Jamais Vu: Thwarting Microarchitectural Replay Attacks

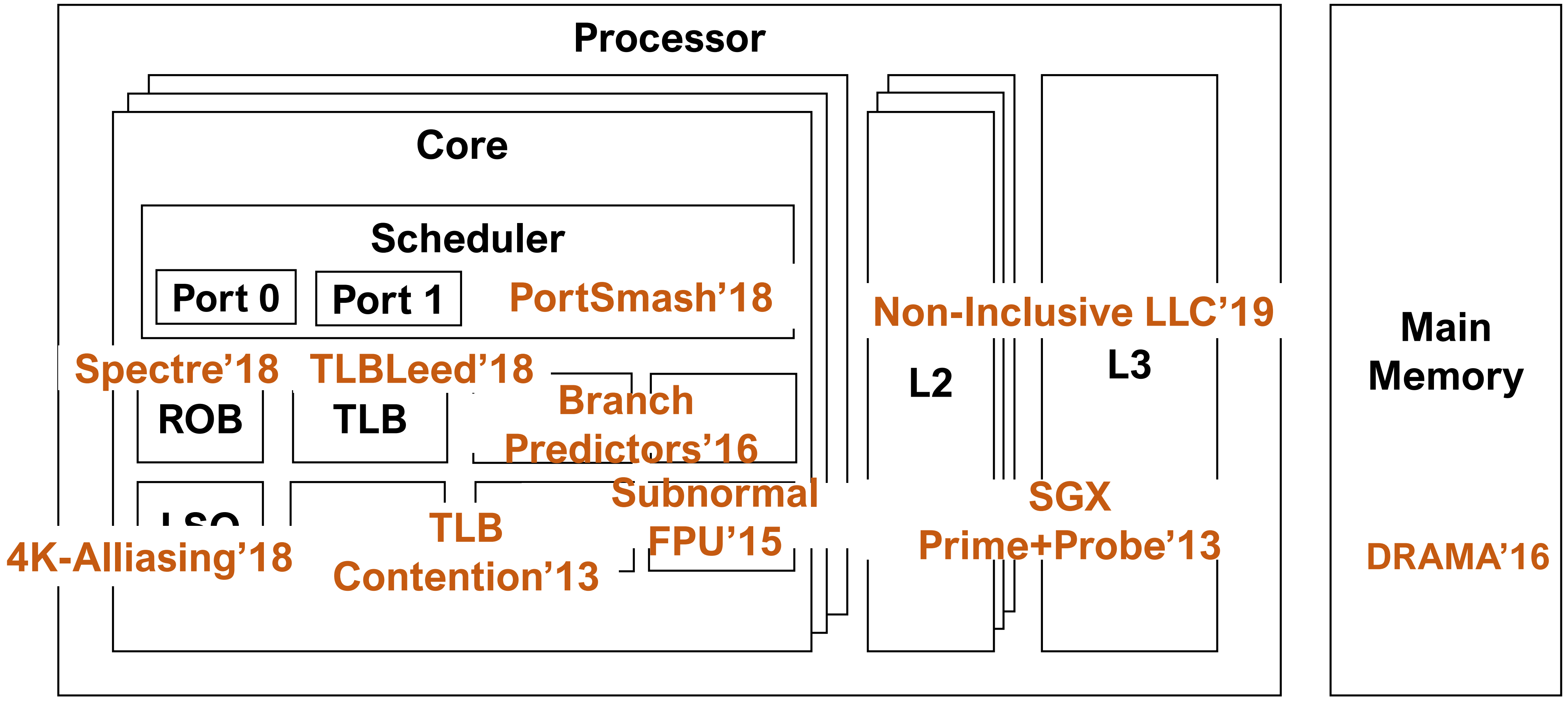
Dimitrios Skarlatos† , **Zirui Neil Zhao**†, Riccardo Paccagnella,
Christopher Fletcher, Josep Torrellas

University of Illinois   Carnegie Mellon University

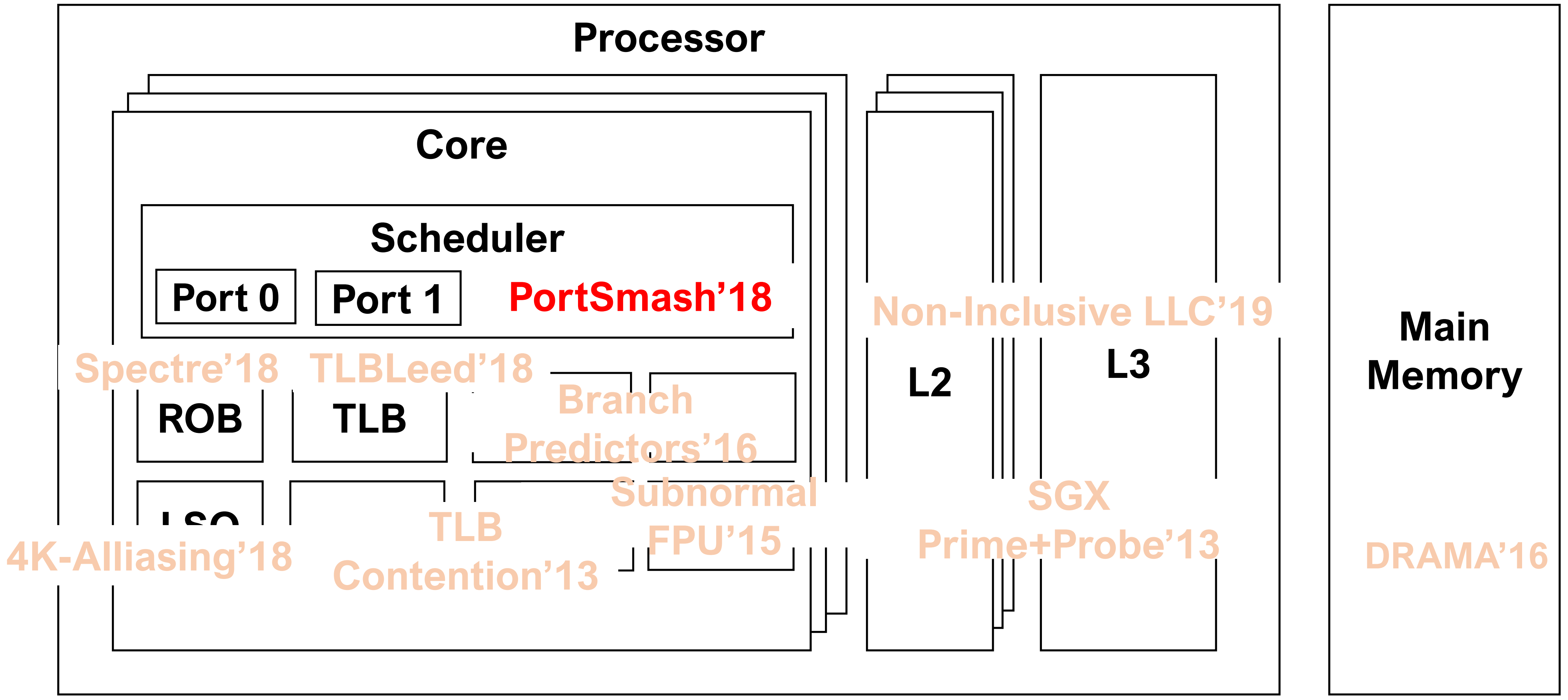
† Authors contributed equally to this work

ASPLOS’21

The Era of Side-Channels



The Era of Side-Channels



Port Contention Attack*

Victim (in SGX):

```
if (secret) {
  // use shared resource
} else {
  // don’t use shared resource
}
```

Attacker (controls OS):

```
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;

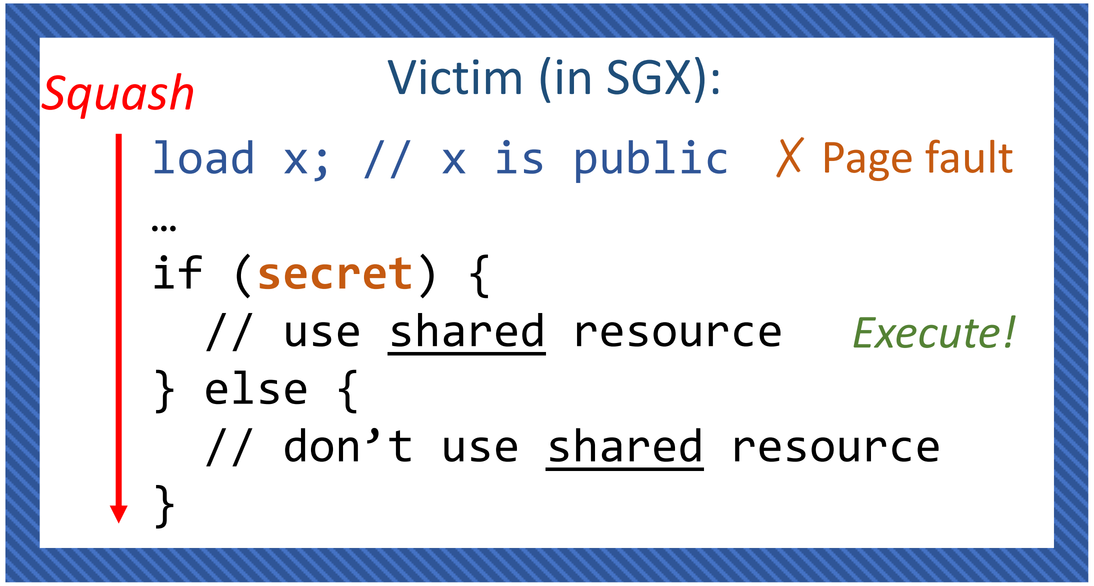
However, this side-channel is noisy, attacker needs repeated victim execution to be confident

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



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

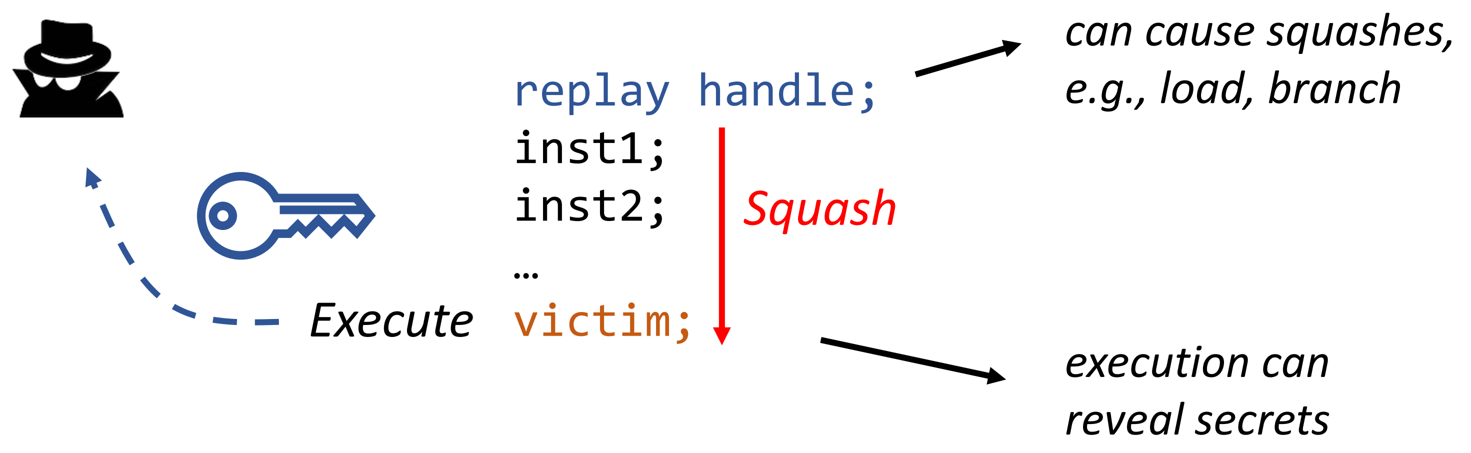
- Attacker clears page-table entry present bit of *x* and flushes TLB
- Victim speculatively executes vulnerable code
- Page fault occurs in the victim thread. Victim squashes the pipeline
- Victim invokes OS (controlled by attacker)

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


replay handle;
inst1;
inst2;
…
victim;
Squash
Execute
can cause squashes, e.g., load, branch
execution can reveal secrets

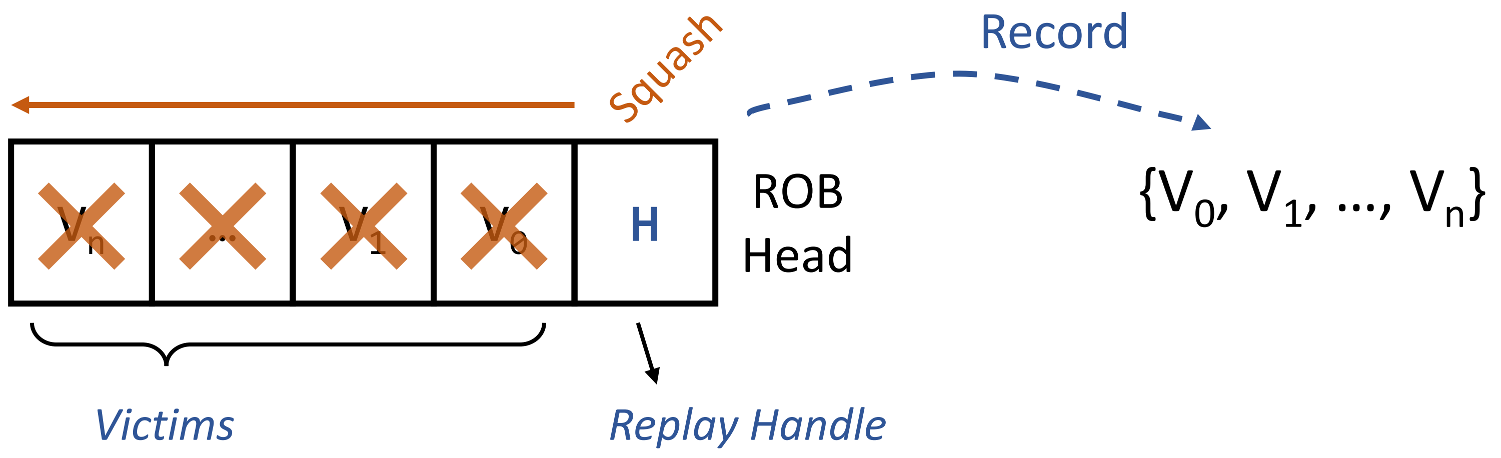
Attacker: Can be either supervisor- or user-level

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

Victim: Any instruction

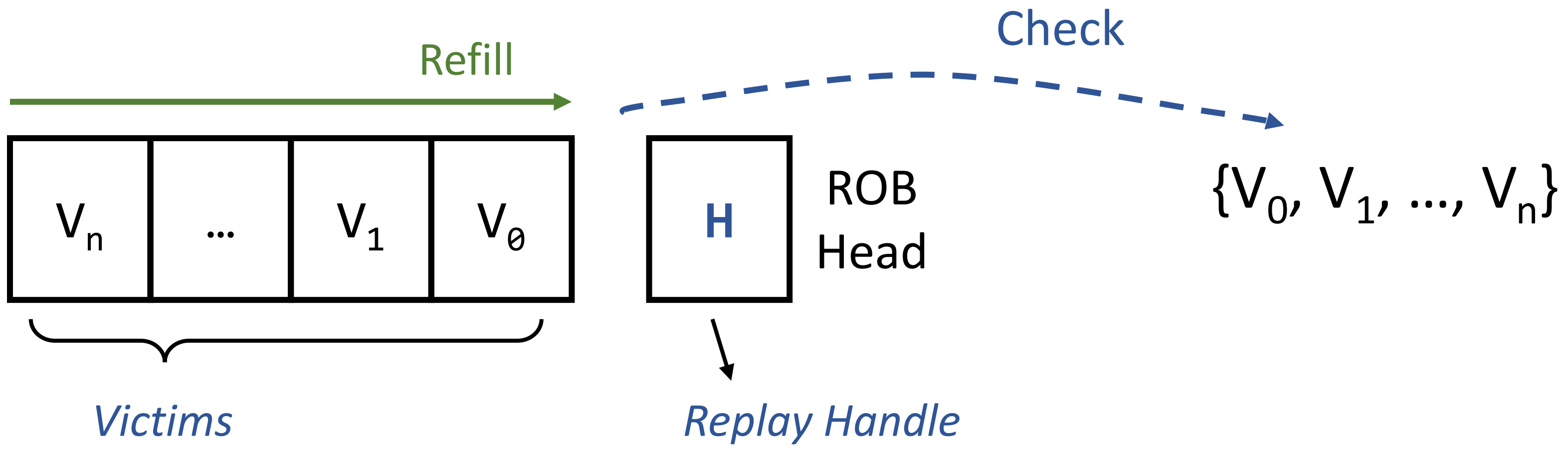
Jamais Vu: the 1st Defense Mechanism to Thwart MRAs

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



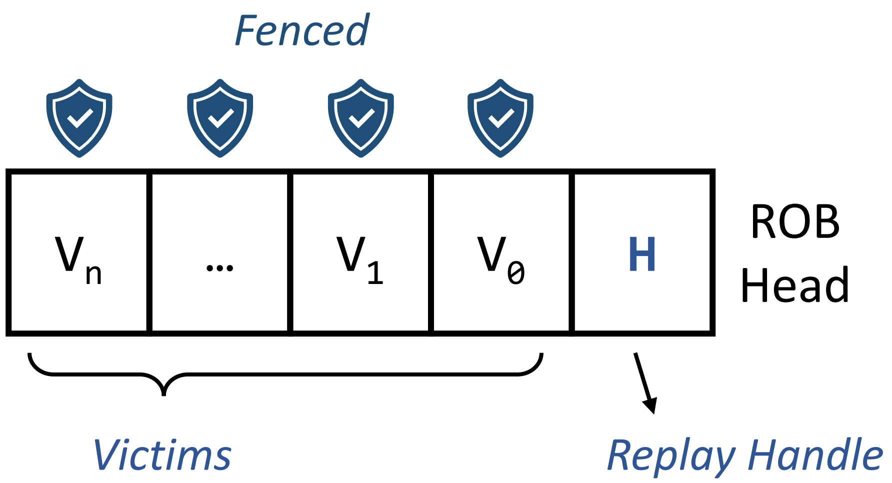
Jamais Vu: the 1st Defense Mechanism to Thwart MRAs

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



Jamais Vu: the 1st Defense Mechanism to Thwart MRAs

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

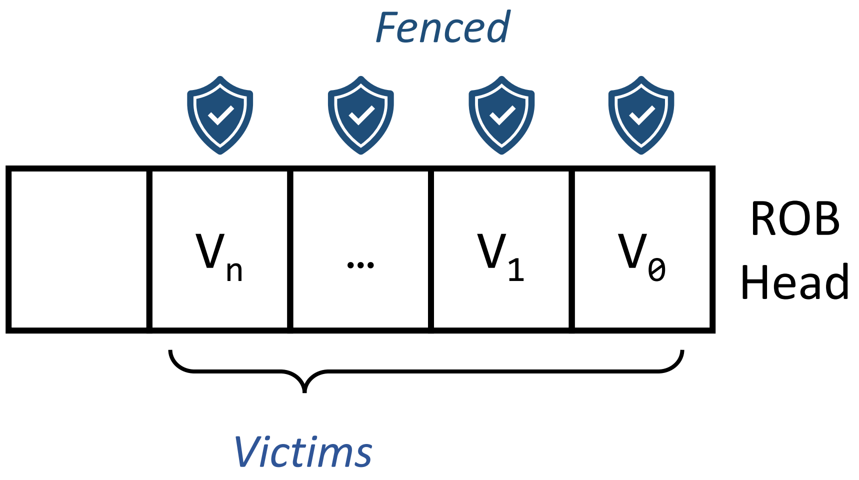


$\{V_0, V_1, \ldots, V_n\}$

Fence delays an instruction execution until it is guaranteed to retire

Jamais Vu: the 1st Defense Mechanism to Thwart MRAs

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



"Forget" the information at some point

$\{V_0, V_1, \ldots, V_n\}$

Jamais Vu: the 1st Defense Mechanism to Thwart MRAs

1. How to record squashed instructions?

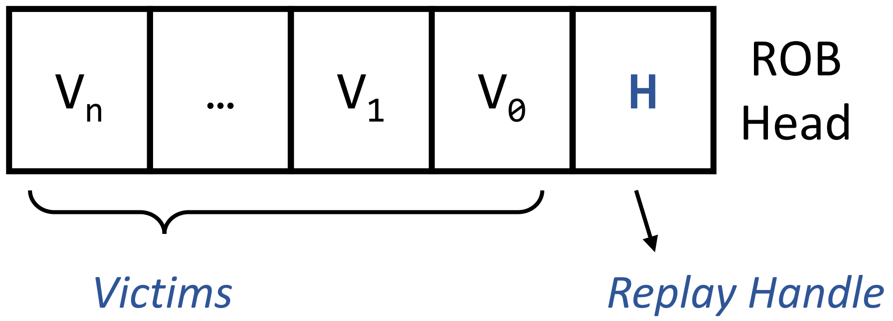
2. For how long to keep it?

$$\begin{tikzcd}
 & \textit{Jamais Vu} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Counter} & \text{Clear-on-Retire} & \text{Epoch}
\end{tikzcd}$$

Trade-offs between security, execution overhead, and implementation complexity

Scheme 1: Counter

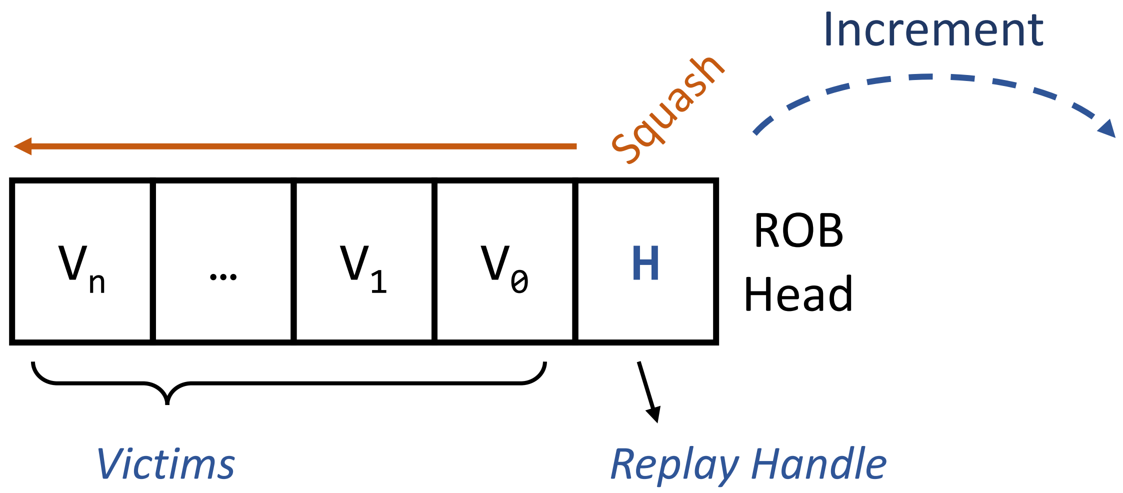
Intuition: For each static instruction, use a counter to record the difference between squashes and retirements



Instruction PC	Counter
PC(H)	0
PC(V_0)	0
PC(V_1)	0
PC(V_n)	0

Scheme 1: Counter

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

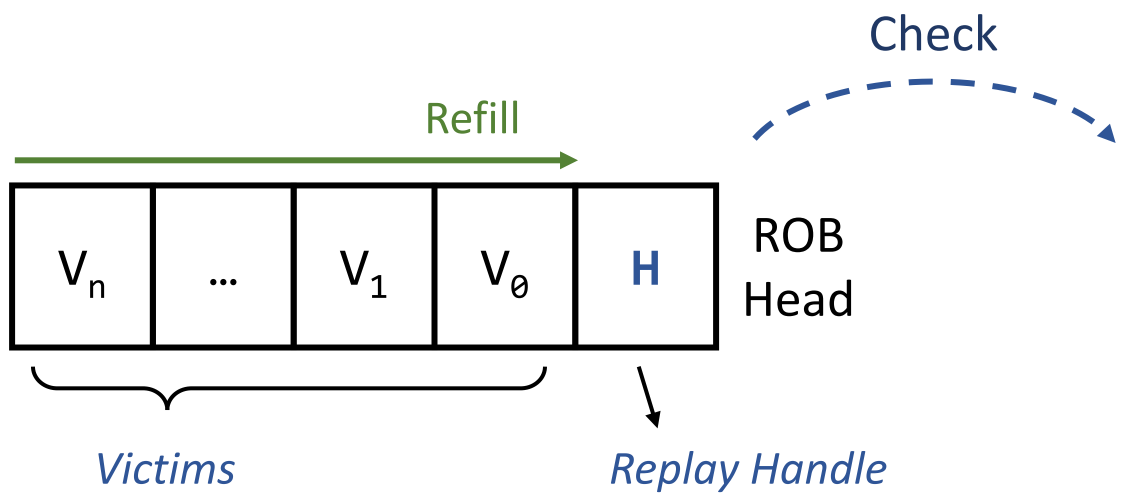


Instruction PC	Counter
PC(H)	0
PC(V_0)	1
PC(V_1)	1
PC(V_n)	1

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

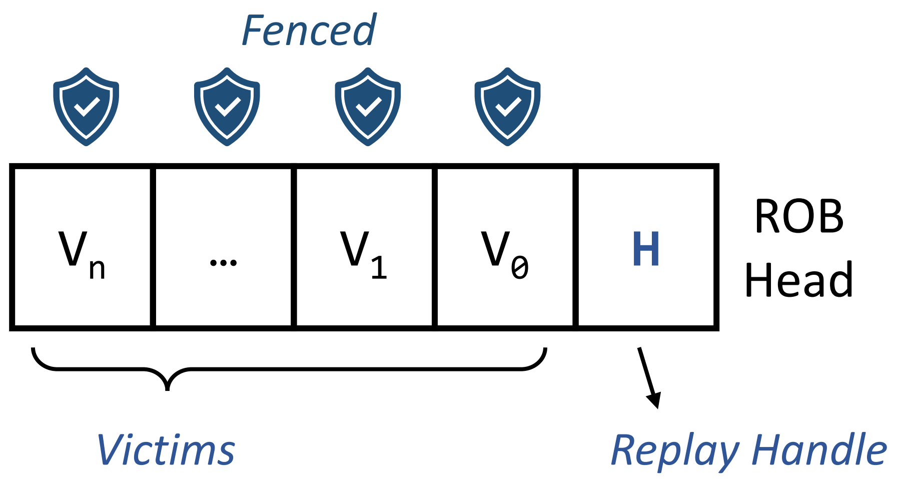


Instruction PC	Counter
PC(H)	0
PC(V_0)	1
PC(V_1)	1
PC(V_n)	1

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

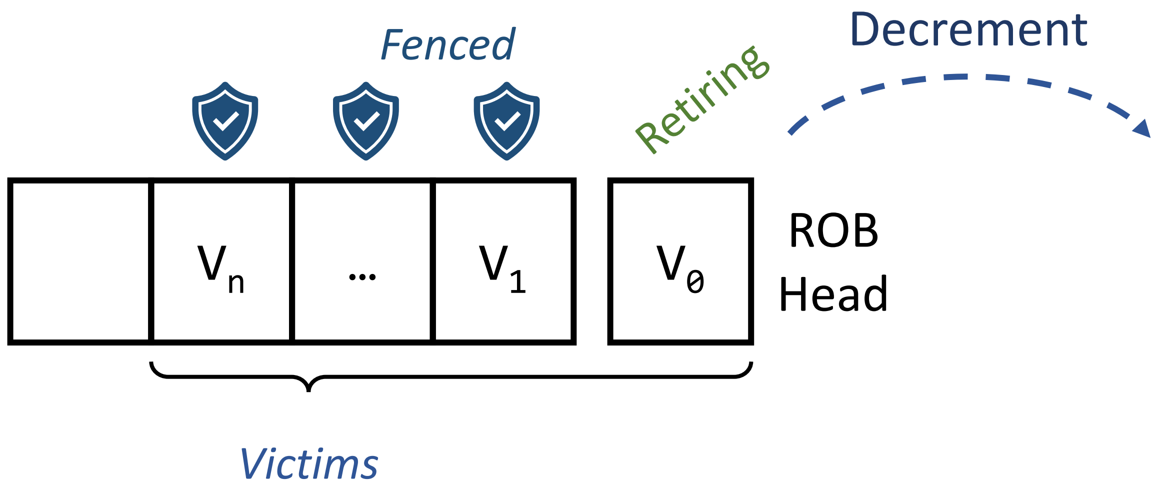


Instruction PC	Counter
PC(H)	0
PC(V_0)	1
PC(V_1)	1
PC(V_n)	1

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



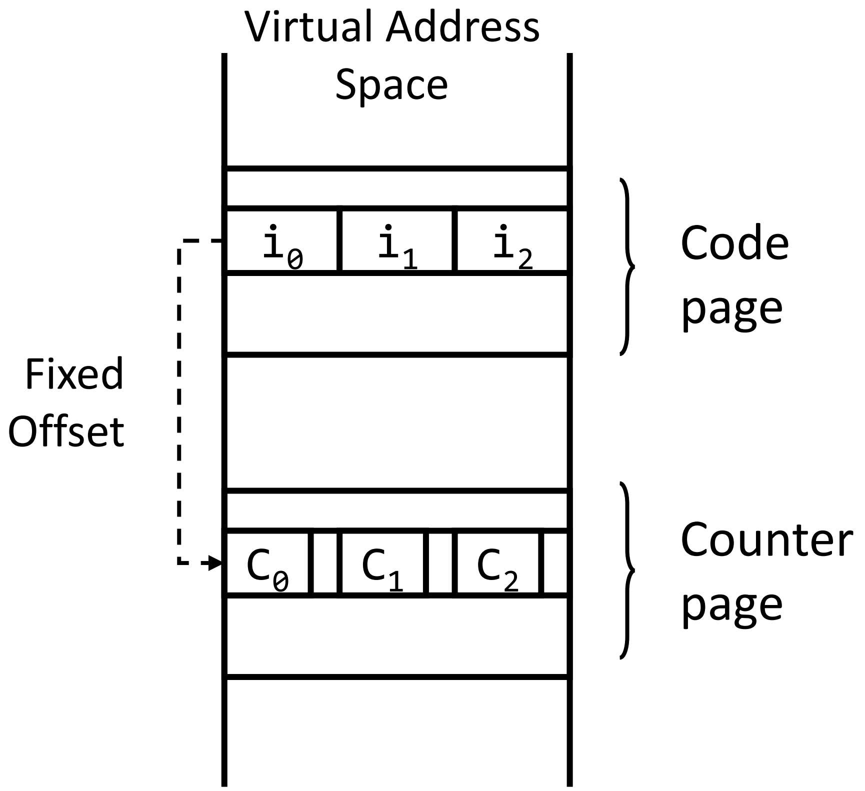
Instruction PC	Counter
PC(H)	0
PC(V_0)	**0**
PC(V_1)	1
PC(V_n)	1

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

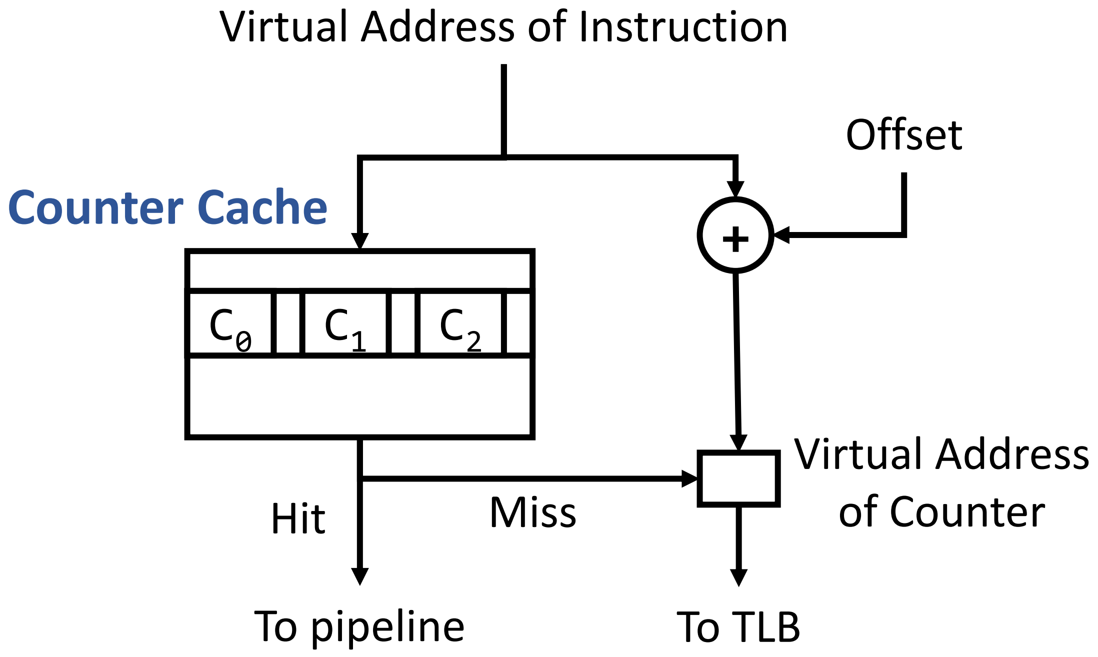
Bound replays to retirements

Counter: Implementation

Find counters in memory



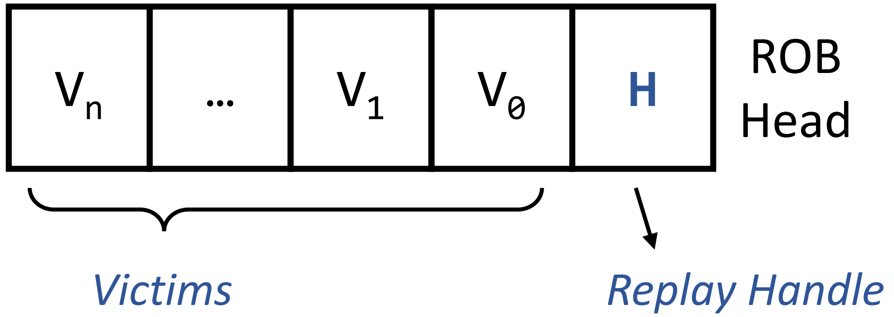
Bring counters to pipeline

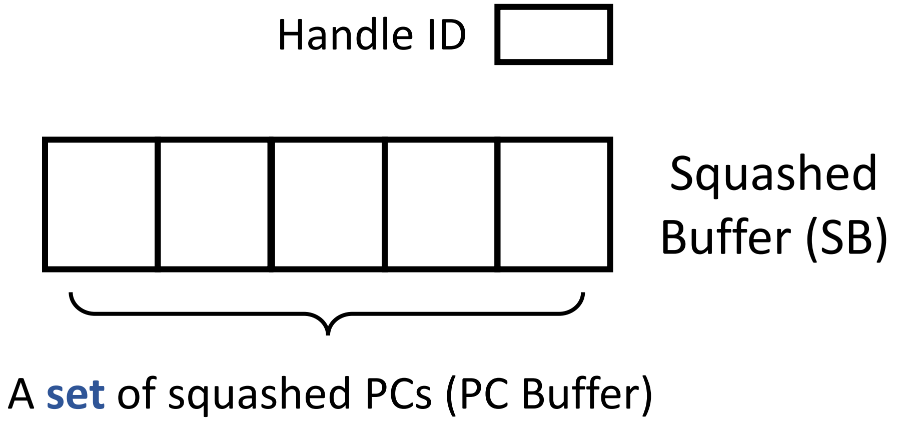


- Hit: check count > 0 before execution
- Miss: apply fence, fetch counter when safe

Scheme 2: Clear-on-Retire (CoR)

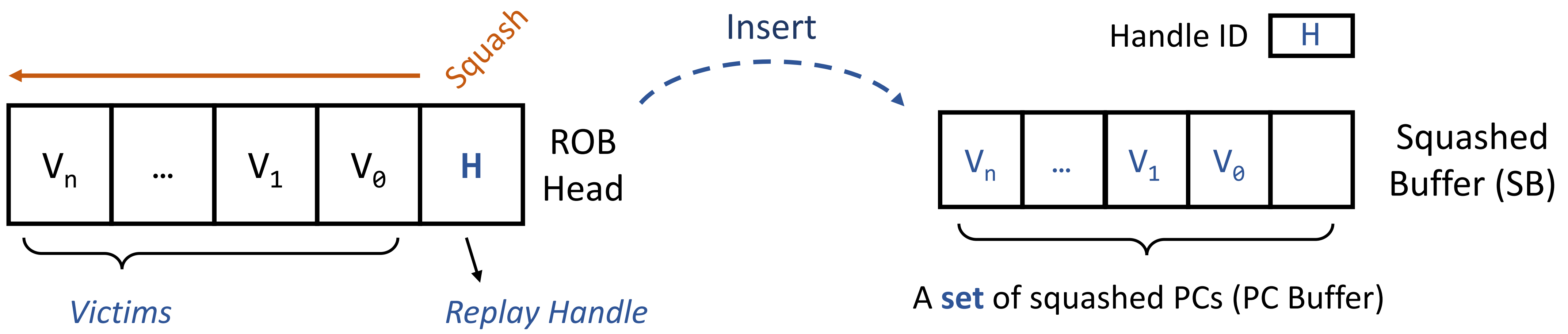
Intuition: Use a set-like structure, namely Squashed Buffer (SB), to record PCs of squashed instructions and the replay handle. Clear the buffer as soon as the program makes forward progress





Scheme 2: Clear-on-Retire (CoR)

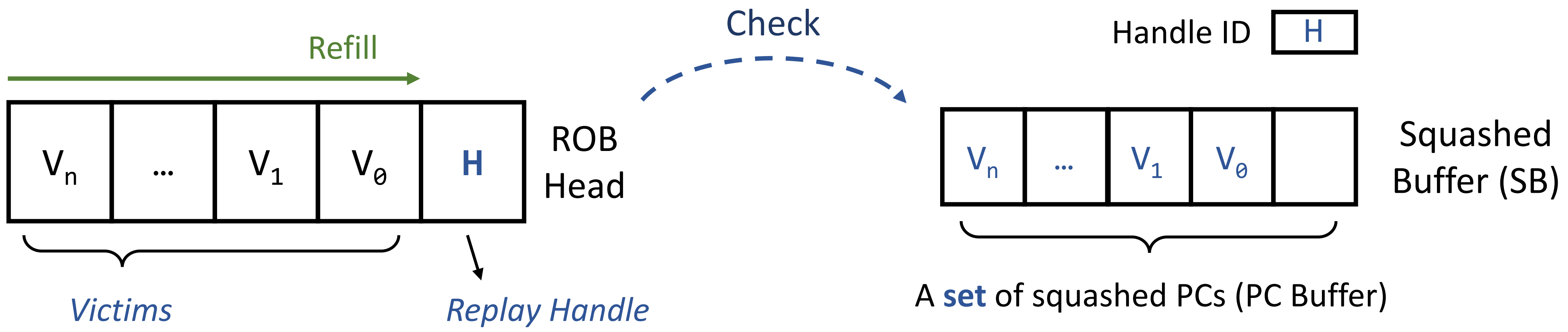
Intuition: Use a set-like structure, namely Squashed Buffer (SB), to record PCs of squashed instructions and the replay handle. Clear the buffer as soon as the program makes forward progress



Squash: Add PCs of squashed instructions to PC Buffer, update Handle ID to the *Replay Handle*

Scheme 2: Clear-on-Retire (CoR)

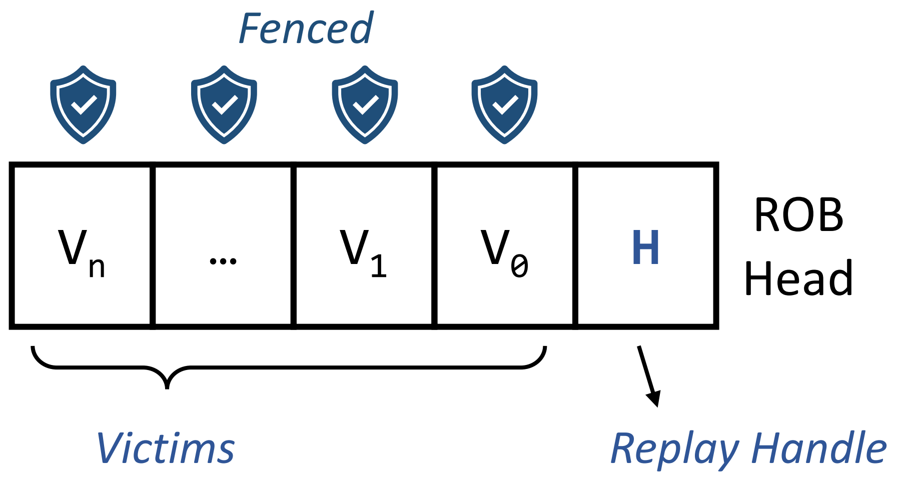
Intuition: Use a set-like structure, namely Squashed Buffer (SB), to record PCs of squashed instructions and the replay handle. Clear the buffer as soon as the program makes forward progress

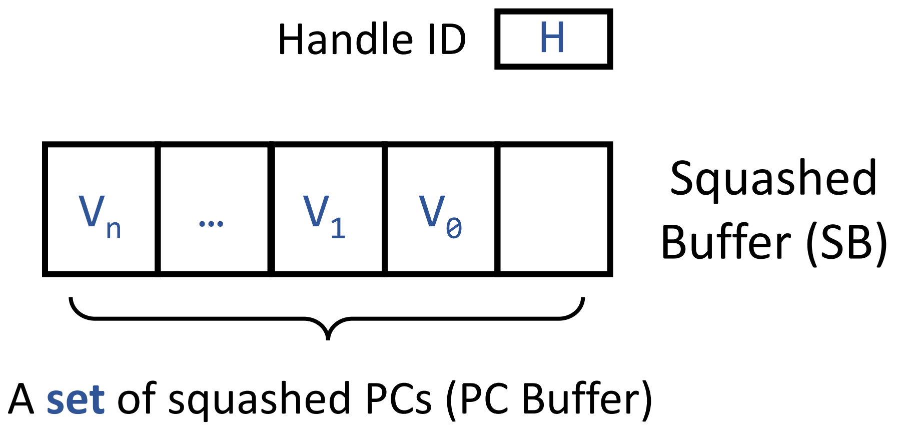


Refill: Fence if the instruction's PC is found in SB

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

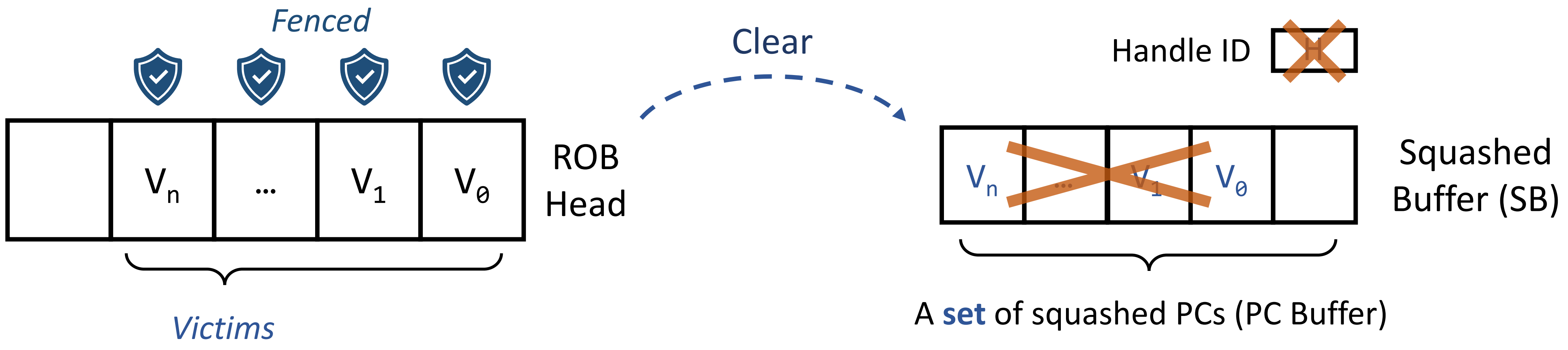




Refill: Fence if the instruction’s PC is found in SB

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

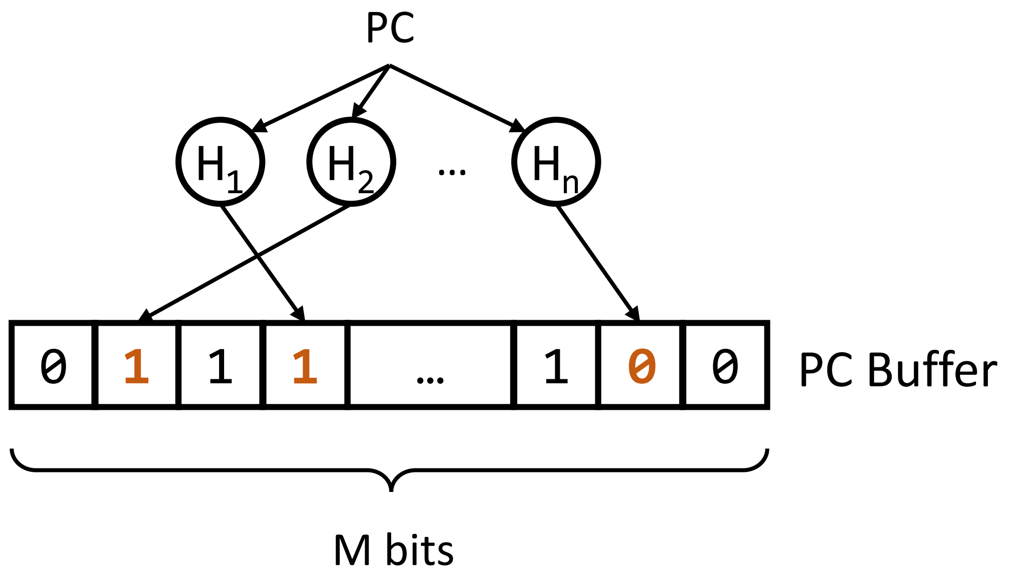


Replay Handle Retire: Clear SB

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)

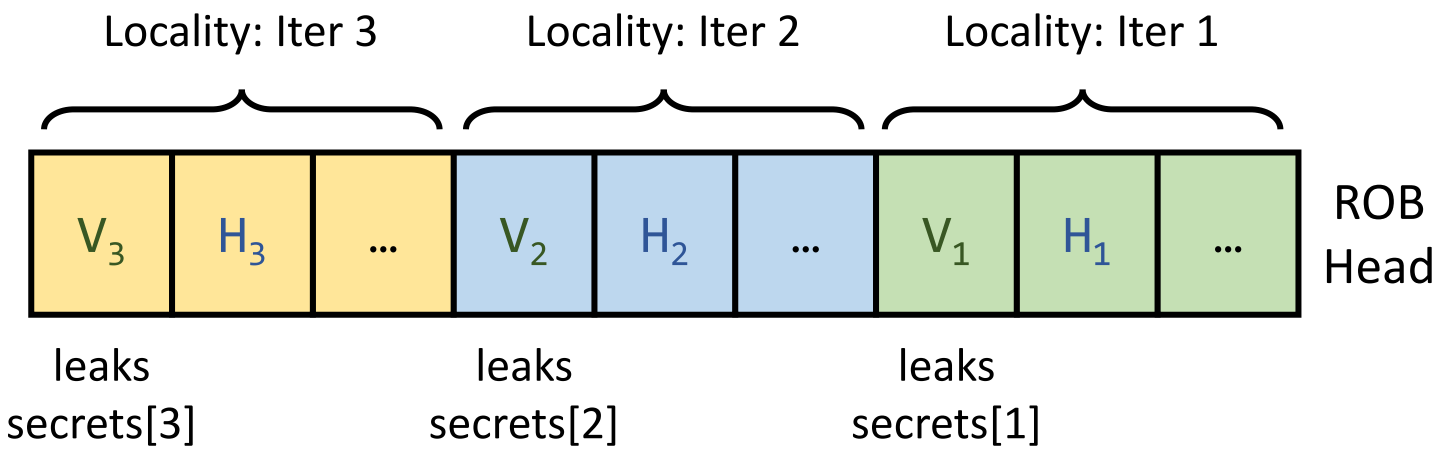
Scheme 3: Epoch

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
```

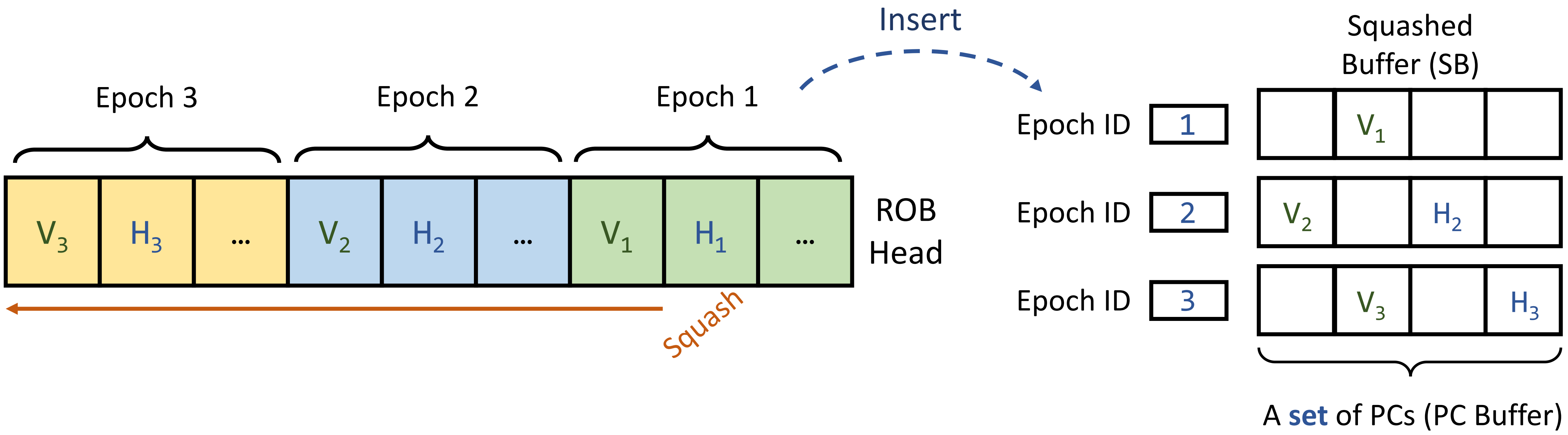
Victim instructions that are from different localities should be handled separately

Possible localities: a loop iteration, a whole loop, or a subroutine



Scheme 3: Epoch

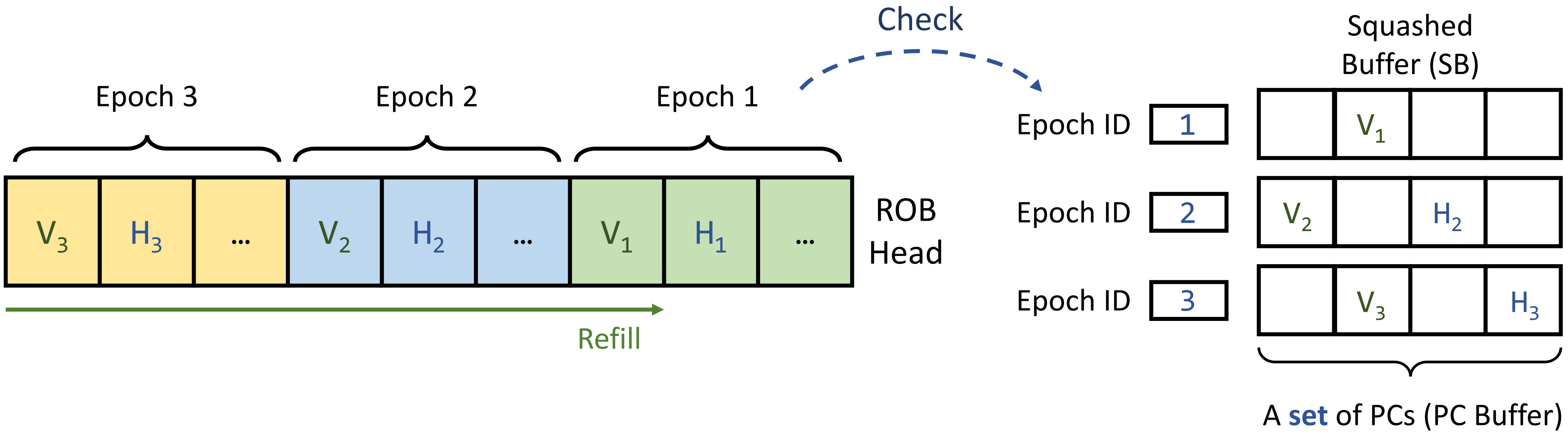
Intuition: Compiler identifies execution localities (i.e., Epochs). Hardware allocates a different PC Buffer for each Epoch.



Squash: Add squashed instructions to their corresponding PC buffers

Scheme 3: Epoch

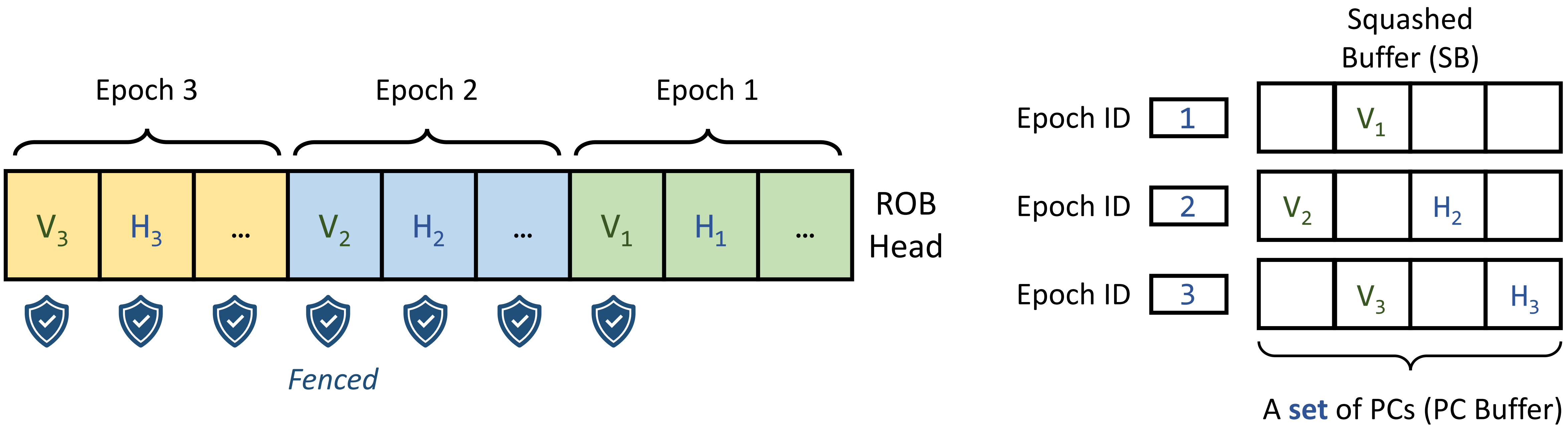
Intuition: Compiler identifies execution localities (i.e., Epochs). Hardware allocates a different PC Buffer for each Epoch.



Refill: Fence if the instruction's PC is found in corresponding PC buffer

Scheme 3: Epoch

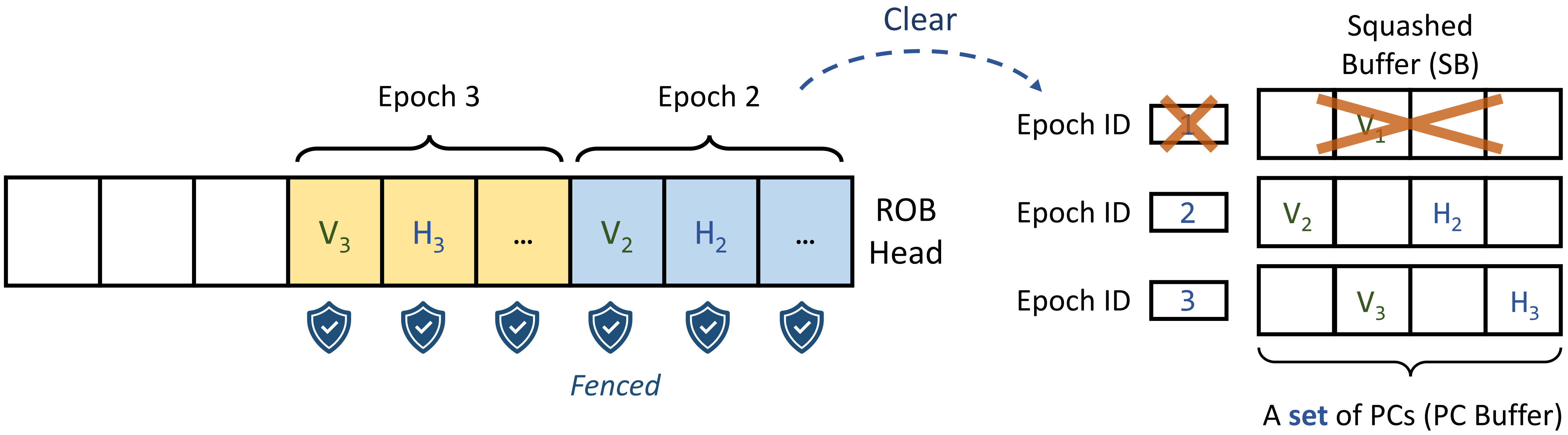
Intuition: Compiler identifies execution localities (i.e., Epochs). Hardware allocates a different PC Buffer for each Epoch.



Refill: Fence if the instruction's PC is found in corresponding PC buffer

Scheme 3: Epoch

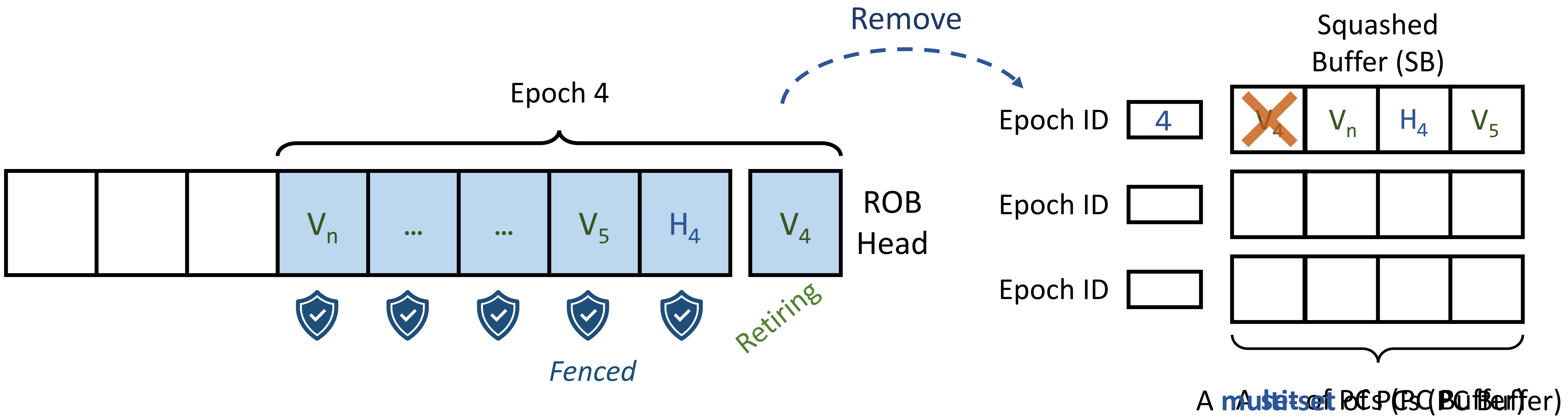
Intuition: Compiler identifies execution localities (i.e., Epochs). Hardware allocates a different PC Buffer for each Epoch.



Epoch Retire: Clear the PC buffer that is associated with the retired Epoch

Scheme 3: Epoch-Rem

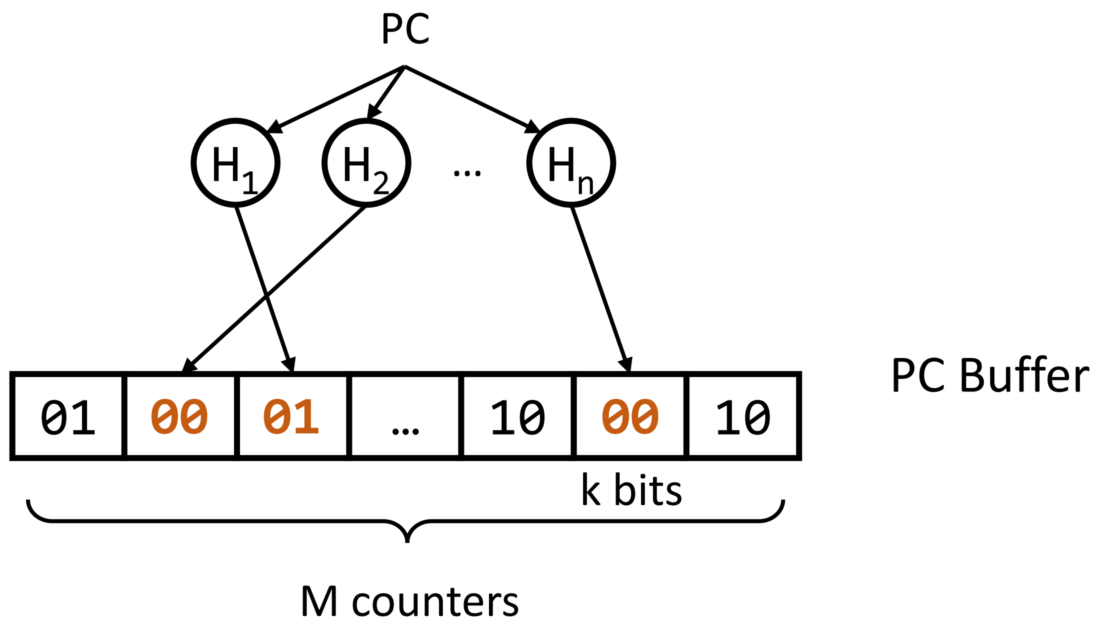
Intuition: Compiler identifies execution localities (i.e., Epochs). Hardware allocates a different PC Buffer for each Epoch.



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

Epoch-Rem: PC Buffer Design

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

Example B: loop, transient victim, branch misprediction

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

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

```
for i in 1..N
  if (/*false*/) { // handle
    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

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

Example B: loop, transient victim, branch misprediction

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

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

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

	Number of Squashes		
Scheme	Example A	Example B	Example C
Counter	1	1	K†
Clear-on-Retire	|ROB|-1	K	K * N
Epoch-Rem-Iter	1	1	N
Epoch-Rem-Loop	1	1	K

† K: number of unrolled iterations that fit in the ROB

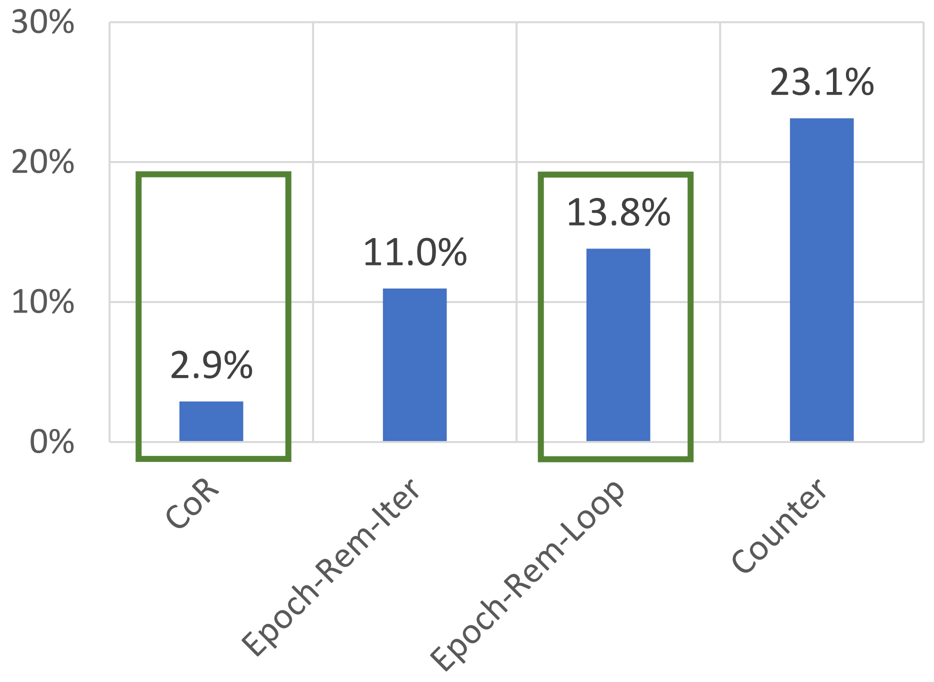
Summary of Designs

Scheme	How to record?	For how long?	Protection	Complexity
Counter	Count associated with static instruction	Forever	Strong	Complex
Clear-on-Retire	Squashed Buffer (SB) associated with ROB	Until replay handle instruction retires	Weak	Simple
Epoch-Rem-Iter		Until an entire loop iteration retires	Medium	Medium
Epoch-Rem-Loop		Until the entire loop retires	Strong	Medium

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

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