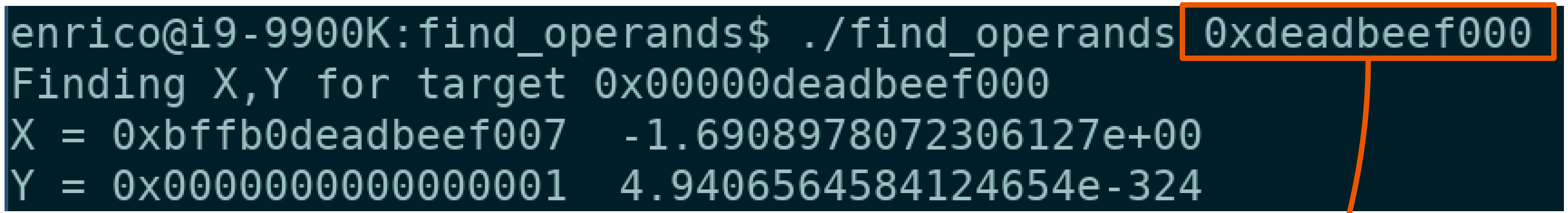
```
Attacker iframe: http://10.0.0.104:8080/attacker.html

Addr: 0xdeadbeef000
```

2. Finding Operands

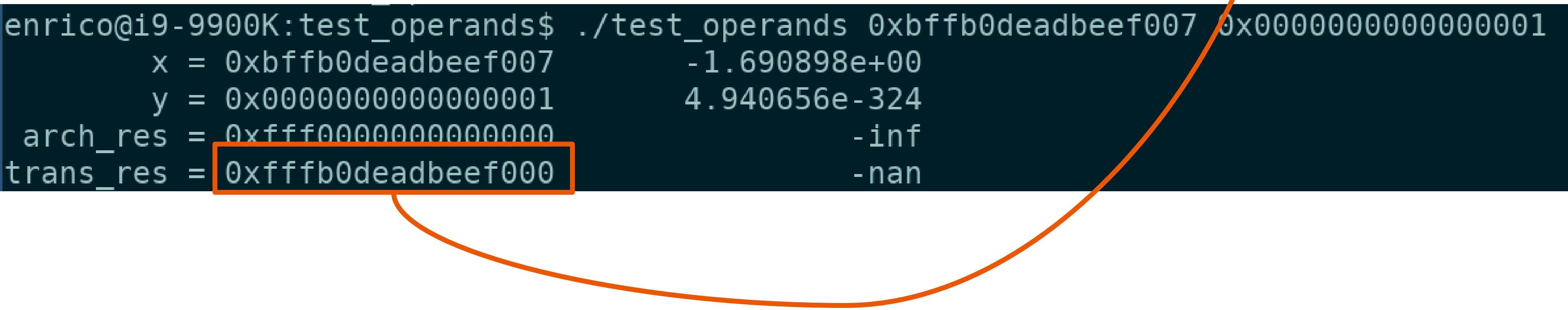


```
enrico@i9-9900K:find_operands$ ./find_operands 0xdeadbeef000
Finding X,Y for target 0x00000deadbeef000
X = 0xbffb0deadbeef007  -1.6908978072306127e+00
Y = 0x0000000000000001  4.9406564584124654e-324
```

2. Finding Operands

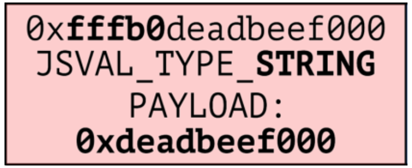


```
enrico@i9-9900K:find_operands$ ./find_operands 0xdeadbeef000
Finding X,Y for target 0x00000deadbeef000
X = 0xbffb0deadbeef007  -1.6908978072306127e+00
Y = 0x0000000000000001  4.9406564584124654e-324
```



```
enrico@i9-9900K:test_operands$ ./test_operands 0xbffb0deadbeef007 0x0000000000000001
        x = 0xbffb0deadbeef007       -1.690898e+00
        y = 0x0000000000000001       4.940656e-324
 arch_res = 0xfff0000000000000                -inf
trans_res = 0xfffb0deadbeef000                -nan
```

3. Memory Leak



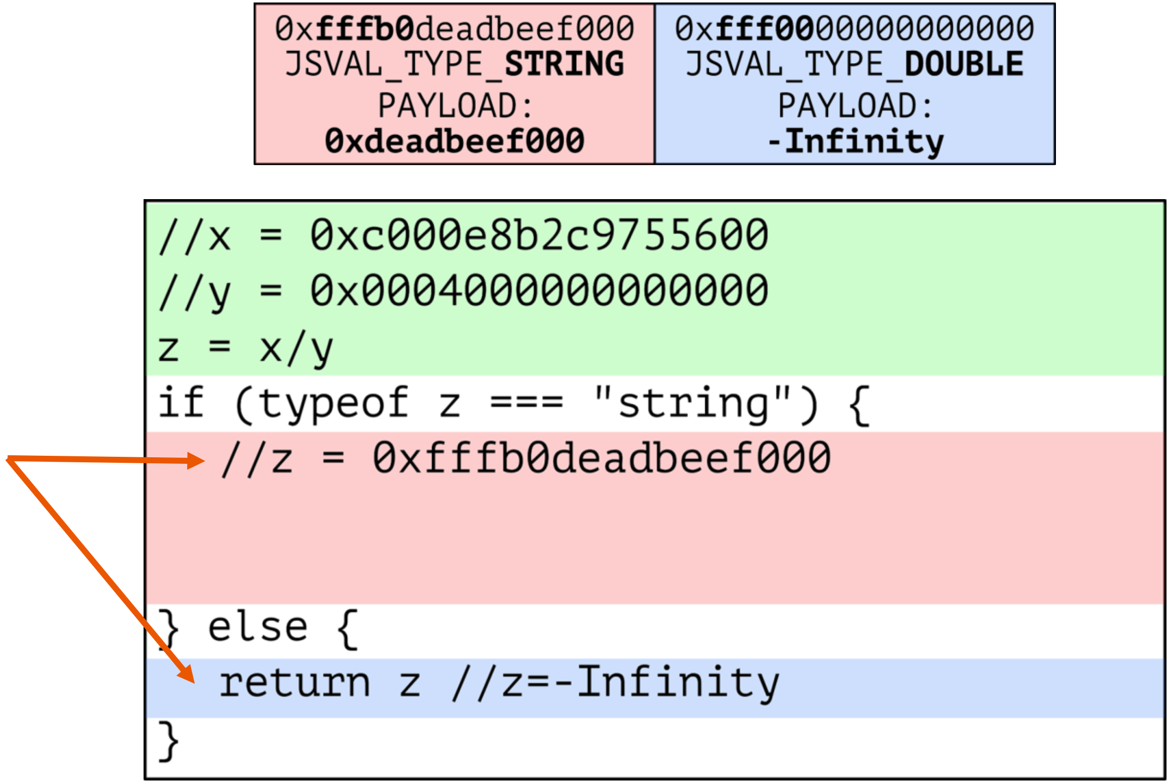
```
0xfffb0deadbeef000
JSVAL_TYPE_STRING
     PAYLOAD:
  0xdeadbeef000
```

3. Memory Leak

```
0xfffb0deadbeef000
JSVAL_TYPE_STRING
PAYLOAD:
0xdeadbeef000
```

```
//x = 0xc000e8b2c9755600
//y = 0x0004000000000000
z = x/y
if (typeof z === "string") {
```

3. Memory Leak



0x**fffb0**deadbeef000 JSVAL_TYPE_**STRING** PAYLOAD: **0xdeadbeef000**	0x**fff00**00000000000 JSVAL_TYPE_**DOUBLE** PAYLOAD: **-Infinity**

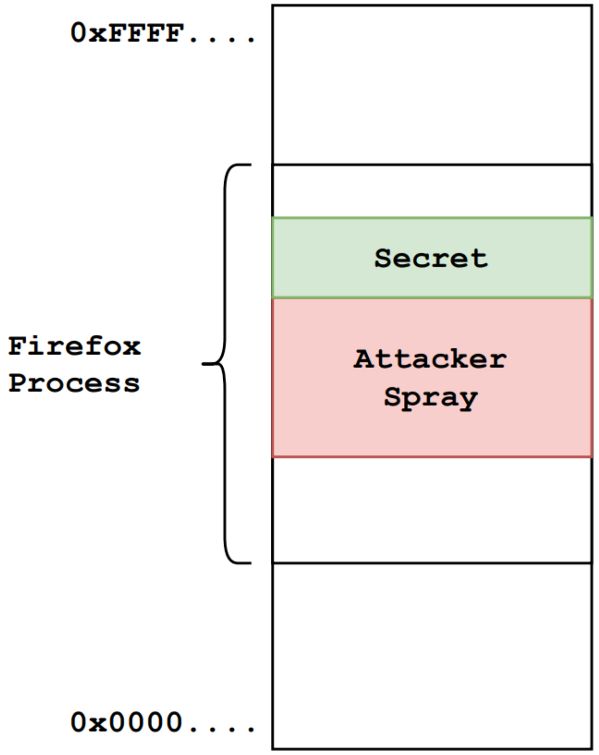
```
//x = 0xc000e8b2c9755600
//y = 0x0004000000000000
z = x/y
if (typeof z === "string") {
  //z = 0xfffb0deadbeef000



} else {
  return z //z=-Infinity
}
```

3. Memory Leak

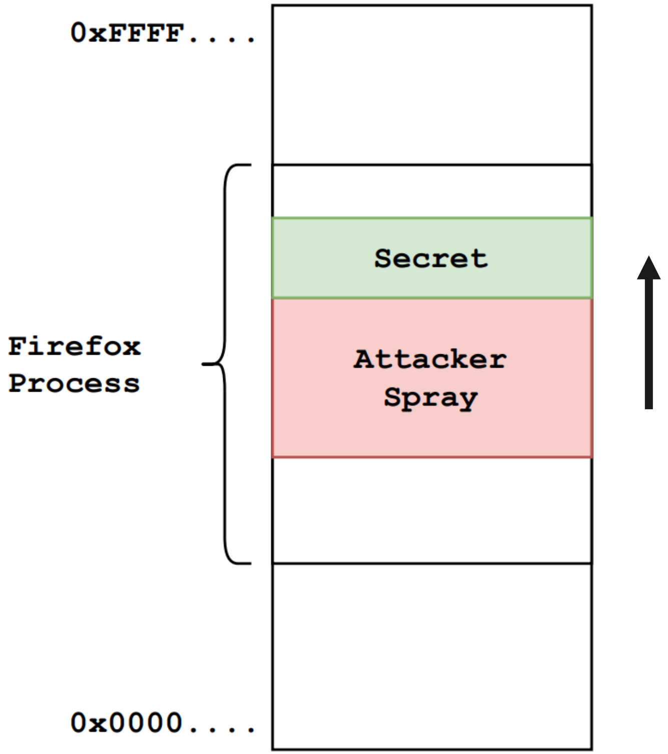
0x**fffb0**deadbeef000 JSVAL_TYPE_**STRING** PAYLOAD: **0xdeadbeef000**	0x**fff00**00000000000 JSVAL_TYPE_**DOUBLE** PAYLOAD: **-Infinity**

```
//x = 0xc000e8b2c9755600
//y = 0x0004000000000000
z = x/y
if (typeof z === "string") {
    //z = 0xfffb0deadbeef000
    //leak byte @ 0xdeadbeef004
    return buf[(z.length&0xff)<<10]
} else {
    return z //z=-Infinity
}
```

4. ASLR Bypass

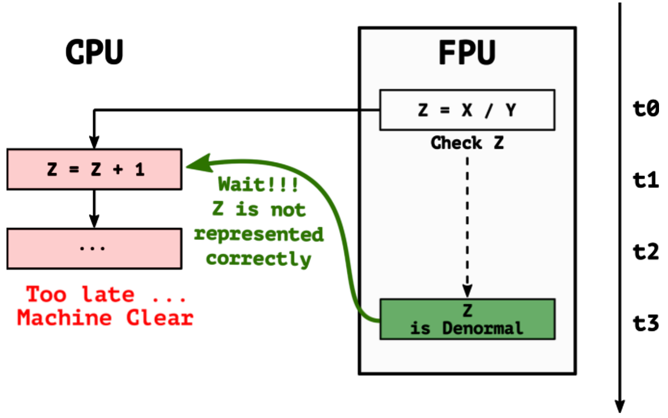


4. ASLR Bypass



Floating-Point Value Injection (FPVI)

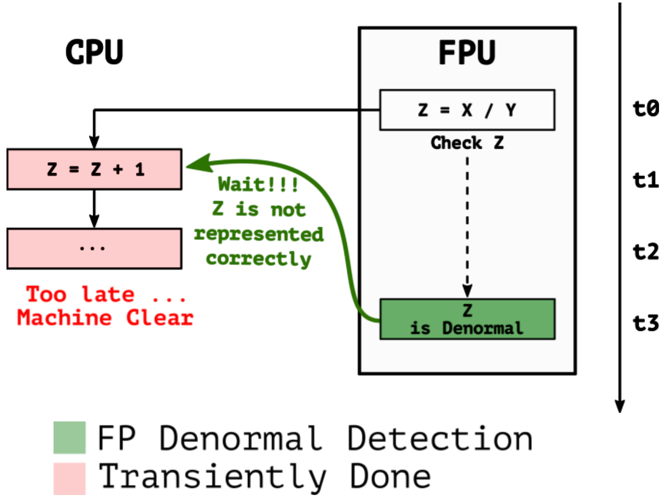
- Exploit leakage rate of 13 KB/s


CPU
FPU
Z = X / Y
Check Z
Z = Z + 1
...
Wait!!! Z is not represented correctly
Too late ... Machine Clear
Z is Denormal
t0
t1
t2
t3

🟩 FP Denormal Detection
🟥 Transiently Done

Floating-Point Value Injection (FPVI)

- **Exploit leakage rate of 13 KB/s**
- **Mitigations:**
  - ➔ **Flush To Zero (FTZ) & Denormal Are Zero (DAZ)**
  - ➔ We implemented a **LLVM pass** adding a serializing instruction in detected FPVI gadgets. With 53% geomean overhead for SPEC FP 2017.
  - ➔ Use **site-isolation** or **conditionally mask FP** operations in the browsers.



MEMORY DISAMBIGUATION MACHINE CLEAR

Memory Disambiguation Machine Clear

When a load instruction is following a store instruction which destination address is not ready yet, the Memory Disambiguation Unit predicts whether the two instructions are operating on the same memory addresses (i.e. Alias) or not (i.e. No-Alias).

Memory Disambiguation Machine Clear

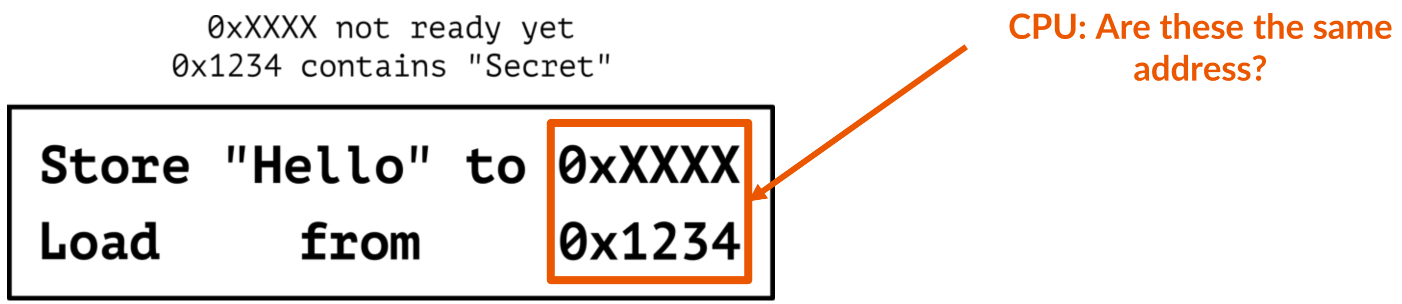
When a load instruction is following a store instruction which destination address is not ready yet, the Memory Disambiguation Unit predicts whether the two instructions are operating on the same memory addresses (i.e. Alias) or not (i.e. No-Alias).

```
0xXXXX not ready yet
0x1234 contains "Secret"
```

```
Store "Hello" to 0xXXXX
Load      from   0x1234
```

Memory Disambiguation Machine Clear

When a load instruction is following a store instruction which destination address is not ready yet, the Memory Disambiguation Unit predicts whether the two instructions are operating on the same memory addresses (i.e. Alias) or not (i.e. No-Alias).

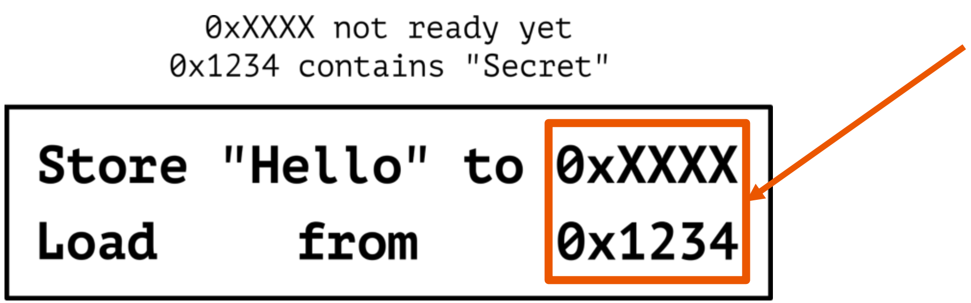


```
0xXXXX not ready yet
0x1234 contains "Secret"
```

```
Store "Hello" to 0xXXXX
Load       from  0x1234
```

Memory Disambiguation Machine Clear

When a load instruction is following a store instruction which destination address is not ready yet, the Memory Disambiguation Unit predicts whether the two instructions are operating on the same memory addresses (i.e. Alias) or not (i.e. No-Alias).



```
0xXXXX not ready yet
0x1234 contains "Secret"
```

```
Store "Hello" to 0xXXXX
Load      from   0x1234
```

CPU: Are these the same address?

MDU: I predict they're not the same (i.e. No-Alias)

Memory Disambiguation Machine Clear

When a load instruction is following a store instruction which destination address is not ready yet, the Memory Disambiguation Unit predicts whether the two instructions are operating on the same memory addresses (i.e. Alias) or not (i.e. No-Alias).

```
0xXXXX not ready yet
0x1234 contains "Secret"
```

```
Store "Hello" to 0xXXXX
Load       from  0x1234
```

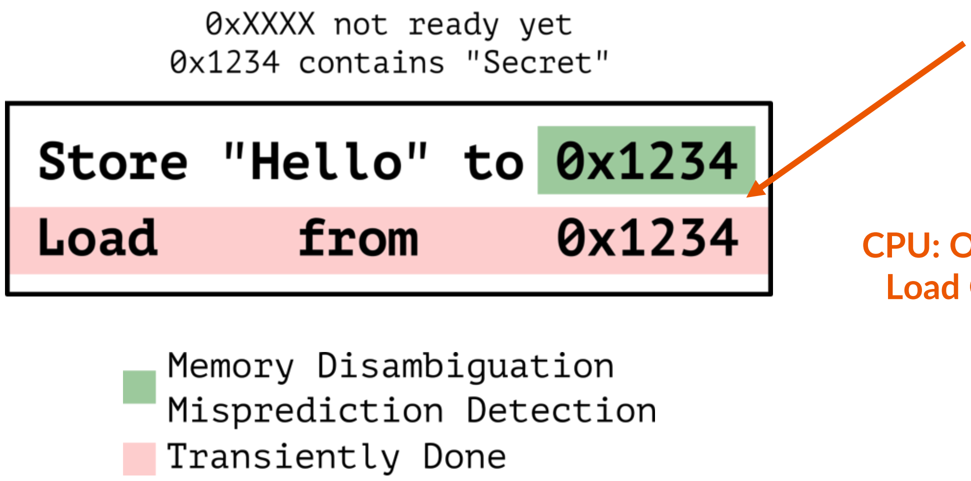
CPU: Are these the same address?

MDU: I predict they're not the same (i.e. No-Alias)

CPU: OK, I will not wait to resolve 0xXXXX, Load Out-of-Order 0x1234 (i.e. "Secret")

Memory Disambiguation Machine Clear

When a load instruction is following a store instruction which destination address is not ready yet, the Memory Disambiguation Unit predicts whether the two instructions are operating on the same memory addresses (i.e. Alias) or not (i.e. No-Alias).



```
0xXXXX not ready yet
0x1234 contains "Secret"

Store "Hello" to 0x1234
Load        from 0x1234
```

🟩 Memory Disambiguation Misprediction Detection

🟥 Transiently Done

CPU: Are these the same address?

MDU: I predict they're not the same (i.e. No-Alias)

CPU: OK, I will not wait to resolve 0xXXXX, Load Out-of-Order 0x1234 (i.e. "Secret")

Memory Disambiguation Machine Clear

Architectural Invariant

Stores followed by Loads are always disambiguated correctly

Invariant Violation

MDU misprediction

Security Implications

Transiently leak stale data

Exploitation

Spectre v4 (Speculative Store Bypass)

Other types of Machine Clear

- AVX vmaskmov
- Exceptions
- Hardware interrupts
- Microcode assists

RESULTS

Let’s zoom out a bit ...

Let's zoom out a bit ...

Self-Modifying Code

```
i1: ...
i2: store nop @ i3
i3: load secret
```

🟩 Machine Clear Detection
🟥 Transiently Done

Let’s zoom out a bit ...

Self-Modifying Code

```
i1: ...
i2: store nop @ i3
i3: load secret
```

🟩 Machine Clear Detection
🟥 Transiently Done

```
 PROCESSOR A        PROCESSOR B
r1 = [X] (slow)    [X] = 1
r2 = [Y] (fast)    [Y] = 1
r3 = f(r2)
```

Memory Ordering

Let's zoom out a bit ...

🟩 Machine Clear Detection
🟥 Transiently Done

Self-Modifying Code

```
i1: ...
i2: store nop @ i3
i3: load secret
```

Floating-Point

```
i1: Z = X / Y
i2: Z = Z + 1
i3: ...
```

```
PROCESSOR A          PROCESSOR B
r1 = [X] (slow)      [X] = 1
r2 = [Y] (fast)      [Y] = 1
r3 = f(r2)
```

Memory Ordering

Let's zoom out a bit ...

🟩 Machine Clear Detection
🟥 Transiently Done

Self-Modifying Code

```
i1: ...
i2: store nop @ i3
i3: load secret
```

Floating-Point

```
i1: Z = X / Y
i2: Z = Z + 1
i3: ...
```

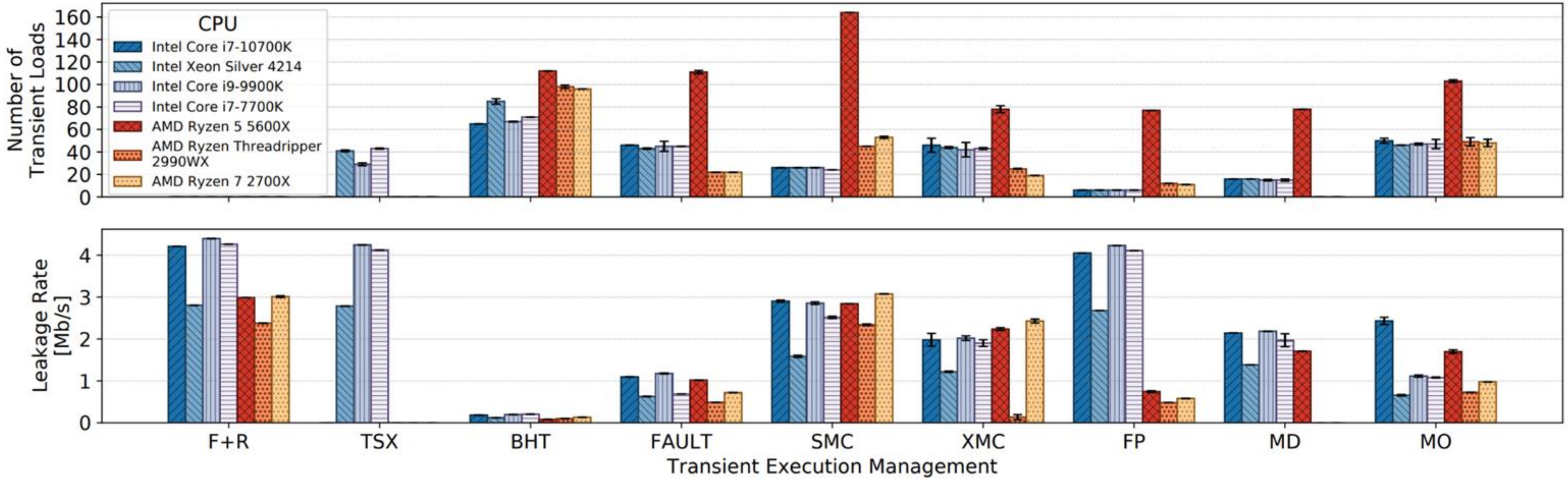
Memory Ordering

```
PROCESSOR A            PROCESSOR B
r1 = [X] (slow)        [X] = 1
r2 = [Y] (fast)        [Y] = 1
r3 = f(r2)
```

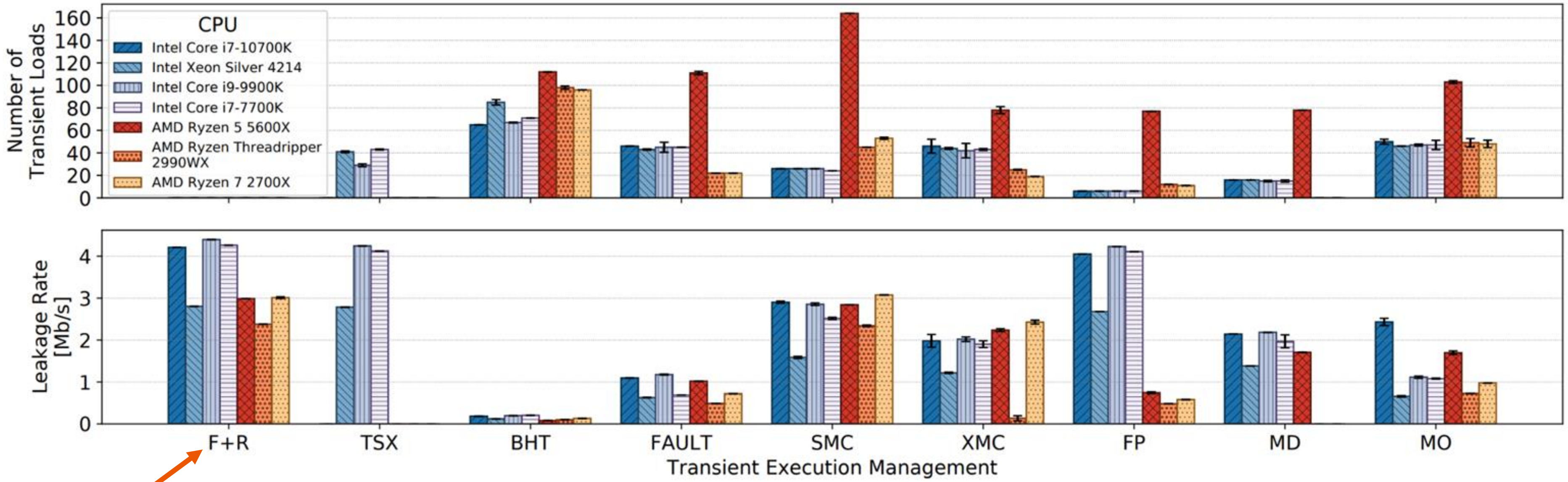
Memory Disambiguation

```
i1: store "Hello" to 0xXXXX
i2: load      from      0x1234
```

Transient Execution Capabilities

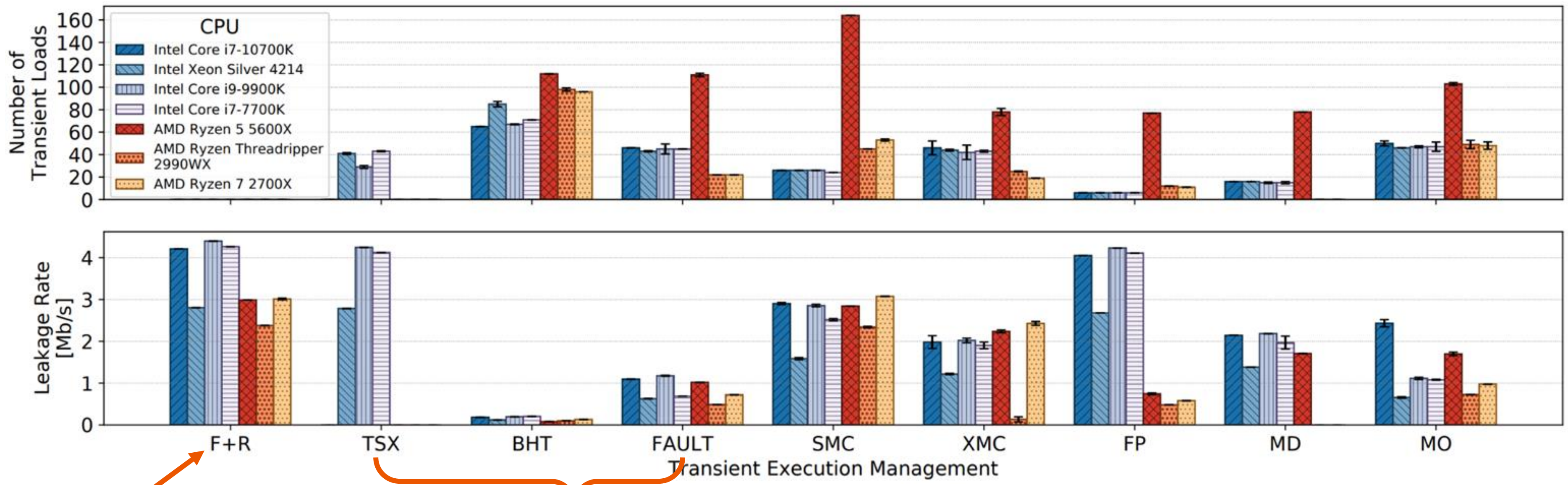


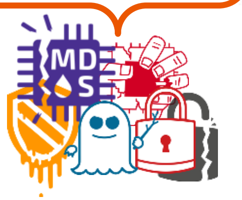
Transient Execution Capabilities



Architectural upper limit leakage rate

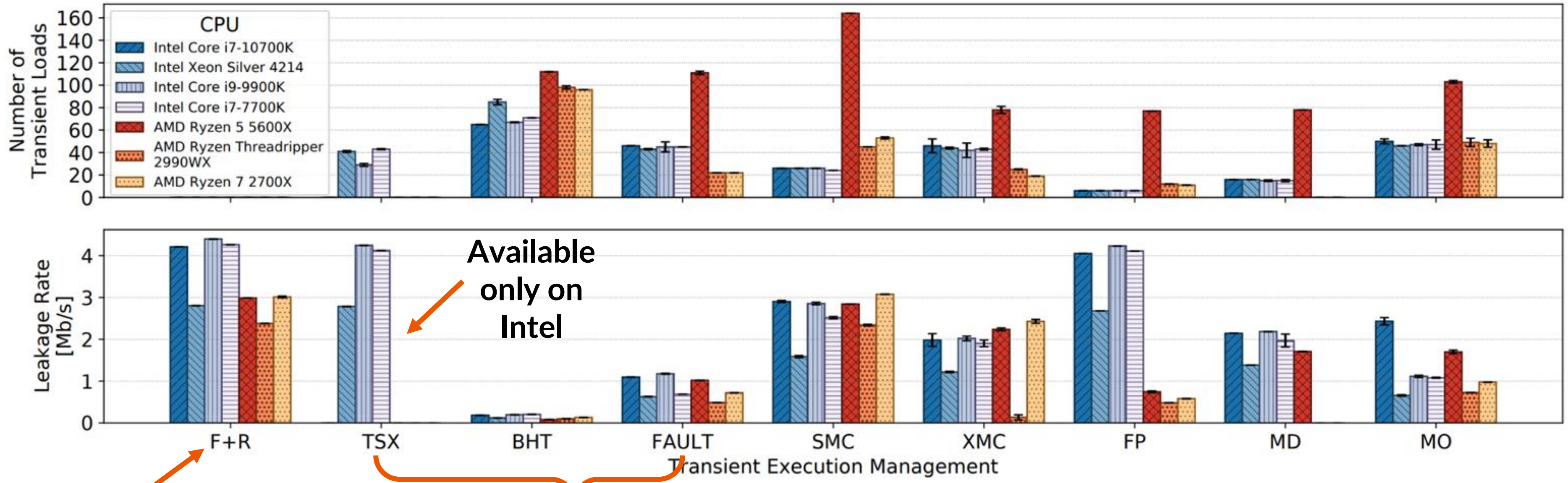
Transient Execution Capabilities





Architectural upper limit leakage rate

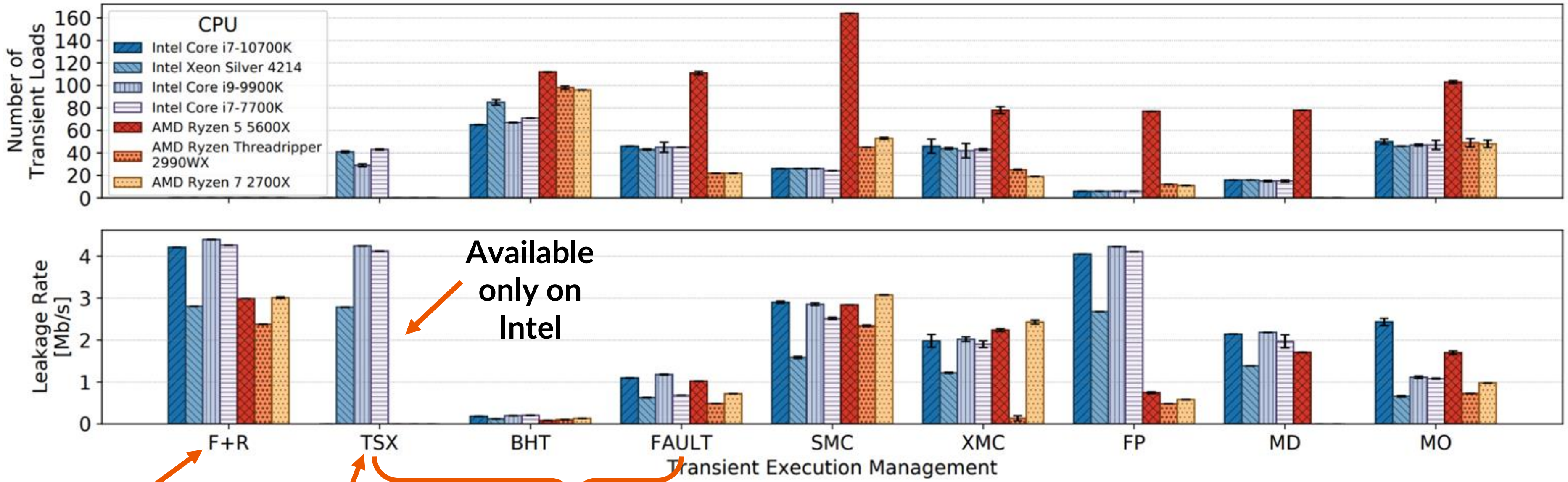
Transient Execution Capabilities



Available only on Intel

Architectural upper limit leakage rate

Transient Execution Capabilities

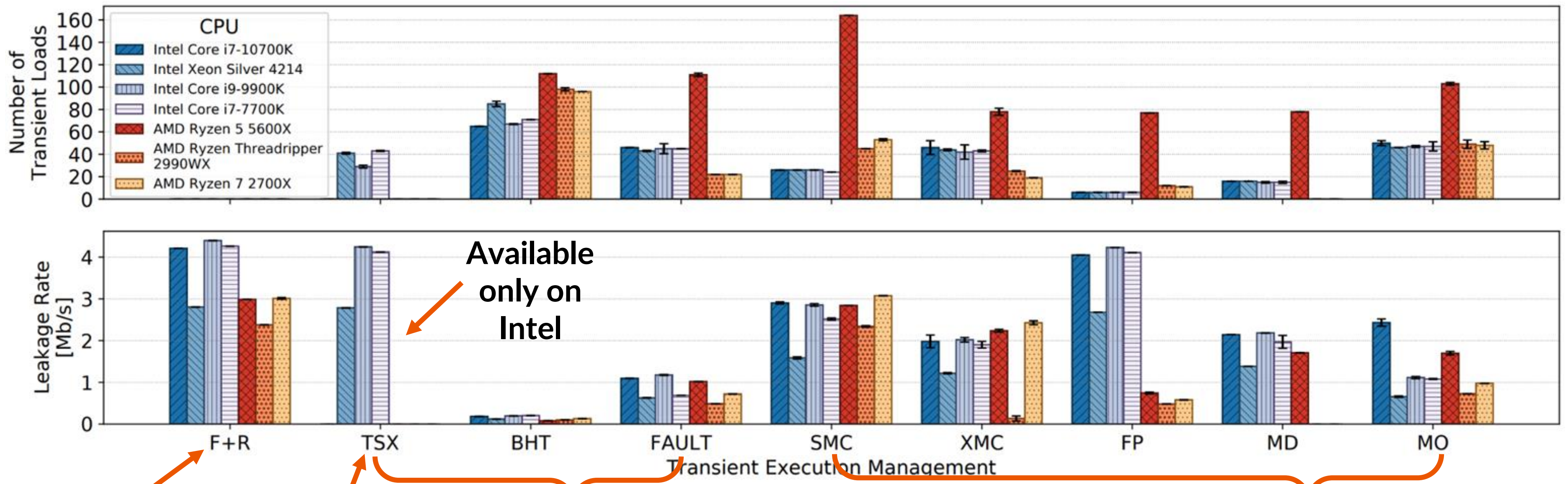


Available only on Intel

Architectural upper limit leakage rate

Not supported anymore on recent CPUs

Transient Execution Capabilities



Available only on Intel

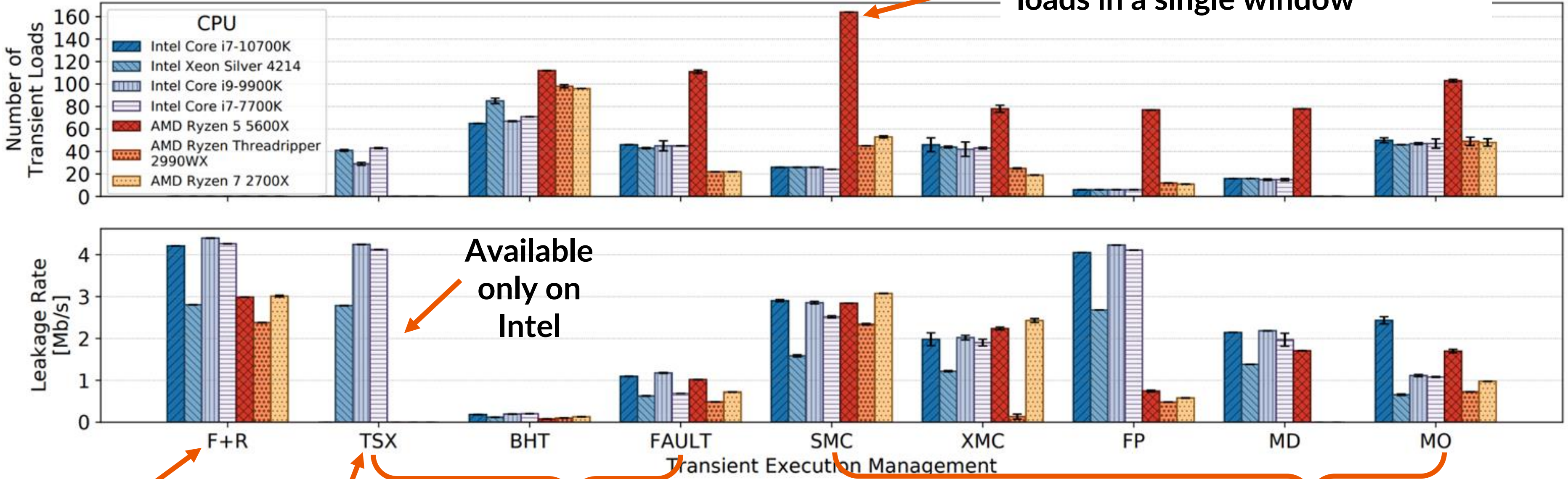
Architectural upper limit leakage rate

Not supported anymore on recent CPUs

Available also on AMD

Transient Execution Capabilities

SMC can reach > 160 transient loads in a single window



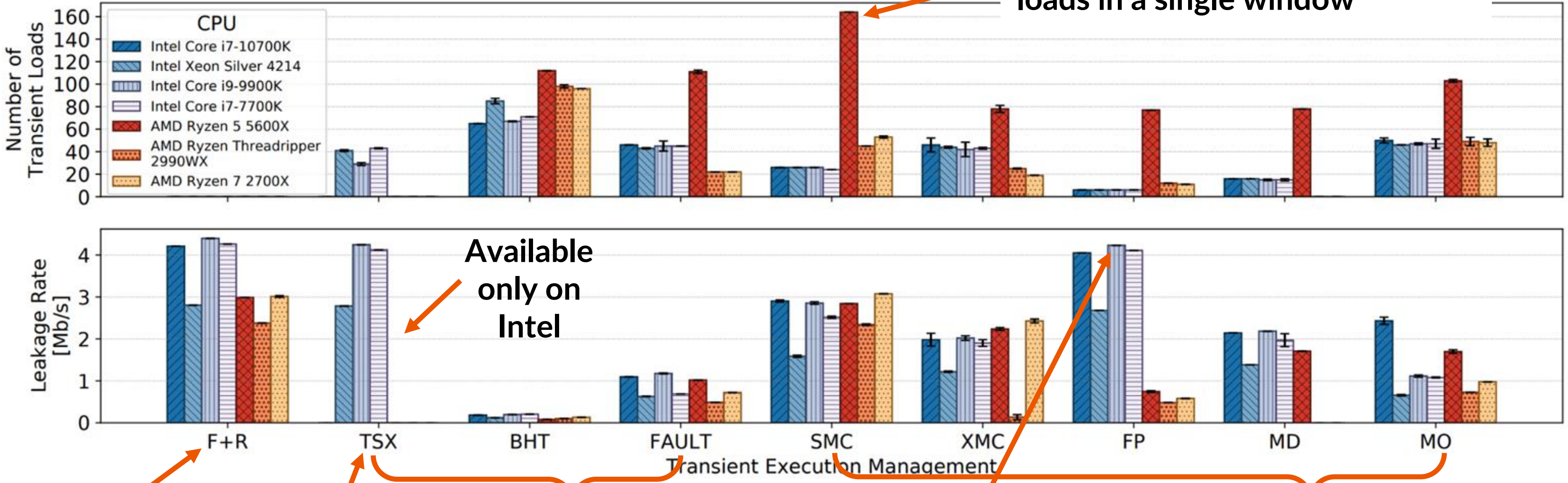
Available only on Intel

Architectural upper limit leakage rate

Not supported anymore on recent CPUs

Available also on AMD

Transient Execution Capabilities

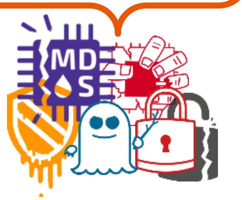


SMC can reach > 160 transient loads in a single window

Available only on Intel

Architectural upper limit leakage rate

Not supported anymore on recent CPUs

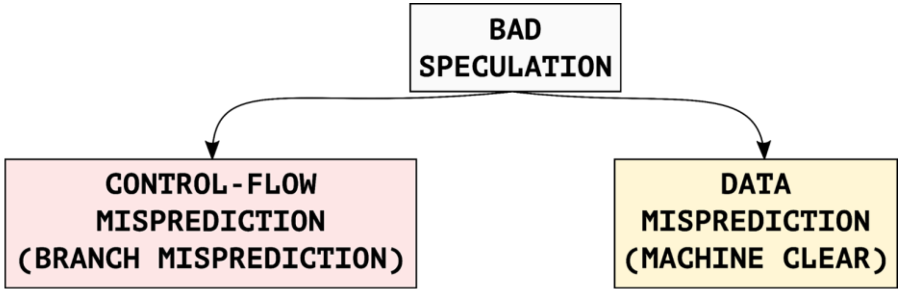


FP has the best leakage rates (>4Mb/s) thanks to its determinism (i.e. No mistraining needed)

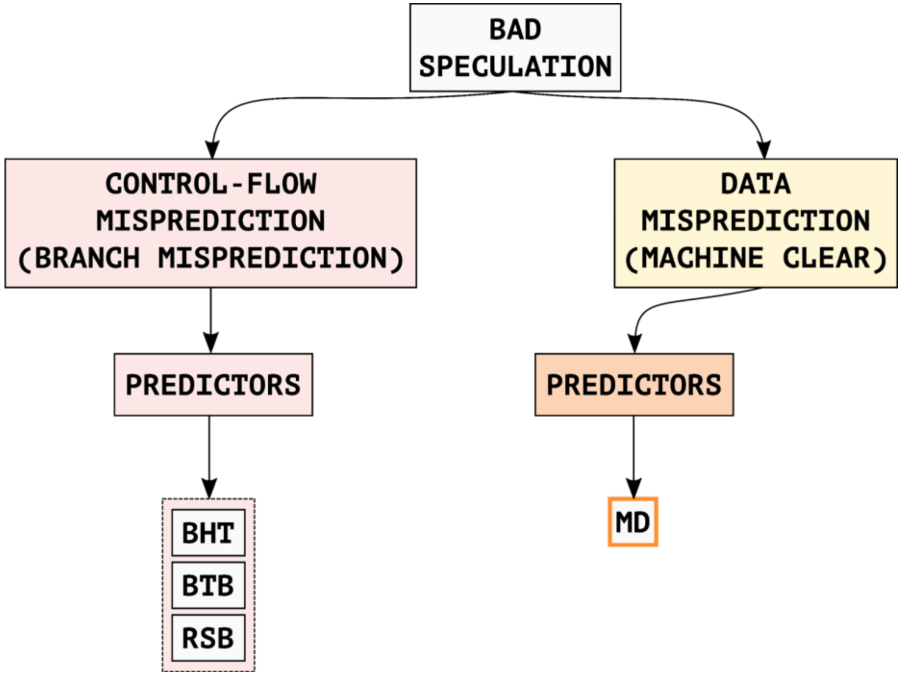
Available also on AMD

Root-Cause Classification of Transient Execution

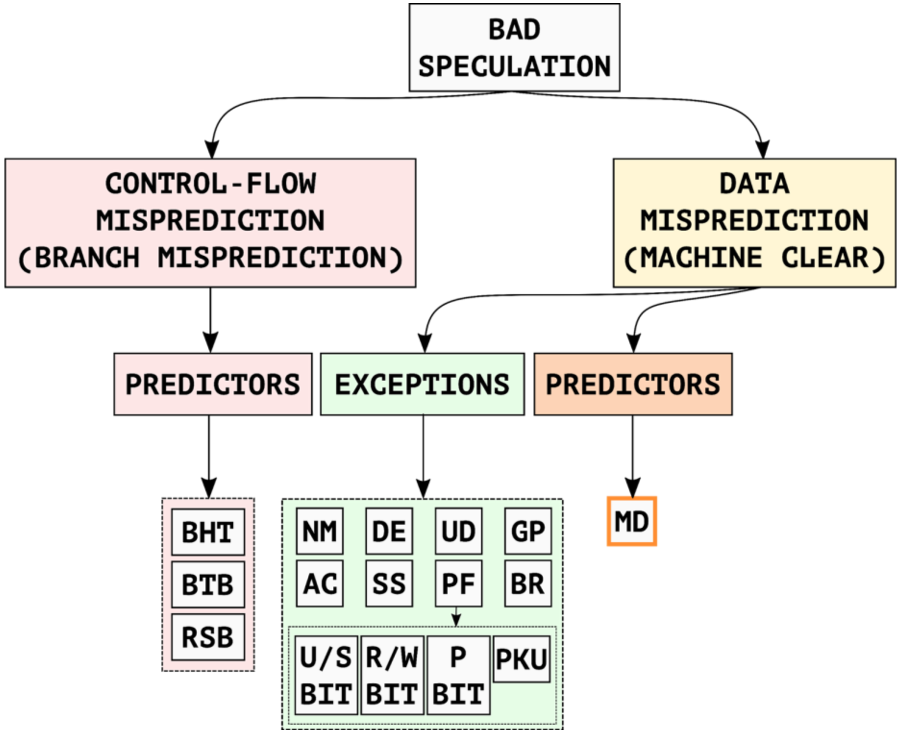
Root-Cause Classification of Transient Execution


BAD SPECULATION
CONTROL-FLOW MISPREDICTION (BRANCH MISPREDICTION)
DATA MISPREDICTION (MACHINE CLEAR)

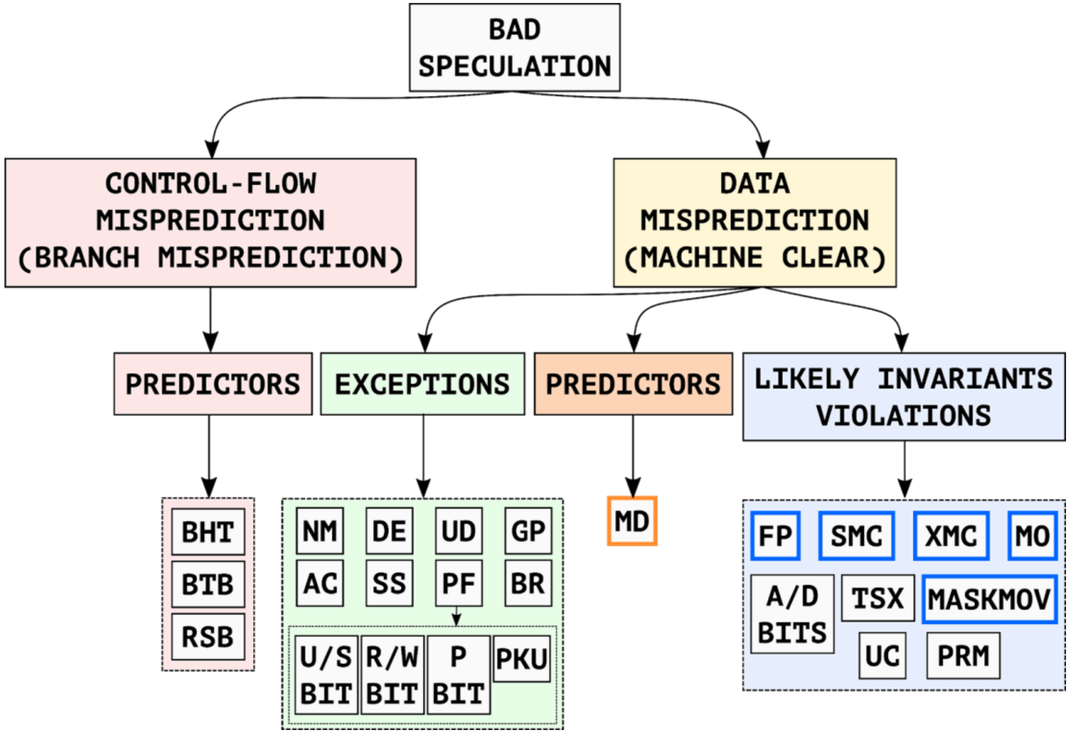
Root-Cause Classification of Transient Execution


BAD SPECULATION
CONTROL-FLOW MISPREDICTION (BRANCH MISPREDICTION)
DATA MISPREDICTION (MACHINE CLEAR)
PREDICTORS
PREDICTORS
BHT
BTB
RSB
MD

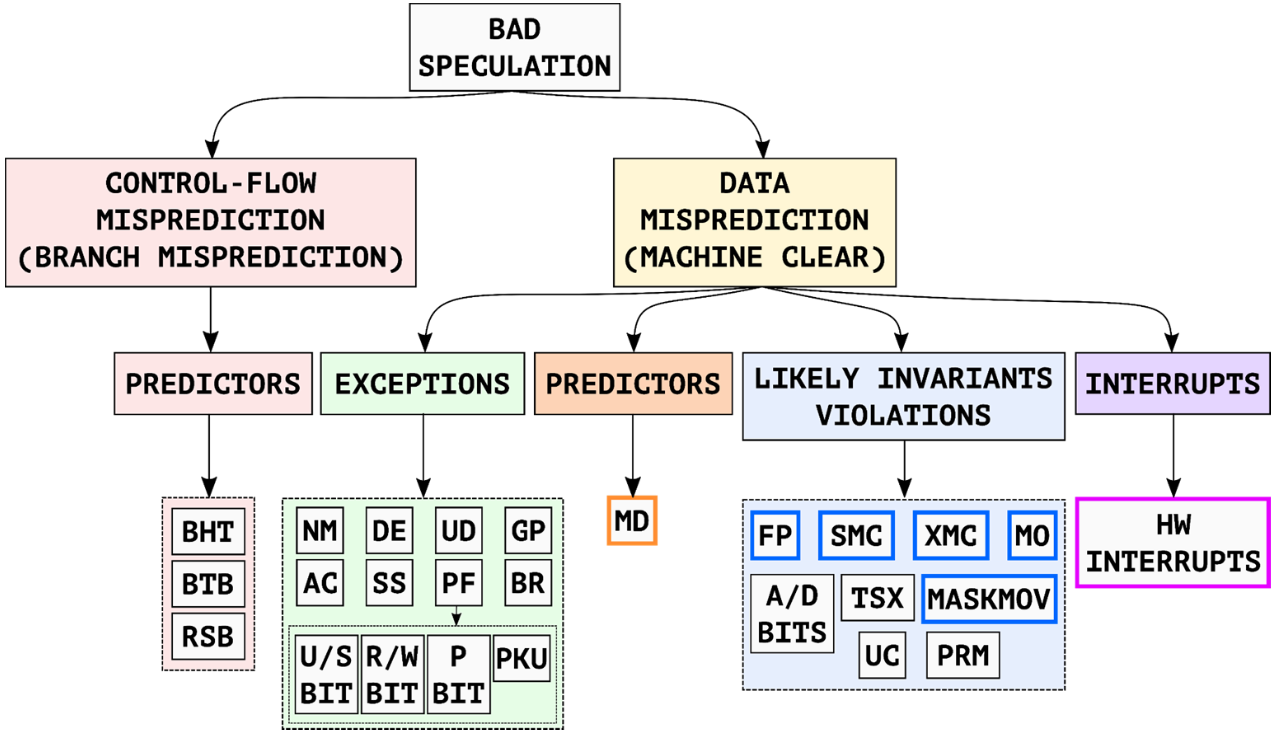
Root-Cause Classification of Transient Execution



Root-Cause Classification of Transient Execution



Root-Cause Classification of Transient Execution



Disclosure & Affected CPUs

- We disclosed FPVI and SCSB to CPU, browser, OS, and hypervisor vendors in February 2021.

Disclosure & Affected CPUs

- We disclosed FPVI and SCSB to CPU, browser, OS, and hypervisor vendors in February 2021.

CPU Vendor	**Affected by SCSB** (CVE-2021-0089) (CVE-2021-26313)	**Affected by FPVI** (CVE-2021-0086) (CVE-2021-26314)
Intel	✓	✓
AMD	✓	✓ *
ARM	✗	✓ **

* No exploitable NaN-boxed transient results were found
** ARM reported that some FPU implementations are affected by FPVI

Disclosure & Affected CPUs

- We disclosed FPVI and SCSB to CPU, browser, OS, and hypervisor vendors in February 2021.

- Mozilla confirmed the FPVI vulnerability (CVE-2021-29955) and deployed a mitigation based on conditionally masking malicious NaN-boxed FP results in Firefox 87.

CPU Vendor	**Affected by SCSB** (CVE-2021-0089) (CVE-2021-26313)	**Affected by FPVI** (CVE-2021-0086) (CVE-2021-26314)
Intel	✓	✓
AMD	✓	✓ *
ARM	✗	✓ **

* No exploitable NaN-boxed transient results were found
** ARM reported that some FPU implementations are affected by FPVI

Disclosure & Affected CPUs

- We disclosed FPVI and SCSB to CPU, browser, OS, and hypervisor vendors in February 2021.
- Mozilla confirmed the FPVI vulnerability (CVE-2021-29955) and deployed a mitigation based on conditionally masking malicious NaN-boxed FP results in Firefox 87.
- Xen hypervisor mitigated SCSB and released a security advisory (XSA-375) following our proposed mitigation.

CPU Vendor	**Affected by SCSB** (CVE-2021-0089) (CVE-2021-26313)	**Affected by FPVI** (CVE-2021-0086) (CVE-2021-26314)
Intel	✓	✓
AMD	✓	✓ *
ARM	✗	✓ **

* No exploitable NaN-boxed transient results were found
** ARM reported that some FPU implementations are affected by FPVI

Rage Against The Machine Clear

- Bad Speculation is not caused only by classic mispredictions

Rage Against The Machine Clear

- Bad Speculation is not caused only by classic mispredictions , but also by architectural invariants violations, i.e. Machine Clear.

Rage Against The Machine Clear

- Bad Speculation is not caused only by classic mispredictions, but also by architectural invariants violations, i.e. Machine Clear.

- Architectural invariants can be exploited, creating new security threats, e.g. FPVI & SCSB

Rage Against The Machine Clear

- Bad Speculation is not caused only by classic mispredictions, but also by architectural invariants violations, i.e. Machine Clear.

- Architectural invariants can be exploited, creating new security threats, e.g. FPVI & SCSB

- Defenses must focus on the wider class of root-causes of bad speculation.

Rage Against The Machine Clear

- Bad Speculation is not caused only by classic mispredictions , but also by architectural invariants violations, i.e. Machine Clear.

- Architectural invariants can be exploited, creating new security threats, e.g. FPVI & SCSB

- Defenses must focus on the wider class of root-causes of bad speculation.

VUSec

@hanyrax @enrico_barberis

https://www.vusec.net/projects/fpvi-scsb/

https://github.com/vusec/fpvi-scsb

http://download.vusec.net/papers/fpvi-scsb_sec21.pdf