Jamais Vu: Thwarting Microarchitectural Replay Attacks

Dimitrios Skarlatos†, Zirui Neil Zhao†, Riccardo Paccagnella, Christopher Fletcher, Josep Torrellas

University of Illinois Carnegie Mellon University

† Authors contributed equally to this work

ASPLOS’21
The Era of Side-Channels
The Era of Side-Channels
Port Contention Attack*

Victim (in SGX):
```c
if (secret) {
    // use _shared_ resource
} else {
    // don’t use _shared_ resource
}
```

Attacker (controls OS):
```c
while (true) {
    start = time();
    // use _shared_ resource
    latency = time() - start;
}
```

Attacker can infer the secret based on the measured latency:
- If latency > threshold: secret = 1;
- If latency <= threshold: secret = 0;

However, this side-channel is noisy, attacker needs repeated victim execution to be confident

How to force victim to repeatedly execute vulnerable code?

*Aldaya et al. "Port contention for fun and profit." (SP’19)*
Microarchitectural Replay Attacks* (MRAs)

Insight: Attacker triggers a large or unlimited number of pipeline squashes in the victim thread to replay vulnerable code

**Squash**

Victim (in SGX):

```
load x; // x is public
...
if (secret) {
    // use shared resource
    Execute!
} else {
    // don’t use shared resource
}
```

- Attacker clears page-table entry present bit of x and flushes TLB
- Victim speculatively executes vulnerable code
- Page fault occurs in the victim thread. Victim squashes the pipeline
- Victim invokes OS (controlled by attacker)

MRAs are beyond speculative execution side-channel attacks (e.g., Spectre)

* Skarlatos et al., “MicroScope: Enabling Microarchitectural Replay Attacks” (ISCA’19)
Generalized MRAs

Sources of squash: Exception, branch misprediction, memory consistency model violation

Attacker: Can be either supervisor- or user-level

Replay handle: Load, branch, instruction that can raise exceptions

Victim: Any instruction
Jamais Vu: the 1st Defense Mechanism to Thwart MRAs

Intuition: detect instructions that have been squashed and protect their re-execution with Fences

\[
\{V_0, V_1, \ldots, V_n\}
\]
Jamais Vu: the 1\textsuperscript{st} Defense Mechanism to Thwart MRAs

**Intuition:** detect instructions that have been squashed and protect their re-execution with Fences

![Diagram showing Victms, Replay Handle, ROB Head, and Check process with V\(_n\), V\(_1\), ..., V\(_n\)]
Jamais Vu: the 1\textsuperscript{st} Defense Mechanism to Thwart MRAs

**Intuition:** detect instructions that have been squashed and protect their re-execution with Fences

Fence delays an instruction execution until it is guaranteed to retire

\textbf{Victims} \quad \textbf{ROB Head} \\
\begin{align*}
V_n & \quad \cdots \quad V_1 \quad V_0 \quad H
\end{align*}

\{V_0, V_1, \ldots, V_n\}
Jamais Vu: the 1\textsuperscript{st} Defense Mechanism to Thwart MRAs

**Intuition:** detect instructions that have been squashed and protect their re-execution with Fences

\[
\{V_0, V_1, \ldots, V_n\}
\]

**Victims**

**Fenced**

"Forget" the information at some point
Jamais Vu: the 1st Defense Mechanism to Thwart MRAs

1. How to record squashed instructions?

2. For how long to keep it?

Jamais Vu

Counter   Clear-on-Retire   Epoch

Trade-offs between security, execution overhead, and implementation complexity
Scheme 1: Counter

**Intuition:** For each static instruction, use a counter to record the difference between squashes and retirements.

<table>
<thead>
<tr>
<th>Instruction PC</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC(H)</td>
<td>0</td>
</tr>
<tr>
<td>PC(V₀)</td>
<td>0</td>
</tr>
<tr>
<td>PC(V₁)</td>
<td>0</td>
</tr>
<tr>
<td>PC(Vₙ)</td>
<td>0</td>
</tr>
</tbody>
</table>
**Scheme 1: Counter**

**Intuition:** For each static instruction, use a counter to record the difference between squashes and retirements

<table>
<thead>
<tr>
<th>Instruction PC</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC(H)</td>
<td>0</td>
</tr>
<tr>
<td>PC(V₀)</td>
<td>1</td>
</tr>
<tr>
<td>PC(V₁)</td>
<td>1</td>
</tr>
<tr>
<td>PC(Vₙ)</td>
<td>1</td>
</tr>
</tbody>
</table>

**Squash:** Increment counters of squashed instructions
Scheme 1: Counter

**Intuition:** For each static instruction, use a counter to record the difference between squashes and retirements

<table>
<thead>
<tr>
<th>Instruction PC</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC(H)</td>
<td>0</td>
</tr>
<tr>
<td>PC(V₀)</td>
<td>1</td>
</tr>
<tr>
<td>PC(V₁)</td>
<td>1</td>
</tr>
<tr>
<td>PC(Vₙ)</td>
<td>1</td>
</tr>
</tbody>
</table>

**Refill:** Fence if the instruction’s counter > 0
Scheme 1: Counter

**Intuition:** For each static instruction, use a counter to record the difference between squashes and retirements

Refill: Fence if the instruction’s counter > 0

<table>
<thead>
<tr>
<th>Instruction PC</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC(H)</td>
<td>0</td>
</tr>
<tr>
<td>PC(V₀)</td>
<td>1</td>
</tr>
<tr>
<td>PC(V₁)</td>
<td>1</td>
</tr>
<tr>
<td>PC(Vₙ)</td>
<td>1</td>
</tr>
</tbody>
</table>
Scheme 1: Counter

**Intuition:** For each static instruction, use a counter to record the difference between squashes and retirements

<table>
<thead>
<tr>
<th>Instruction PC</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC(H)</td>
<td>0</td>
</tr>
<tr>
<td>PC(V₀)</td>
<td>0</td>
</tr>
<tr>
<td>PC(V₁)</td>
<td>1</td>
</tr>
<tr>
<td>PC(Vₙ)</td>
<td>1</td>
</tr>
</tbody>
</table>

**Retire:** Decrement counters of retired instructions (if counter > 0)

*Bound replays to retirements*
Counter: Implementation

Find counters in memory

Bring counters to pipeline

- **Hit**: check count > 0 before execution
- **Miss**: apply fence, fetch counter when safe
Scheme 2: Clear-on-Retire (CoR)

**Intuition:** Use a set-like structure, namely Squashed Buffer (SB), to record PCs of squashed instructions and the replay handle. Clear the buffer as soon as the program makes forward progress.
Scheme 2: Clear-on-Retire (CoR)

**Intuition:** Use a set-like structure, namely Squashed Buffer (SB), to record PCs of squashed instructions and the replay handle. Clear the buffer as soon as the program makes forward progress.

**Squash:** Add PCs of squashed instructions to PC Buffer, update Handle ID to the *Replay Handle*.
Scheme 2: Clear-on-Retire (CoR)

**Intuition:** Use a set-like structure, namely Squashed Buffer (SB), to record PCs of squashed instructions and the replay handle. Clear the buffer as soon as the program makes forward progress.

Refill: Fence if the instruction’s PC is found in SB
Scheme 2: Clear-on-Retire (CoR)

**Intuition:** Use a set-like structure, namely Squashed Buffer (SB), to record PCs of squashed instructions and the replay handle. Clear the buffer as soon as the program makes forward progress.

- **Fenced**
  - \( V_n \) \( \ldots \) \( V_1 \) \( V_0 \) \( H \)
- **Victims**
- **ROB Head**
- **Replay Handle**

**Refill:** Fence if the instruction’s PC is found in SB.
Scheme 2: Clear-on-Retire (CoR)

Intuition: Use a set-like structure, namely Squashed Buffer (SB), to record PCs of squashed instructions and the replay handle. Clear the buffer as soon as the program makes forward progress.

Replay Handle Retire: Clear SB

Ensure program makes forward progress
Clear-on-Retire: PC Buffer Design

**PC Buffer:** Tests whether a given PC belongs to a set of PCs ⇒ Bloom Filter

False Negatives? Impossible!

False Positives? Possible, lead to over-fencing (safe)
Insight: Leakages are typically associated with execution locality. Once program execution moves to another locality, the same victim instruction is likely to reveal different information:

```
for i in 1..N
    x = secrets[i];
    handle; // H
    victim(x); // V
```

Victim instructions that are from different localities should be handled separately.

Possible localities: a loop iteration, a whole loop, or a subroutine.
Scheme 3: Epoch

**Intuition:** Compiler identifies execution localities (i.e., Epochs). Hardware allocates a different PC Buffer for each Epoch.

**Squash:** Add squashed instructions to their corresponding PC buffers
Scheme 3: Epoch

**Intuition:** Compiler identifies execution localities (i.e., Epochs). Hardware allocates a different PC Buffer for each Epoch.

**Refill:** Fence if the instruction’s PC is found in corresponding PC buffer
Scheme 3: Epoch

**Intuition:** Compiler identifies execution localities (i.e., Epochs). Hardware allocates a different PC Buffer for each Epoch.

Refill: Fence if the instruction’s PC is found in corresponding PC buffer.
**Scheme 3: Epoch**

**Intuition:** Compiler identifies execution localities (i.e., Epochs). Hardware allocates a different PC Buffer for each Epoch.

**Epoch Retire:** Clear the PC buffer that is associated with the retired Epoch.
Scheme 3: Epoch-Rem

**Intuition:** Compiler identifies execution localities (i.e., Epochs). Hardware allocates a different PC Buffer for each Epoch.

Instruction Retire (Optional): Remove the instruction’s PC from the PC buffer (**Epoch-Rem**)

---

**Diagram:**
- **Epoch 4**
- **ROB Head**
- **Squashed Buffer (SB)**
  - Epoch ID
  - PC Buffer (PC Buffer)
  - **Fenced**
  - **Retiring**

- **Instruction Retire (Optional):** Remove the instruction’s PC from the PC buffer (**Epoch-Rem**)
Test whether a PC belongs to a multi-set of PCs and support removal ⇒ Counting Bloom Filter

False Negatives? Possible, lead to under-fencing (unsafe)
- Rarely happen (~0.02%)
- Cannot be controlled by attackers

False Positives? Possible, lead to over-fencing (safe)
Bounding Squashes

Example A: straight-line code, non-transient victim, exception

\[
x = \text{secret}; \\
\text{handle 1; // except.} \\
\text{handle 2; // except.} \\
\ldots \\
\text{victim(x);} \\
\]

Example B: loop, transient victim, branch misprediction

\[
\text{for } i \text{ in } 1..N \\
x = \text{secrets}[i]; \\
\text{if } (/*false*/) \\
\text{ victim(x);} \\
\]

Example C: loop, transient victim leaks the same data, branch misprediction

\[
\text{for } i \text{ in } 1..N \\
x = \text{secrets}[i]; \\
\text{if } (/*false*/) \\
\text{ victim(x);} \\
\]

1. Source of squash?

2. Victim is transient?

3. Victim is in a loop leaking the same secret every iteration?
Bounding Squashes

Example A: straight-line code, non-transient victim, exception

```plaintext
x = secret;
handle 1; // except.
handle 2; // except.
...
victim(x);
```

Example B: loop, transient victim, branch misprediction

```plaintext
for i in 1..N
    x = secrets[i];
    if(/*false*/) { // handle
        victim(x);
    }
```  

Example C: loop, transient victim leaks the same data, branch misprediction

```plaintext
for i in 1..N
    if(/*false*/) { // handle
        victim(x);
    }
```

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Example A</th>
<th>Example B</th>
<th>Example C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter</td>
<td>1</td>
<td>1</td>
<td>(K^\dagger)</td>
</tr>
<tr>
<td>Clear-on-Retire</td>
<td></td>
<td></td>
<td>(</td>
</tr>
<tr>
<td>Epoch-Rem-Iter</td>
<td>1</td>
<td>1</td>
<td>(N)</td>
</tr>
<tr>
<td>Epoch-Rem-Loop</td>
<td>1</td>
<td>1</td>
<td>(K)</td>
</tr>
</tbody>
</table>

\(?^\dagger\): number of unrolled iterations that fit in the ROB
## Summary of Designs

<table>
<thead>
<tr>
<th>Scheme</th>
<th>How to record?</th>
<th>For how long?</th>
<th>Protection</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter</td>
<td>Count associated with static instruction</td>
<td>Forever</td>
<td>Strong</td>
<td>Complex</td>
</tr>
<tr>
<td>Clear-on-Retire</td>
<td></td>
<td>Until replay handle instruction retires</td>
<td>Weak</td>
<td>Simple</td>
</tr>
<tr>
<td>Epoch-Rem-Iter</td>
<td>Squashed Buffer (SB) associated with ROB</td>
<td>Until an entire loop iteration retires</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Epoch-Rem-Loop</td>
<td></td>
<td>Until the entire loop retires</td>
<td>Strong</td>
<td>Medium</td>
</tr>
</tbody>
</table>
Evaluation: Execution Overhead (SPEC 2017)

Evaluated Schemes:
- **CoR**: Clear-on-Retire scheme
- **Epoch-Rem-Iter**: Epoch-Rem with iteration
- **Epoch-Rem-Loop**: Epoch-Rem with loop
- **Counter**: Counter scheme

Geo. Mean of Execution Overhead over unsafe core
Conclusion

- Jamais Vu is the first defense mechanism to thwart MRAs
- Jamais Vu includes several designs with different tradeoffs between security, execution overhead, and complexity
- *Epoch-Rem-Loop*, the most secure design, only has an average execution overhead of 13.8% in benign execution; *CoR*, the simplest scheme, only has an average execution overhead of 2.9%

Open Source: [https://github.com/dskarlatos/JamaisVu](https://github.com/dskarlatos/JamaisVu)
Jamais Vu: Thwarting Microarchitectural Replay Attacks

Dimitrios Skarlatos†, Zirui Neil Zhao†, Riccardo Paccagnella,
Christopher Fletcher, Josep Torrellas

University of Illinois    Carnegie Mellon University

† Authors contributed equally to this work

ASPLOS’21