Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks

Hany Ragab*, Enrico Barberis*, Herbert Bos and Cristiano Giuffrida

*Equal contribution joint first authors

VUSec

Vrije Universiteit Amsterdam
Speculative Execution

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

![Data cache diagram]

- ...  
- `array[x-1]`  
- `array[x]`  
- `array[x+1]`  
- ...  

- Not cached
if (x < \texttt{array	extunderscore size} ) {
    y = \texttt{array}[x]
}

Data cache

\begin{array}{cccc}
\cdots & \texttt{array}[x-1] & \texttt{array}[x] & \texttt{array}[x+1] & \cdots \\
\end{array}

\quad \square \text{ Not cached}
if (x < \texttt{array\_size}) {
    y = \texttt{array}[x]
}
Speculative Execution

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

Data cache

```
... array[x-1] array[x] array[x+1] ...
```

- Cached
- Not cached
if (x < array_size) {
    y = array[x]
}

Speculative Execution

Data cache

...  array[x-1]  array[x]  array[x+1]  ...

Cached
Not cached
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 & Intel TSX  |  Machine Clear
Bad Speculation

The root cause of discarding issued µOps on x86 processors

Branch Misprediction & Intel TSX

Machine Clear
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
Security Analysis of Machine Clear

1. Architectural Invariant
Security Analysis of Machine Clear

1. Architectural Invariant

2. Invariant Violation
Security Analysis of Machine Clear

1. Architectural Invariant
2. Invariant Violation
3. Security Implications
Security Analysis of Machine Clear

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

IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
IF ID EX MEM WB
Self-Modifying Code is a program storing instructions as data, modifying its own code as it is being executed.

1. \(i_1: \ldots\)
2. \(i_2: \text{store \ nop @ i3}\)
3. \(i_3: \text{load secret}\)
4. \(i_4: \ldots\)
5. \(i_5: \ldots\)
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: ...

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

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

Diagram:
- IF ID EX MEM WB
- IF ID EX MEM WB
- IF ID EX MEM WB
- IF ID EX MEM WB
- IF ID EX MEM WB

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

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

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

Security Implications
Transiently execute stale code

<table>
<thead>
<tr>
<th>i1: ...</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>i2: store nop @ i3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>i3: load secret</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>i4: ...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>i5: ...</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Wait!!! You're a nop now
Too late ... Machine Clear

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

Security Implications
Transiently execute stale code

Exploitation
?

### Diagram

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

#### Execution Stages

- **IF**: Instruction Fetch
- **ID**: Instruction Decode
- **EX**: Execute
- **MEM**: Memory Access
- **WB**: Write Back

- **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)
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)

8.1.3 Handling Self- and Cross-Modifying Code

Architectural Invariant
Stores always target data memory

Invariant Violation
Self-Modifying Code

Security Implications
Transiently execute stale 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

Architectural Invariant
Stores always target data memory

Invariant Violation
Self-Modifying Code

Security Implications
Transiently execute stale code

Exploitation
Speculative Code Store Bypass
Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. $2^{-1022}$)
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: ...
Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. $2^{-1022}$)
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

i1: $Z = X / Y$
i2: $Z = Z + 1$
i3: ...
Subnormal/Denormal numbers are a special range of floating-point numbers with a value smaller than the smallest Normal number (i.e. $2^{-1022}$).
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

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

FP Denormal Detection
Transiently Done
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: ...

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

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

Security Implications
 transiently inject arbitrary FP values
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: ...

Security Implications

Transiently inject arbitrary FP values

Exploitation

Diagram:

- CPU: $Z = X / Y$
- FPU: $Z = Z + 1$, $Z$ is not represented correctly,
  - Wait!!!
  - Too late ...
  - Machine Clear
- FP Denormal Detection
- Transiently Done
Floating-Point Value Injection (FPVI)

0xffffdeadbeef000
JSVAL_TYPE_STRING
PAYLOAD:
0xdeadbeef000
Floating-Point Value Injection (FPVI)

```cpp
// x = 0xc000e8b2c9755600
// y = 0x0004000000000000
z = x/y
if (typeof z === "string") {
}
```
Floating-Point Value Injection (FPVI)

```
//x = 0xc000e8b2c9755600
//y = 0x0004000000000000
z = x/y
if (typeof z === "string") {
    //z = 0xffffb0deadbeef000
} else {
    return z //z=-Infinity
}
```
Floating-Point Value Injection (FPVI)

```javascript
// x = 0xc000e8b2c9755600
// y = 0x0004000000000000
z = x/y
if (typeof z === "string") {
    // z = 0xffffb0deadbeef000
    // leak byte @ 0xdeadbeef004
    return buf[(z.length&0xff)<<10]
} else {
    return z // z=-Infinity
}
```
Floating-Point Value Injection (FPVI)

Architectural Invariant
FPU always operates on normal numbers

Invariant Violation
Denormal FP operations

Security Implications
Transiently inject arbitrary FP values

Exploitation
Floating-Point Value Injection
Floating-Point Value Injection (FPVI)

- Exploit leakage rate of 13 KB/s
Floating-Point Value Injection (FPVI)

- Exploit leakage rate of 13 KB/s
- Mitigations:
  - Flush To Zero (FTZ) & Denormal Are Zero (DAZ)
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.
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.
Transient Execution Capabilities

![Graph showing transient execution capabilities for different CPU models.](image-url)
Transient Execution Capabilities

Architectural baseline leakage rate
Transient Execution Capabilities

Architectural baseline leakage rate
Transient Execution Capabilities

Available only on Intel

Architectural baseline leakage rate
Transient Execution Capabilities

- Not supported anymore on recent CPUs
- Available only on Intel
- Architectural baseline leakage rate
- Not supported anymore on recent CPUs
Transient Execution Capabilities

- Not supported anymore on recent CPUs
- Available only on Intel
- Available also on AMD

Architectural baseline leakage rate

Not supported anymore on recent CPUs

Transient Execution Management
Transient Execution Capabilities

SMC can reach > 160 transient loads in a single window

Available only on Intel

Available also on AMD

Architectural baseline leakage rate

Not supported anymore on recent CPUs
Transient Execution Capabilities

SMC can reach > 160 transient loads in a single window

Available only on Intel

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

Available also on AMD

Architectural baseline leakage rate

Not supported anymore on recent CPUs

Transient Execution Management
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
Root-Cause Classification of Transient Execution
Root-Cause Classification of Transient Execution
Root-Cause Classification of Transient Execution

BAD SPECULATION

CONTROL-FLOW MISPREDICTION (BRANCH MISPREDICTION)
- PREDICTORS
  - BHT
  - BTB
  - RSB
- Exceptions
  - NM
  - DE
  - UD
  - GP
  - BR
  - U/S
  - R/W
  - P
  - PKU

DATA MISPREDICTION (MACHINE CLEAR)
- PREDICTORS
  - MD
- Likely Invariants Violations
  - FP
  - SMC
  - XMC
  - MO
  - A/D
  - TSX
  - MASKMOV
  - UC
  - PRM

INTERRUPTS
  - HW INTERRUPTS
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.

<table>
<thead>
<tr>
<th>CPU Vendor</th>
<th>Affected by SCSB (CVE-2021-0089) (CVE-2021-26313)</th>
<th>Affected by FPVI (CVE-2021-0086) (CVE-2021-26314)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>AMD</td>
<td>✔️</td>
<td>✔️*</td>
</tr>
<tr>
<td>ARM</td>
<td>❌</td>
<td>✔️**</td>
</tr>
</tbody>
</table>

* 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.

<table>
<thead>
<tr>
<th>CPU Vendor</th>
<th>Affected by SCSB (CVE-2021-0089) (CVE-2021-26313)</th>
<th>Affected by FPVI (CVE-2021-0086) (CVE-2021-26314)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>AMD</td>
<td>✔</td>
<td>✔*</td>
</tr>
<tr>
<td>ARM</td>
<td>✘</td>
<td>✔**</td>
</tr>
</tbody>
</table>

* 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.

<table>
<thead>
<tr>
<th>CPU Vendor</th>
<th>Affected by SCSB (CVE-2021-0089) (CVE-2021-26313)</th>
<th>Affected by FPVI (CVE-2021-0086) (CVE-2021-26314)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>AMD</td>
<td>✔</td>
<td>✔*</td>
</tr>
<tr>
<td>ARM</td>
<td>✘</td>
<td>✔**</td>
</tr>
</tbody>
</table>

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