Jamais Vu: Thwarting Microarchitectural Replay Attacks

Dimitrios Skarlatos†, Zirui Neil Zhao†, Riccardo Paccagnella, Christopher W. Fletcher, Josep Torrellas
{skarlat2, ziruiz6, rp8, cwfletch, torrella}@illinois.edu
University of Illinois at Urbana-Champaign, IL, USA
† Authors contributed equally to this work.

ABSTRACT
Microarchitectural Replay Attacks (MRAs) enable an attacker to eliminate the measurement variation in potentially any microarchitectural side channel—even if the victim instruction is supposed to execute only once. In an MRA, the attacker forces pipeline flushes in order to repeatedly re-execute the victim instruction and denoise the channel. MRAs are not limited to transient execution attacks: the replayed victim can be an instruction that will eventually retire.

This paper presents the first technique to thwart MRAs. The technique, called Jamais Vu, detects when an instruction is squashed. Then, as the instruction is re-inserted into the pipeline, Jamais Vu automatically places a fence before it to prevent the attacker from squashing it again. This paper presents several Jamais Vu designs that offer different trade-offs between security, execution overhead, and implementation complexity. One design, called Epoch-Loop-Retire, effectively mitigates MRAs, has an average execution time overhead of 13.8% in benign executions, and only needs counting Bloom filters. An even simpler design, called Clear-on-Retire, has an average execution time overhead of only 2.9%, although it is less secure.

CCS CONCEPTS
• Security and privacy → Side-channel analysis and countermeasures.

KEYWORDS
Side-channel countermeasures, Processor design, Replay attack

1 INTRODUCTION
The microarchitecture of modern computer systems creates many side channels that allow an attacker running on a different process to exfiltrate execution information from a victim. Indeed, hardware resources such as caches [38, 43, 45, 59–61], TLBs [22], branch predictors [1, 16, 17], load-store units [39], execution ports [4, 6, 21], functional units [4, 5], and DRAM main memory [46] have all been shown to leak information.

Luckily, a limitation of these microarchitectural side channels is that they are often very noisy. To extract information, the execution of the attacker and the victim processes has to be carefully orchestrated [43, 45, 60], and often does not work as planned. Hence, the attacker needs to rely on many executions of the victim code section to obtain valuable information. Further, secrets in code sections that are executed only once or only a few times are hard to exfiltrate.

Unfortunately, a recently-introduced type of attack called Microarchitectural Replay Attack (MRA) [50] is able to eliminate the measurement variation in (i.e., to denoise) most microarchitectural side channels. This is the case even if the victim runs only once. Such capability makes the plethora of existing side-channel attacks look formidable and suggests the potential for a new wave of powerful side-channel attacks.

MRAs use the fact that, in out-of-order cores, pipeline squashes due to events such as exceptions, branch mispredictions, and memory consistency model violations trigger the re-execution of dynamic instructions. Hence, in an MRA, the attacker repeatedly squashes one or more instructions to force the squash and re-execution of a younger victim instruction V many times. This ability enables the attacker to cleanly observe the side-effects of V.

MRAs are powerful because they exploit a central mechanism in modern processors: out-of-order execution with in-order retirement. Moreover, MRAs are not limited to transient execution attacks: the instruction V that is replayed can be a correct instruction that will eventually retire. Finally, MRAs come in many forms. While the first MRA [50] exposed the side effects of V by repeatedly causing a page fault on an older instruction, similar results can be attained with other events that trigger pipeline flushes.

To thwart MRAs, one has to eliminate instruction replay or at least bound the number of replays that a victim instruction V may suffer. The goal is to deny the attacker the opportunity to see many executions of V.

This paper presents the first mechanism to thwart MRAs. We call it Jamais Vu. The simple idea is to record the instructions that are squashed. When any of these instructions is re-inserted into the Reorder Buffer (ROB), Jamais Vu automatically places a fence before it to prevent the attacker from squashing the instruction execution again. In reality, pipeline refill after a squash may not
Microarchitectural Replay Attack

ASPLOS ’21, April 19–23, 2021, Virtual, USA

D. Skarlatos, Z. N. Zhao, R. Paccagnella, C. W. Fletcher, J. Torrellas

Loop-Rem

2.3 Microarchitectural Replay Attacks

A Microarchitectural Replay Attack (MRA) \cite{skarlatos2020jamaisvu} uses one or more instructions to repeatedly trigger pipeline flushes, therefore forcing the re-execution of a younger instruction \( I \) multiple times. This capability enables the attacker to observe any side-effects of \( I \) multiple times, eliminating the measurement noise.

Skarlatos et al. \cite{skarlatos2020jamaisvu} introduced MRAs by using a malicious Operating System (OS) to repeatedly trigger page faults on a memory access instruction in an SGX environment. Specifically, the OS picks a memory access instruction called \textit{Replay Handle} that occurs shortly before a security-sensitive instruction \( I \). The OS sets up the attack by flushing the TLB entry that stores the virtual-to-physical translation of the replay handle access, and clearing the Present bit of the corresponding page table entry. The OS allows the program to resume execution and execute the replay handle. A TLB miss occurs, followed by a page walk. The instructions following the replay handle, including \( I \), execute in the shadow of the page walk, creating side effects: they leave some state in the cache subsystem or create contention for hardware structures in the core. This allows an attacker thread running in the system to perform a measurement of the secret data. At the end of the page walk, the hardware raises a page fault exception and squashes the instructions in the pipeline. The OS is then invoked to handle the page fault, but chooses to keep the Present bit cleared. The program then resumes and re-executes the replay handle, creating a TLB miss and page walk again. The instructions following the replay handle, including \( I \), execute again until a pipeline flush occurs. This process is repeated as many times as desired until the attacker extracts the secret information.

MRAs are more general than the specific instantiation prototyped by Skarlatos et al. \cite{skarlatos2020jamaisvu}. For example, there are multiple events that cause a pipeline flush, such as various exceptions, branch mispredictions, memory consistency model violations, and interrupts. Moreover, to trigger the repeated pipeline flushes, one does not need a privileged process. For example, Appendix A provides evidence, for the first time, that memory consistency model violations triggered by a non-privileged process can also create MRAs.

In this paper, we refer to the instruction that causes the pipeline flush as the \textit{Squashing} (S) instruction: we refer to the younger instructions in the ROB that the Squashing one squashes as the \textit{Victims} (V). The type of Victim instruction that the attacker wants to replay is one whose usage of and/or contention for a shared resource depends on a secret. We call such an instruction a \textit{transmitter}. Loads are obvious transmitters, as they use the shared cache hierarchy. However, many instructions can be transmitters, including those that use functional units.

3 THWARTING MRAS

3.1 Understanding the Types of MRAs

MRAs come in many forms. Table 1 shows three orthogonal characteristics that can help us understand these threats. The first one is the source of the squash. Recall that there are many sources, namely various exceptions (e.g., page faults \cite{skarlatos2020jamaisvu}), branch mispredictions, memory consistency model violations as shown in Appendix A, and interrupts \cite{skarlatos2020jamaisvu}. With some sources, a single Squashing instruction can trigger pipeline flushes repeatedly, while with others, a Squashing instruction can only flush the pipeline a very limited number of times. Examples of the former are attacker-controlled page faults and memory consistency model violations; examples of the latter are branch mispredictions. The former can create more leakage.

Moreover, some sources trigger the flush when the Squashing instruction is at the ROB head, while others can do it at any position in the ROB. The former include exceptions, while the latter include branch mispredictions and memory consistency violations. The former create more leakage because they typically squash and replay more Victims.

Figure 1(a) shows an example where repeated exceptions on one or more instructions \textit{inst} \(_j\) can squash and replay a transmitter
Table 1: Characteristics of microarchitectural replay attacks.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Why It Matters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source of squash?</td>
<td>Determines: (i) the number of pipeline flushes and (ii) where in the ROB the flush occurs</td>
</tr>
<tr>
<td>Victim is transient?</td>
<td>If yes, it can leak a wider variety of secrets</td>
</tr>
<tr>
<td>Victim is in a loop accessing the same secret every iteration?</td>
<td>If yes, it is harder to defend: (i) leaks from multiple iterations add up and (ii) multi-instance squashes</td>
</tr>
</tbody>
</table>

many times. This is one of the examples used in [50]. Figure 1(b) shows an example where attacker-instigated mispredictions in multiple branches can result in the repeated squash and replay of a transmitter. Different branch structures and different orders of branch resolution result in different replay counts.

The second characteristic in Table 1 is whether the Victim is transient. Transient instructions are speculatively-executed dynamic instructions that will not commit. MRAs can target both transient and non-transient instructions. Transient Victims are more concerning: since the programmer and compiler do not expect their execution, they can leak a wider variety of secrets.

Figure 1(d) shows an example where an MRA can attack a transient instruction through branch misprediction. The transmitter should never execute, but the attacker trains the branch predictor so that it does. Figure 1(e) shows a related example. The transmitter should not execute using the secret, but the attacker trains the branch predictor so that it does.

The third characteristic in Table 1 is whether the Victim is in a loop accessing the same secret in every iteration. If it is, MRAs are more effective for two reasons. First, the attacker has more opportunities to force the re-execution of the transmitter and leak the secret. Second, since the loop is dynamically unrolled in the ROB during execution, the ROB may contain multiple instances of the transmitter, already leaking the secret multiple times. Only when a squash occurs will any MRA defense engage. We call a squash that squashes multiple transmitter instances leaking the same secret in an unrolled loop a multi-instance squash.

Figures 1(e) and (f) are like (c) and (d), but with the transmitter in a loop. In these cases, the attacker can create more leaks of the transmitter by training the branch predictor so these branches mispredict in every iteration. In the worst case, the branch in the first iteration resolves after K loop iterations are loaded into the ROB and have executed. By the time the multi-instance squash occurs, x has been leaked as many as K times. Only then is the MRA defense engaged.

Figure 1(g) is like (f) except that the transmitter leaks a different secret every iteration. In this case, it is easier to minimize the leakage.

3.2 Our Approach to Thwarting MRAs

To see how to thwart MRAs, consider Figure 2(a), where a Squashing instruction $S$ causes the squash of all the younger instructions in the ROB (Victims $V_0 \ldots V_n$). The idea is to detect this event and record all the Victim instructions. Then, as the Victim instructions are re-inserted into the ROB, precede each of them with a fence. We want to prevent the re-execution of each $V_i$ until $V_i$ cannot be squashed anymore. In this way, the attacker cannot observe the side effects of $V_i$ more than once. The point when $V_i$ cannot be squashed anymore is (i) when $V_i$ is at the head of the ROB, or (ii) when no older instruction than $V_i$ in the ROB or any other event (e.g., a memory consistency violation) can squash $V_i$. This point has been called the Visibility Point (VP) of $V_i$ [58].

For highest performance, the type of fence used should be one that only prevents the execution of the $V_i$ instruction, where $V_i$ can be any type of transmitter instruction. Further, when $V_i$ reaches its
We consider supervisor- and user-level attackers. In both cases, we assume the attacker can monitor a microarchitectural side channel (e.g., those in Section 2.1). This is easily realized when the attacker has supervisor-level privileges, as in the original MRA paper for the SGX setting [50]. It is also possible, subject to OS scheduler assumptions, when the attacker runs unprivileged code [25]. In addition, we assume that the attacker can trigger squashes in the victim program to perform MRAs. Which squashes are possible depends on the attacker. In the supervisor-level setting, the attacker can trigger squashes due to exceptions such as page faults, or due to branch mispredictions by priming the branch predictor state. In the user-level setting, the attacker has more limited capabilities. For example, it may be able to trigger branch mispredictions by priming the branch predictor state [35] but cannot cause exceptions.

5 PROPOSED DEFENSE SCHEMES

5.1 Outline of the Schemes

A highly secure defense against MRAs would keep a fine-grain record of all the dynamic instructions that were squashed. When one of these instructions would later attempt to re-execute, the hardware would fence it and, when it reached the VP, remove it from the record. In reality, such a scheme is not practical due to the potentially large storage requirements and the difficulty of identifying the same dynamic instruction. Hence, Jamais Vu proposes three classes of schemes that discard this state early. The schemes differ on when and how they discard the state.

A scheme called Clear-on-Retire discards any Victim information as soon as the program makes forward progress—i.e., when the Squashing instruction reaches its VP (and hence will retire). A scheme called Epoch discards the state when the current “execution locality” or epoch terminates, and execution moves to another one. Finally, a scheme called Counter keeps the state forever, but it compresses it so that all dynamic instances of the same static instruction keep their state merged. Each of these policies to discard or compress state creates a different attack surface.

5.2 Clear-on-Retire Scheme

The rationale for the simple Clear-on-Retire scheme is that an MRA leaks information by stalling a program’s forward progress and repeatedly re-executing the same set of instructions. Hence, when an MRA defense manages to force forward progress, it is appropriate to discard the record of Victim instructions. Therefore, Clear-on-Retire clears the Victim state when the Squashing instruction reaches its VP.

Clear-on-Retire stores information about the Victim instructions in a buffer associated with the ROB called the Squashed Buffer (SB). Figure 2(b) shows a conceptual view of the SB. It is composed of a PC Buffer and an identifier register (ID). The PC Buffer contains the set of program counters (PCs) of the Victim instructions. Since a squash may discard multiple iterations of a loop in the ROB, the SB may contain the same PC multiple times. The ID register contains information that identifies the Squashing instruction—i.e., the one that caused the squash. Such information includes the PC of the instruction and its position in the ROB.

Multiple instructions in the ROB may cause squashes, in any order. For example, in Figure 1(b), the branch in Line 3 may cause a squash first, and then the branch in Line 1 may cause a squash. At every squash, the Victims’ PCs are added to the PC Buffer. However, ID is only updated if the Squashing instruction is older than the one currently in ID. This is because the older instruction will retire first and hence its retirement is needed to make forward progress.
The **Clear-on-Retire** algorithm works as follows. On a squash, the PCs of the Victims are added to the PC Buffer, and ID is updated if necessary. When trying to insert an instruction in the ROB, if it is in the PC Buffer, a fence is placed before it. When the instruction in ID reaches its VP, since the program is making forward progress, the SB is cleared and all the fences introduced by **Clear-on-Retire** are nullified.

To understand why ID needs to store both the Squashing instruction’s PC and its ROB index, note that there are two types of Squashing instructions. One type, such as mispredicted branches, remain in the ROB after they trigger a squash; the other type, such as instructions suffering an exception or loads suffering a memory consistency violation, are removed from the ROB after they trigger a squash. For the first type, **Clear-on-Retire** does not use the PC field in ID; it only uses the ROB index in ID to determine the relative age of any two Squashing instructions. For the second type, since the instruction is removed from the ROB, the ROB index in ID is meaningless. Hence, **Clear-on-Retire** uses the PC in ID to identify the Squashing instruction when it is re-inserted into the ROB. At that point, **Clear-on-Retire** saves into ID the instruction’s new ROB index.

The first row of Table 2 describes **Clear-on-Retire**. The scheme is simple and has the most inexpensive hardware. The SB can be implemented as a simple Bloom filter (Section 6.1).

One shortcoming of **Clear-on-Retire** is that it has some unfavorable security scenarios. Specifically, the attacker could choose to make slow forward progress toward the transmitter in, forcing every single instruction encountered to be a Squashing one.

In practice, this scenario may be hard to set up since, for maximum effectiveness, the squashes have to occur in strict order, from older to younger predecessor of . Indeed, if a Squashing instruction , and is then re-inserted into the ROB with a fence, a second Squashing instruction older than will not squash ’s execution again. The reason is that is fencned and has not yet executed.

### 5.3 Epoch Scheme

The rationale for the **Epoch** scheme is that an MRA attacks an “execution locality” of a program, which has a certain combination of Victim instructions. Once program execution moves to another locality, the re-execution (and squash) of some of the original Victims is not seen as dangerous. Hence, it is appropriate to discard the record of Victim instructions from a locality when moving to another locality. We refer to an execution locality as an **Epoch**. Possible epochs are a loop iteration, a whole loop, or a subroutine.

Like **Clear-on-Retire**, **Epoch** uses an SB to store information about the Victim instructions. However, the design is a bit different. First, **Epoch** requires the hardware to find “start-of-epoch” markers as it inserts instructions into the ROB. We envision that such markers are added by the compiler. Second, the SB needs one (ID, PC-Buffer) pair for each in-progress epoch. The ID now stores a small-sized, monotonically-increasing epoch identifier; the PC Buffer stores the PCs of the Victims squashed in that particular epoch.

The **Epoch** algorithm works as follows. As instructions are inserted into the ROB, the hardware records every start-of-epoch marker. On a squash, the Victim PCs are stored in different PC Buffers depending on the epoch they belong to. The IDs of the PC Buffers are set to the corresponding epoch IDs. Note that a given PC may be in multiple PC Buffers and even multiple times in the same PC Buffer. Then, when trying to insert an instruction in the ROB, if it is in the PC Buffer of the current epoch, is fenced. Finally, when the first instruction of an epoch reaches its VP, the hardware clears the (ID, PC-Buffer) of any older epoch.

When a program re-starts after a squash, the first instruction re-enters the ROB with the same epoch ID as that of the oldest squashed instruction. For example, suppose that instruction of epoch suffers a page fault while younger instructions from epochs and are also in the ROB. The hardware flushes and all subsequent instructions. After the page fault is repaired, re-enters the pipeline as belonging to epoch , not epoch . Effectively, **Epoch** resets the epoch ID to the point of the squash.

**Epoch** protects the scenario where, after the squash, the re-execution exercises the same set of epochs that were executed speculatively before the squash and left Victim instructions in the PC Buffers—although, perhaps, the re-execution executes different instructions than before in such epochs. However, **Epoch** does not target the case when, after the squash, the re-execution exercises a different set of epochs: e.g., when, because of a branch misprediction, a subroutine is now called that was not called before, or a loop that was initially executed is now not executed anymore. In these cases, we consider that the re-execution has moved to different localities and, therefore, **Epoch** does not need to match the new instructions with the older Victims.

### Table 2: Proposed defense schemes against microarchitectural replay attacks.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Removal Policy</th>
<th>Rationale</th>
<th>Pros/Cons</th>
</tr>
</thead>
</table>
| Clear-on-Retire | When the Squashing instruction reaches its visibility point (VP) | The program makes forward progress when the Squashing instruction reaches its VP | + Simple scheme  
+ Most inexpensive hardware  
- Some unfavorable security scenarios |
| Epoch        | When an epoch completes                             | An epoch captures an execution locality         | + Inexpensive hardware  
+ High security if epoch chosen well  
- Need compiler support |
| Counter      | No removal, but information is compacted            | Keeping the difference between squashes and retirements low minimizes leakage beyond natural program leakage | + Conceptually simple  
- Intrusive hardware  
- May require OS changes  
- Some pathological patterns |

The **Clear-on-Retire** algorithm works as follows. On a squash, the PCs of the Victims are added to the PC Buffer, and ID is updated if necessary. When trying to insert an instruction in the ROB, if it is in the PC Buffer, a fence is placed before it. When the instruction in ID reaches its VP, since the program is making forward progress, the SB is cleared and all the fences introduced by **Clear-on-Retire** are nullified.

The first row of Table 2 describes **Clear-on-Retire**. The scheme is simple and has the most inexpensive hardware. The SB can be implemented as a simple Bloom filter (Section 6.1).

One shortcoming of **Clear-on-Retire** is that it has some unfavorable security scenarios. Specifically, the attacker could choose to make slow forward progress toward the transmitter in, forcing every single instruction encountered to be a Squashing one.

In practice, this scenario may be hard to set up since, for maximum effectiveness, the squashes have to occur in strict order, from older to younger predecessor of . Indeed, if a Squashing instruction , and is then re-inserted into the ROB with a fence, a second Squashing instruction older than will not squash ’s execution again. The reason is that is fencned and has not yet executed.

### 5.3 Epoch Scheme

The rationale for the **Epoch** scheme is that an MRA attacks an “execution locality” of a program, which has a certain combination of Victim instructions. Once program execution moves to another locality, the re-execution (and squash) of some of the original Victims is not seen as dangerous. Hence, it is appropriate to discard the record of Victim instructions from a locality when moving to another locality. We refer to an execution locality as an **Epoch**. Possible epochs are a loop iteration, a whole loop, or a subroutine.

Like **Clear-on-Retire**, **Epoch** uses an SB to store information about the Victim instructions. However, the design is a bit different. First, **Epoch** requires the hardware to find “start-of-epoch” markers as it inserts instructions into the ROB. We envision that such markers are added by the compiler. Second, the SB needs one (ID, PC-Buffer) pair for each in-progress epoch. The ID now stores a small-sized, monotonically-increasing epoch identifier; the PC Buffer stores the PCs of the Victims squashed in that particular epoch.

The **Epoch** algorithm works as follows. As instructions are inserted into the ROB, the hardware records every start-of-epoch marker. On a squash, the Victim PCs are stored in different PC Buffers depending on the epoch they belong to. The IDs of the PC Buffers are set to the corresponding epoch IDs. Note that a given PC may be in multiple PC Buffers and even multiple times in the same PC Buffer. Then, when trying to insert an instruction in the ROB, if it is in the PC Buffer of the current epoch, is fenced. Finally, when the first instruction of an epoch reaches its VP, the hardware clears the (ID, PC-Buffer) of any older epoch.

When a program re-starts after a squash, the first instruction re-enters the ROB with the same epoch ID as that of the oldest squashed instruction. For example, suppose that instruction of epoch suffers a page fault while younger instructions from epochs and are also in the ROB. The hardware flushes and all subsequent instructions. After the page fault is repaired, re-enters the pipeline as belonging to epoch , not epoch . Effectively, **Epoch** resets the epoch ID to the point of the squash.

**Epoch** protects the scenario where, after the squash, the re-execution exercises the same set of epochs that were executed speculatively before the squash and left Victim instructions in the PC Buffers—although, perhaps, the re-execution executes different instructions than before in such epochs. However, **Epoch** does not target the case when, after the squash, the re-execution exercises a different set of epochs: e.g., when, because of a branch misprediction, a subroutine is now called that was not called before, or a loop that was initially executed is now not executed anymore. In these cases, we consider that the re-execution has moved to different localities and, therefore, **Epoch** does not need to match the new instructions with the older Victims.
The second row of Table 2 describes Epoch. The scheme is also simple and has inexpensive hardware. It can also implement the PC Buffers as Bloom filters. Epoch has high security if epochs are chosen appropriately, as the Victim information remains for the whole duration of the epoch. A drawback of Epoch is that it needs compiler support.

An epoch can be long, in which case its PC Buffer may contain too many PCs to operate efficiently. Hence, our preferred implementation of this scheme is a variation of Epoch called Epoch-Rem that admits PC removal. Specifically, when a Victim from an epoch reaches its VP, the hardware removes its PC from the corresponding PC Buffer. This support is implemented by employing the PC Buffers as counting Bloom filters (Section 6.2).

5.4 Counter Scheme

The Counter scheme never discards information about Victim squashes. However, to be implementable, the scheme merges the squash information from all the dynamic instances of the same static instruction into a single variable. Specifically, Counter records, for any given static instruction, the difference between the number of times it has been squashed and the number of times it has retired. Counter’s goal is to keep such difference small. The rationale is that, if both counts are similar, the MRA is unlikely to exfiltrate much more information than what the program naturally leaks.

While Counter can be implemented like the two previous schemes, a more intuitive implementation associates Victim information with each static instruction. A simple design adds a Squashed bit to each static instruction I. When I gets squashed, its Squashed bit is set. From then on, an attempt to insert I in the ROB causes a fence to be placed before I. When I reaches its VP, the bit is reset. After that, a future invocation of I is allowed to execute with no fence.

In reality, multiple dynamic instances of the same static instruction may be in the ROB at the same time and get squashed together. Hence, we use a Squashed Counter per static instruction rather than a bit. The algorithm works as follows. Every time that dynamic instances of the instruction get squashed, the counter increases by the number of squashed instances; every time that an instance reaches its VP, the counter is decremented by one. The counter does not go below zero. Finally, when an instruction is inserted in the ROB, if its counter is not zero, the hardware fences it. This is the Counter scheme that we propose.

To reduce the number of stalls, a variation of this scheme allows a Victim to execute without a fence as long as its counter is lower than a threshold.

The third row of Table 2 describes Counter. The scheme is conceptually simple. However, it requires somewhat intrusive hardware. One possible design requires counters that are stored in memory and get cached on demand into a special cache next to the L1 (Section 6.3). This counter cache or the memory needs to be updated every time a counter changes. In addition, the OS needs changes to allocate and manage pages of counters for the instructions.

Counter has some pathological patterns. Specifically, an attacker may be able to repeatedly squash an instruction by interleaving the squashes with retirements of the same static instruction. In this case, one access leaks a secret before being squashed, while the other access is benign, retires, and decreases the counter. This pattern is shown in Figure 1(e). In every iteration, the branch predictor incorrectly predicts the condition to be true, x is set to secret, and the transmitter leaks x. The execution is immediately squashed, the else code executes, and the transmitter retires. This process is repeated in every iteration, causing the counter to toggle between one and zero.

5.5 Analysis of the Security of the Schemes

To assess the relative security of the schemes, we compare their worst-case leakage for each of the code snippets in Figure 1. While the snippets in Figure 1 only show some of the possible patterns, they cover a broad spectrum of cases. Indeed, they show examples of transmitters in straight-line code and in loops; replays due to exceptions (Figure 1(a)) and branch mispredictions; transmitters executed transiently (e.g., Figure 1(d)) and non-transiently; and transmitters with iteration-independent and iteration-dependent secrets.

A summary of the analysis is shown in Table 3. We measure leakage as the number of executions of the transmitter for a given secret. We report Transient Leakage (TL) when the transmitter is a transient instruction and Non-Transient Leakage (NTL) when it is not. For the Epoch scheme, we show the leakage for one design that uses iterations as epochs (Iter) and for one that uses loops (Loop). For each of these two designs, we consider an implementation without removal of Victim PCs from the PC Buffers when they reach their VP (NR) and with removal of them (R).

Table 3: Worst-case leakage count in the proposed defense schemes for some of the examples in Figure 1. For a loop, N is the number of iterations and, as the loop dynamically unrolls in the ROB, K is the number of iterations that fit in the ROB.

<table>
<thead>
<tr>
<th>Case</th>
<th>Non-Trans!</th>
<th>Transient Leakage (TL)</th>
<th>Clear-on-Retire</th>
<th>Epoch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NTL</td>
<td>Iter</td>
<td>Loop</td>
<td></td>
</tr>
<tr>
<td>(a)</td>
<td>1</td>
<td>ROB-1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>1</td>
<td>BRROB-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c), (d)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(e)</td>
<td>0</td>
<td>K*N</td>
<td>N</td>
<td>K</td>
</tr>
<tr>
<td>(f)</td>
<td>0</td>
<td>K*N</td>
<td>N</td>
<td>K</td>
</tr>
<tr>
<td>(g)</td>
<td>0</td>
<td>K</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

In Figure 1(a), since the transmitter should commit, the NTL is one. The TL is found as follows. In Clear-on-Retire, the attacker could make each instruction older than the transmitter a Squashing one. In the very worst case, the squashes occur in program order, and the timing is such that the transmitter is squashed as many times as the ROB size minus one. Hence TL is ROB size minus 1. While this is a large number, it is smaller than the leakage in the original MicroScope attack [50], where TL is infinite because one instruction can cause any number of squashes. In all Epoch designs, the transmitter is squashed only once. Hence, TL is 1. Counter
sets the transmitter’s counter to 1 on the first squash; no other speculative re-execution is allowed. Hence, TL is 1.

Figure 1(b) is conceptually like (a). The NTL in all schemes is 1. The TL of Counter and Epoch is 1. In Clear-on-Retire, in the worst-case where all the branches are mispredicted and resolve in program order, the TL is equal to the number of branches that fit in the ROB minus one slot (for the transmitter).

Figures 1(c) and (d) are very simple examples. NTL is 0 (since in Figure 1(c) x is never set to the secret in a non-speculative execution) and TL is 1 for all schemes.

In Figure 1(e), NTL is zero. However, the attacker may cause the branch to be mispredicted in every iteration. To assess the worst-case TL in Clear-on-Retire, assume that, as the N-iteration loop dynamically unrolls in the ROB, K iterations fit in the ROB. In this case, the worst-case is that each iteration (beyond the first K − 1 ones) is squashed K times. Hence, TL in Clear-on-Retire is K + N. In Epoch with iteration, since each epoch allows one squash, the TL is N (with and without PC removal). In Epoch with loop without removal, in the worst case, the initial K iterations are in the ROB when the squash occurs, and we have a multi-instance squash (Section 3.1). Hence, the TL is K. In Epoch with loop with removal, since every retirement of the transmitter removes the transmitter PC from the SB, TL is N. Finally, in Counter, since every iteration triggers a squash and then a retirement, TL is N.

Figure 1(f) is like Figure 1(e), except that the transmit instruction never retires for any value of x. As a consequence, Epoch with loop with removal does not remove it from the SB, and Counter does not decrease the counter. Hence, their TL is K.

Finally, Figure 1(g) is like 1(f) except that each iteration accesses a different secret. The NTL is zero. The TL for Clear-on-Retire is K because of the dynamic unrolling of iterations in the ROB. For the other schemes, TL is 1 in the worst case.

Overall, for the examples shown in Table 3, Epoch at the right granularity (i.e., loop level) without removal has the lowest leakage. With removal, the scheme is similar to Counter, and better than Epoch with iteration. Clear-on-Retire has the highest worse-case leakage. Further analysis with more code patterns is part of our future work, and will provide more insights.

Appendix B analyzes the implications of the leakage bounds in Table 3 on the security of a system.

6 MICROARCHITECTURAL DESIGN

6.1 Implementing Clear-on-Retire

The PC Buffer in the SB needs to support three operations. First, on a squash, the PCs of all the Victims are inserted in the PC Buffer. Second, before an instruction is inserted in the ROB, the PC Buffer is queried to see if it contains the instruction’s PC. Third, when the instruction in the ID reaches its VP, the PC Buffer is cleared.

These operations are easily supported with a hardware Bloom filter [8]. Figure 3 shows the filter’s structure. It is an array of M entries, each with a single bit. To insert an item in the filter (BF), the instruction’s PC is hashed with n hash functions (H_i) and n bits get set: BF[H_1], BF[H_2], ..., BF[H_n]. The filter can be implemented as an n-port direct-mapped cache of M 1-bit entries.

A Bloom filter can have false positives but no false negatives. A false positive occurs when a PC is not in the PC Buffer but it is deemed to be there due to a conflict. This situation is safe, as it means that Clear-on-Retire will end up putting a fence before an instruction that does not need it.

In practice, if we size the filter appropriately, we do not see many false positives when running benign programs. Specifically, as we will see in Section 9.3, for a 192-entry ROB, a filter with 1232 bits and 7 hash functions has less than 0.5% false positives.

6.2 Implementing Epoch

The SB for Epoch is like the one for Clear-on-Retire with two differences. First, there are multiple (ID, PC-Buffer) pairs—one for each in-progress epoch. Second, in Epoch-Rem, which supports the removal of individual PCs from a PC Buffer, each PC Buffer is a counting Bloom filter [18].

Figure 4 shows the SB with multiple counting Bloom filters. The latter are like plain filters except that each entry has k bits. To insert an item in a filter, the n entries selected by the hashes are incremented by one—i.e., BF[H_1]++, BF[H_2]++, ..., BF[H_n]++. To remove the item, the same entries are decremented by one. An n-port direct-mapped cache of M k-bit entries is used.

Figure 3: SB with a PC Buffer organized as a Bloom filter.

Figure 4: SB with multiple PC Buffers organized as counting Bloom filters.
False negatives reduce security because no fence is inserted where there should be one. However, by appropriately sizing the Bloom filter relative to the ROB size, we can reduce the upper bound of false negatives [26]. In practice, as we will see in Section 9.3, because each counting Bloom filter only contains Victims from one epoch, we find that only 0.02% and 0.006% of the accesses are false negatives in Epoch with loops and iterations, respectively.

Note that an attacker cannot explicitly cause hashed addresses to bunch-up into a few Bloom-filter entries and saturate them. The reason is that the attacker does not control how the Victim instructions follow a squash scatter into the Bloom filter.

6.2.1 Handling Epoch Overflow. The SB has a limited number of [ID, PC-Buffer] pairs. Therefore, it is possible that, on a squash, the Victim instructions belong to more epochs than PC Buffers exist in the SB. For this reason, Epoch augments the SB with one extra ID not associated with any PC Buffer called Over/f_lowID. To understand how it works, recall that epoch IDs are monotonically increasing. Hence, we may find that Victims from a set of high-numbered epochs have no PC Buffer to go. In this case, we store the ID of the highest-numbered epoch of any Victim in Over/f_lowID. From now on, when a new instruction is inserted in the ROB, if it belongs to an epoch whose ID: (i) owns no PC Buffer and (ii) is no higher than the one in Over/f_lowID, we place a fence before the instruction. The reason is that, since we have lost information about Victims in that epoch, we do not know whether the instruction is a Victim. When the epoch whose ID is in Over/f_lowID is fully retired, Over/f_lowID is cleared.

As an example, consider Figure 5(a), which shows a ROB full of instructions. The figure groups the instructions according to their epoch and labels the group with the epoch ID. Assume that all of these instructions are squashed and that the SB only has four [ID, PC-Buffer] pairs. Figure 5(b) shows the resulting assignment of epochs to [ID, PC-Buffer] pairs. Epochs 14 and 15 overflow and, therefore, Over/f_lowID is set to 15. Any future insertion in the ROB of an instruction from epochs 14 and 15 will be preceded by a fence. Eventually, some [ID, PC-Buffer] pairs will free-up and may be used by newer epochs such as Epoch 16. However, all instructions from Epochs 14 and 15 will always be fenced.

![Figure 5: Handling epoch overflow.](image)

6.3 Implementing Counter

To implement Counter, Jamais Vu stores the counters for all the instructions in data pages, and the core has a small Counter Cache (CC) that keeps the recently-used counters close to the pipeline for easy access. Since the most frequently-executed instructions are in loops, a small CC typically captures the majority of the counters needed.

We propose a simple design where, for each page of code, there is an associated data page at a fixed Virtual Address (VA) Offset that holds the counters of the instructions in the page of code. Further, the VA offset between each instruction and its counter is fixed, to ease access. In effect, this design increases the memory consumed by a program by the size of its instruction page working set.

Figure 6(a) shows a page of code and its page of counters at a fixed VA offset. When the former is brought into physical memory, the latter is also brought in. The figure shows a memory line with several instructions and the associated line with their counters. We envision each counter to be 4 bits.

![Figure 6: Allocating and caching instruction counters.](image)
6.4 Handling Context Switches

To operate correctly, Jamais Vu performs the following actions at context switches. In Clear-on-Retire and Epoch, the SB state is saved to and restored from memory as part of the context. This enables the defense to remember the state when execution resumes. In Counter, the CC is flushed to memory to leave no traces behind that could potentially lead to a side-channel exploitable by the newly scheduled process. The new process loads the CC on demand. These operations can be done safely by the trusted environment.

7 COMPILER PASS

Epoch includes a program analysis pass that places “start-of-epoch” markers in the program. The pass accepts as input a program in source code or binary. Source code is preferred, since it contains more information and allows a better analysis.

We consider two designs: one that uses loops as epochs and one that uses loop iterations as epochs. In the former, an epoch includes the instructions between the beginning and the end of a loop, or between the end of a loop and the beginning of the next loop; in the latter, an epoch includes the instructions between the beginning and the end of an iteration, or between the end of the last iteration in a loop and the beginning of the first iteration in the next loop. In both Epoch designs, procedure calls and returns are also epoch boundaries.

The analysis is intra-procedural and uses conventional control flow compiler techniques [3]. It searches for back edges in the control flow of each function, and from there identifies the natural loops. Once back edges and loops are identified, the Epoch compiler inserts the epoch boundary markers.

To mark an x86 program, our analysis pass places a previously-ignored instruction prefix [29] in front of every first instruction of an epoch. The processor ignores this prefix, and our simulator recognizes that a new epoch starts. This approach changes the executable, but because current processors ignore this prefix, the new executable runs on any x86 machine. The size of the executable increases by only 1 byte for every static epoch. For epoch boundaries formed by procedure calls and returns, the compiler does not need to mark anything. The simulator recognizes the x86 procedure call and return instructions and starts a new epoch.

8 EXPERIMENTAL SETUP

Architectures Modeled. We model the architecture shown in Table 4 using cycle-level simulations with gem5 [7]. The baseline architecture is called Unsafe, because it has no protection against MRAs. The defense schemes are: (i) Clear-on-Retire (CoR), (ii) Epoch with iteration (Epoch-ITER), (iii) Epoch-Rem with iteration (Epoch-ITER-REM), (iv) Epoch with loop (Epoch-LOOP), (v) Epoch-Rem with loop (Epoch-LOOP-REM), and (vi) Counter (Counter).

From Table 4, we can compute the sizes of the Jamais Vu hardware structures. Clear-on-Retire uses 1 non-counting Bloom filter. The size is 1232 bits. Epoch uses 12 Bloom filters. For Epoch-Rem, since the counting Bloom filters use 4 bits per entry, the total size is 12 times 4,928 bits, or slightly above 7KB. A Bloom filter has 14 read and 7 write ports. The Counter Cache (CC) in Counter contains 128 entries, each with the counters of one I-cache line. Since the shortest x86 instruction is 1 byte and a counter is 4 bits, each line in the CC is shifted 4 bits every byte, compacting the line into 32B. Hence, the CC size is 4KB.

Table 4: Parameters of the simulated architecture.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>2.0 GHz out-of-order x86 core</td>
</tr>
<tr>
<td>Core</td>
<td>8-issue, no SMT, 62 load queue entries, 32 store queue entries, 192 ROB entries, L-TAGE branch predictor, 4096 BTB entries, 16 RAS entries</td>
</tr>
<tr>
<td>L1-I Cache</td>
<td>32 KB, 64 B line, 4-way, 2 cycle Round Trip (RT) latency, 1 port, 1 hardware prefetcher</td>
</tr>
<tr>
<td>L1-D Cache</td>
<td>64 KB, 64 B line, 8-way, 2 cycle RT latency, 3 Rd/Wr ports, 1 hardware prefetcher</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>2 MB, 64 B line, 16-way, 8 cycles RT latency</td>
</tr>
<tr>
<td>DRAM</td>
<td>50 ns RT latency after L2</td>
</tr>
<tr>
<td>Counter Cache</td>
<td>32 sets, 4-way, 2 cycle RT latency, 4b/counter</td>
</tr>
<tr>
<td>Bloom Filter</td>
<td>1232 entries, 7 hash functions. Non-counting: 4b/entry, Counting: 4b/entry</td>
</tr>
<tr>
<td>[ID, PC-Buffer]</td>
<td>12 pairs in Epoch, 1 pair in Clear-on-Retire</td>
</tr>
</tbody>
</table>

9 EVALUATION

9.1 Thwarting Proof-of-Concept (PoC) MRA

To demonstrate Jamais Vu’s ability to thwart MRAs, we implement a PoC MRA on gem5 similar to the port contention attack in [50]. After testing a secret, the victim thread performs a division operation and returns the result. The attacker picks 10 Squashing instructions that precede the test and the division. The code is similar to Figure 1(a). In Unsafe, the attacker causes 5 squashes on each of the 10 Squashing instructions, for a total of 50 replays of the division operation. With Clear-on-Retire, the number of replays decreases to 10, since each Squashing instruction can only cause a single replay. With Epoch, there is a single replay because the division only commits once.

9.2 Execution Time

Jamais Vu proposes several schemes that offer different performance, security, and implementation complexity trade-offs. Figure 7 shows the normalized execution time of SPEC17 applications on all schemes but Epoch without removals, which we consider later. Time is normalized to Unsafe.

Among all the schemes, CoR has the lowest execution time overhead. It incurs only a geometric mean overhead of 2.9% over Unsafe. It is also the simplest but least secure design (Table 3). Epoch-ITER-REM has the next lowest average execution overhead, namely 11.0%. This design is also very simple and is more secure,
especially as we will see that false negatives are very rare. The next design, Epoch-Loop-Rem, has higher average execution time overhead, namely 13.8%. However, it has simple hardware and is one of the two most secure designs (Table 3)—again, given that, as we will see, false negatives are very rare. Finally, COUNTER has the highest average execution overhead, namely 23.1%. It is one of the two most secure schemes, but the implementation proposed is not as simple as the other schemes. From all these schemes, Epoch-Loop-Rem and perhaps CoR appear to be the most appealing.

The schemes not shown in the figure, namely Epoch-Iter and Epoch-Loop are not competitive. They have an average execution overhead of 22.6% and 63.8%, respectively. These are substantial increases over the schemes with removals, with modest gains in simplicity and security.

9.3 Sensitivity Study
Each Jamais Vu scheme has several architectural parameters that set its hardware requirements and efficiency. Recall that CoR uses a Bloom filter, while Epoch-Iter-Rem and Epoch-Loop-Rem use counting Bloom filters. To better understand the different Bloom filters, we first perform a sensitivity study of their parameters. Then, we evaluate several Counter Cache organizations for COUNTER.

Number of Bloom Filter Entries. Figure 8 shows the geometric mean of the normalized execution time and the false positive rates (FP) on SPEC17, when varying the size of the Bloom filter. We consider several sizes, which we measure in number of entries. Recall that each entry is 1 bit for CoR and 4 bits for the other schemes. We pick each of these number of entries by first selecting a projected element count (i.e., the number of items that we expect to be inserted in the Bloom filter, as shown in parenthesis in the figure) and running an optimization pass [44] for a target false positive probability of 0.01. From the figure, we see that a Bloom filter of 1232 entries strikes a good balance between execution and area overhead, with a false positive rate of less than 0.5% for all the schemes.

Number of [ID, PC-Buffer] Pairs. Another design decision for Epoch-Iter-Rem and Epoch-Loop-Rem is how many [ID, PC-Buffer] pairs to have. If they are too few, overflow will be common. Figure 9 shows the average normalized execution time and the overflow rates on SPEC17, when varying the number of [ID, PC-Buffer] pairs. The overflow rate is the fraction of insertions into PC Buffers that overflow. From the figure, we see that, as the number of [ID, PC-Buffer] pairs decreases, the execution time and overflow rates increase. Supporting 12 [ID, PC-Buffer] pairs is a good design point.

Number of Bits Per Counting Bloom Filter Entry. The counting Bloom filters in Epoch-Iter-Rem and Epoch-Loop-Rem use a few bits per entry to keep the count. Figure 10 shows the average normalized execution time and the false negative rates (FN) on SPEC17, when varying the number of bits per entry. We see
from the figure that the number of bits per entry has little impact on the performance. However, as the number of bits per entry decreases beyond four, the false negative rate increases rapidly. For four bits per entry, the false negative rate is an acceptable 0.02% for Epoch-Loop-Rem and 0.006% for Epoch-Iter-Rem.

Figure 10: Average normalized execution time and false negative rate (FN) when varying the number of bits per counting Bloom filter entry.

False negatives can be caused either by conflicts in the filter or by not having enough bits in an entry. In the latter case, when the counter in the entry saturates, it cannot record further squashes and information is lost. To estimate the relative impact of these two sources of false negatives, we took our default Bloom filter of 1232 entries and four bits per entry, and artificially eliminated conflicts. We did this by recording the inserted items in an ideal hash table that has no conflicts. We found that the resulting false negative rates are 0.004% and 0.002% for Epoch-Loop-Rem and Epoch-Iter-Rem, respectively. These numbers are comparable to the false negative rates obtained by taking the default Bloom filter and simply adding one extra bit per entry.

Counter Cache (CC) Geometry. Figure 11 shows the CC hit rate as we vary the ways and sets of the CC. We see that the CC hit rate increases with the number of entries, but that changing the associativity of the CC from 4 to full does not help. Overall, our default configuration of 32 sets and 4 ways performs well. It attains an average hit rate of 93.7%, while a larger cache or a fully-associative configuration of 32 sets and 4 ways performs well. It attains an average execution time overhead of only 2.9%, although it is less secure.

Figure 11: CC hit rate when varying the cache geometry.

10 RELATED WORK

There are some works related to mitigating MRAs.

Preventing Pipeline Squashes. The literature includes several solutions that can mitigate specific instances of MRAs. For example, page fault protection schemes [11, 41, 42, 49] can be used to mitigate MRAs that rely on page faults to cause pipeline squashes. The goal of these countermeasures is to block controlled-channel attacks [55, 57] by terminating victim execution when an OS-induced page fault is detected. The most recent of these defenses, Autarky [42], achieves this through a hardware/software co-design that delegates paging decisions to the enclave. However, attacks that rely on events other than page faults to trigger pipeline squashes (Section 3.1) would still overcome these point-mitigation strategies. In contrast, Jamais Vu is the first comprehensive defense that addresses the root cause of MRAs, namely that instructions can be forced to execute more than once.

Preventing Side Channel Leakage. Another strategy to mitigate MRAs is to prevent speculative instructions from leaking data through side channels. For example, several works have proposed to mitigate side channels by isolating or partitioning microarchitectural resources [10, 12, 13, 23, 34, 37, 41, 52, 63], thus preventing the attacker from accessing them during the victim process’ execution. These defenses prevent adversaries from leaking data through specific side channels, which ultimately makes MRAs’ ability to denoise these channels less useful. In practice, however, no holistic solution exists that can block all side channels. Further, new adversarial applications of MRAs may be discovered that go beyond denoising side-channel attacks.

11 CONCLUSION

This paper presented Jamais Vu, the first technique to thwart MRAs. Jamais Vu detects when an instruction is squashed and, as it is re-inserted into the pipeline, places a fence before it. The three main Jamais Vu designs are Clear-on-Retire, Epoch, and Counter, which offer different trade-offs between security, execution overhead, and implementation complexity. One design, called Epoch-Loop-Rem, effectively mitigates MRAs, has an average execution time overhead of 13.8% in benign executions, and only needs counting Bloom filters. An even simpler design, called Clear-on-Retire, has an average execution time overhead of only 2.9%, although it is less secure.

ACKNOWLEDGMENTS

This work was partially funded by NSF grants CNS 1763658 and 1942888, Semiconductor Research Corporation (SRC) contract no. 2020-HW-2995, and a Google Faculty Research Award.

A A MEMORY CONSISTENCY MODEL VIOLATION MRA

A.1 Description of the Attack

In Section 3.1, we have argued that MRAs can rely on various sources to trigger pipeline squashes repeatedly. The paper that introduced MRAs [50] used OS-induced page faults to cause replays. In this appendix, we show for the first time that memory consistency model violations can also create MRAs. In these MRAs, a load issued speculatively to a location x is squashed because either the cache receives an invalidation for the line where x resides, or the line where x resides is evicted from the cache [15, 48]. A user-level attacker can force either event.
Our proof-of-concept (PoC) experiment consists of a victim and an attacker running on two sibling CPU threads and sharing a cache line A. In reality, the PoC variant using cache evictions could be carried out without victim and attacker sharing memory and, instead, using other eviction techniques [54].

The PoC is shown in Figure 12. The victim first brings shared cache line A into the L1 cache and evicts private cache line B from the cache. The victim then reads line B and misses in the entire cache hierarchy. While B is being loaded from memory, the victim speculatively loads shared cache line A, followed by other speculative instructions. The attacker’s goal is to evict A or to write to A after it has been speculatively loaded by the victim into the cache but before load B completes. If this is accomplished, the load(A) instruction will be squashed, together with the subsequent instructions due to a violation of the memory model.

```plaintext
for i in 1..N
  LFENCE
  LOAD(A) // Bring A to the cache
  CLFLUSH(B) // Evict B from the cache
  LFENCE
  LOAD(B) // LOAD(B) misses in the cache
  LOAD(A) // LOAD(A) hits in the cache and then
  // is evicted/invalidated by attacker
  ADD ... // 40 unrelated add instructions
(a) Victim.

while(1)
  CLFLUSH(A) or STORE(A) // Evict/Invalidate A
  .REPT 100 // Do nothing for a small interval
  NOP // by executing 100 nops
  .ENDR
(b) Attacker.
```

Figure 12: Pseudocode of our proof-of-concept victim and attacker causing pipeline squashes due to memory consistency model violations.

### A.2 Experimental Evaluation

We run our experiment on a 4.00 GHz quad-core Intel i7-6700K CPU. We configure the victim to run in a loop and set up the attacker to evict or invalidate the shared cache line A periodically. If, during a victim loop iteration, the attacker’s eviction or invalidation occurs after the victim has speculatively loaded A but before instruction load(B) has retired, the victim will incur a pipeline squash.

To detect if the victim incurs any pipeline squashes, we read the number of `machine clears` (Intel’s terminology for pipeline squashes), micro-ops issued, and micro-ops retired from the hardware performance counters [30] (using Intel VTune [31] to monitor only the victim’s function of interest).

We compare these numbers under three scenarios: (1) there is no attacker; (2) the attacker evicts line A; (3) the attacker writes to line A. Table 5 reports the results of our experiment, with a victim configured with 10 million loop iterations. When there is no attacker, we measure zero pipeline squashes in the victim and all the issued micro-ops retire successfully. When the attacker evicts line A, more than 3 million pipeline squashes occur in the victim, and 30% of the issued micro-ops never retire. Finally, when the attacker writes to A, more than 5 million pipeline squashes occur in the victim and 53% of the issued micro-ops never retire.

These results confirm that memory consistency violations can be used as a source of pipeline squashes and replays.

### Table 5: Results of experiment. The numbers are collected over 10 million victim loop iterations.

<table>
<thead>
<tr>
<th>Number of squashes</th>
<th>Percentage of micro-ops</th>
<th>Percentage of micro-ops issued that did not retire</th>
</tr>
</thead>
<tbody>
<tr>
<td>No attacker</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Attacker evicting A</td>
<td>3,221,778</td>
<td>30%</td>
</tr>
<tr>
<td>Attacker writing to A</td>
<td>5,677,938</td>
<td>53%</td>
</tr>
</tbody>
</table>

### B SECURITY ANALYSIS

This appendix analyzes the implications of the leakage bounds in Table 3 on the security of a system. We consider the MRA prototype by MicroScope [50], where a victim program performs two multiplications or two divisions based on a test on a secret value. The attacker forces the victim to continuously replay the operations, while a monitor thread keeps performing division operations, and recording what fraction of the divisions take longer than a certain threshold latency. The authors found that, if the victim is performing divisions, the monitor sees 64 divisions with over-the-threshold latency in 10000 samples; if the victim is performing multiplications, the monitor sees 4 divisions with over-the-threshold latency in 10000 samples.

Based on this prototype, we model an MRA environment as follows. The attacker observes X operations with over-the-threshold latency in N samples. X follows a binomial distribution. When the secret is 0, the probability of observing an over-the-threshold operation is P₀, thus \( X \sim Bin(N, P₀) \). When the secret is 1, the probability is \( P₁ \), thus \( X \sim Bin(N, P₁) \). Based on the MicroScope prototype, we use \( P₀ = 4/10000 \) and \( P₁ = 64/10000 \).

During an attack, the attacker can have two hypotheses:

1. \( H₀ \): the secret is 0, i.e., \( X \sim Bin(N, P₀) \).
2. \( H₁ \): the secret is 1, i.e., \( X \sim Bin(N, P₁) \).

To test which one of \( H₀ \) and \( H₁ \) to accept, the attacker runs the Uniformly Most Powerful (UMP) test [36] with a single cut-off \( C \). If the attacker measures \( X/N < C \), she accepts \( H₀ \) and predicts that the secret is 0; if the attacker measures \( X/N > C \), she accepts \( H₁ \) and predicts that the secret is 1. There are four possible outcomes:

- True secret s is 0:
  1. The attacker correctly predicts 0 with a probability \( P(\text{correct}|s = 0) \).
  2. The attacker incorrectly predicts 1 with a probability \( P(\text{incorrect}|s = 0) \).
- True secret s is 1:
Table 6: The probability of each test outcome.

<table>
<thead>
<tr>
<th>Truth</th>
<th>Prediction</th>
<th>secret = 0</th>
<th>secret = 1</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>secret = 0</td>
<td>$P(\text{correct}</td>
<td>s = 0) = \sum_{x/N &lt; C} \left( \frac{N}{x} \right) P_0^x (1 - P_0)^{N-x}$</td>
<td>$P(\text{incorrect}</td>
<td>s = 0) = \sum_{x/N &gt; C} \left( \frac{N}{x} \right) P_0^x (1 - P_0)^{N-x}$</td>
</tr>
<tr>
<td>secret = 1</td>
<td>$P(\text{incorrect}</td>
<td>s = 1) = \sum_{x/N &lt; C} \left( \frac{N}{x} \right) P_1^x (1 - P_1)^{N-x}$</td>
<td>$P(\text{correct}</td>
<td>s = 1) = \sum_{x/N &gt; C} \left( \frac{N}{x} \right) P_1^x (1 - P_1)^{N-x}$</td>
</tr>
</tbody>
</table>

(3) The attacker correctly predicts 1 with a probability $P(\text{correct}|s = 1)$.

(4) The attacker incorrectly predicts 0 with a probability $P(\text{incorrect}|s = 1)$.

Among the four possible outcomes, the first and third cases result in a correct prediction, while the second and fourth cases result in an incorrect prediction. Table 6 shows the probability of each outcome.

To determine an optimal cut-off $C$, we calculate the likelihood ratio and require it to be 1:

$$L(H_0) = \frac{1}{L(H_1)} = \frac{C \left( \frac{N}{C} \right) P_0^C (1 - P_0)^{N-C}}{C \left( \frac{N}{C} \right) P_1^C (1 - P_1)^{N-C}} = 1$$

After canceling the common parts of the numerator and denominator:

$$\left[ \frac{P_0(1 - P_1)}{P_1(1 - P_0)} \right]^C \frac{(1 - P_0)}{(1 - P_1)} = 1$$

then applying $\ln$ to both sides:

$$C \ln \left[ \frac{P_0(1 - P_1)}{P_1(1 - P_0)} \right] + N \ln \left( \frac{1 - P_0}{1 - P_1} \right) = 0$$

finally:

$$C = -\frac{N \ln \left( \frac{1 - P_0}{1 - P_1} \right)}{\ln \left( \frac{P_0(1 - P_1)}{P_1(1 - P_0)} \right)}$$

Using the values of $P_0 = 4/10000$ and $P_1 = 64/10000$ from the MicroScope experiment, we obtain $C = 21.67N/10000$. This is an optimal choice for the cut-off.

If the attacker wants to exfiltrate the secret bit with more than 80% success rate, each of the probabilities of correct outcomes, namely $P(\text{correct}|s = 0)$ and $P(\text{correct}|s = 1)$, need to be greater than 80%. By solving the equations of $P(\text{correct}|s = 0) > 0.8$ and $P(\text{correct}|s = 1) > 0.8$ in Table 6 for $C = 21.67N/10000$, we find that $N$ needs to be $N = 251$. This means that the attacker needs at least 251 replays to extract a single bit with 80% success rate. If the attacker wants to exfiltrate an entire byte with 80% success rate, then she needs $\approx 97.2\%$ success rate on extracting every single bit. In our case, this means that she requires at least 1017 replays for each bit extraction and 8856 replays in total. The longer the secret is, the more the replays required.

These replay counts are higher than the very worst leakage counts of the Jamais Vu schemes in Table 3. It is true that, in the cases of loops (Rows e and f in the table), the number of iterations $N$ of the loop may be large. However, these leakage counts require that all the loop iterations read from the same location, which is very rare given loop-invariant code-motion compiler optimizations.

Furthermore, the values of aforementioned probabilities $P_0$ and $P_1$ from MicroScope [50] were obtained by re-executing the same set of instructions with the same replay handle. Jamais Vu, instead, forces the attacker to continuously change replay handle. Hence, the attack’s success rate will be even smaller.

Overall, from this estimation, we conclude that the leakage bounds provided by our proposed Jamais Vu schemes make the schemes reasonably secure. Without Jamais Vu, the attacker can extract a secret that has an arbitrary length with 100% success rate [50].

C ARTIFACTS

C.1 Abstract

Our artifact provides a complete gem5 implementation of Jamais Vu, along with scripts to evaluate the SPEC'17 benchmarks. We also provide a GitHub repository with the gem5 implementation and required scripts to reproduce our simulation results. Finally, we provide a binary analysis infrastructure based on Radare2 that allows the compilation of binaries with the proposed Epoch markings.

C.2 Artifact Check-List (Meta-Information)

- **Program**: SPEC'17
- **Compilation**: We compiled SPEC'17 with clang-3.9 and the gem5 simulation infrastructure with gcc-5.4.0.
- **Binary**: Our pass is implemented on top of Radare2 4.3.0.
- **Data set**: Reference input size of SPEC'17 benchmarks.
- **Run-time environment**: Linux with Docker containers.
- **Run-time state**: We use SimPoint methodology to generate up to 10 representative intervals that accurately characterize end-to-end performance. Each interval consists of 30 million instructions.
- **Output**: Plots are output by the provided scripts. Scripts are provided to generate each of the Evaluation figures.
- **Experiments**: Please refer to Section C.5.3.
- **How much disk space required (approximately)**: 1GB.
- **How much time is needed to prepare workflow (approximately)**: 10 minutes.
- **How much time is needed to complete experiments (approximately)**: 1 day.
- **Publicly available**: Yes.
- **Code licenses (if publicly available)**: MIT License.
- **Workflow framework used**: HTCondor for job management.
- **Archived**: DOI: 10.5281/zenodo.4429956. But we recommend using the latest version from GitHub.

C.3 Description

C.3.1 How to Access. Our complete simulation implementation is available at https://github.com/dskarlatos/JamaisVu.
C.3.2 **Hardware Dependencies.** Any hardware capable of running the gem5 simulator is sufficient.

C.3.3 **Software Dependencies.** We use Docker and provide a complete Dockerfile that captures all the software dependencies required to build our simulation infrastructure.

C.3.4 **Data Sets.** We run SPEC17 with the reference input size. Because of a simulation issue with gem5, we exclude 2 applications (cactuBSSN and imagick) out of 23 from SPEC17.

C.4 **Installation**

Build time: 5 to 10 minutes depends on the machine.

**Required libraries.** All libraries that are required by gem5. The instruction can be found at [https://www.gem5.org/documentation/learning_gem5/part1/building/](https://www.gem5.org/documentation/learning_gem5/part1/building/). We also provide a Docker image for building gem5.

C.5 **Experiment Workflow**

C.5.1 **Overview.** To reproduce our results, we created 5 studies under directory $GEM5_ROOT/scripts. Each study corresponds to a figure in the Evaluation section. The description of each study is as the following:

1. perf, which corresponds to Figure 7 in the paper. It simulates all three schemes plus unsafe baseline and measures normalized execution time;
2. elemCnt, which corresponds to Figure 8 in the paper. It performs a sensitivity study on the number of entries per bloom filter for CoR, Epoch-Iter-REM, and Epoch-Loop-REM;
3. activeRecord, which corresponds to Figure 9 in the paper. It performs a sensitivity study on the number of (ID, PC-Buffer) pairs for Epoch-Iter-REM and Epoch-Loop-REM;
4. CBFBits, which corresponds to Figure 10 in the paper. It performs a sensitivity study on the number of bits per counting bloom filter entry for Epoch-Iter-REM and Epoch-Loop-REM;
5. CCGeometry, which corresponds to Figure 11 in the paper. It performs a sensitivity study on the counter cache geometry for Counter.

C.5.2 **Clone Jamais Vu.** Jamais Vu is publicly available on GitHub. To clone the repository, run

```bash
git clone https://github.com/dskarlatos/JamaisVu.git
```

C.5.3 **Environment Setup.** Set environment variables

```bash
export GEM5_ROOT=/path/to/gem5
export WORKLOADS_ROOT=/path/to/SPEC2017
```

Note that, the workload directory must be structured appropriately before using any of the scripts. Please refer to this instruction1 for more details.

C.5.4 **Compile gem5.** Due to a gem5 bug2, it must be compiled in Ubuntu 16.04 to avoid crashing on some benchmarks. To address this issue, we provide a Docker image for compilation. To build the Docker image and compile gem5, run command

```bash
cd docker && docker build -t JamaisVu.
```

C.5.5 **Submit Jobs.** Assuming that the system has HTCondor installed, enter $GEM5_ROOT/scripts/, the script submit.t is used for job submission. Run command

```bash
cd $GEM5_ROOT/scripts/ & & ./submit */*.cfg
```

will submit jobs for every study. It takes about 20 minutes to submit all the jobs.

C.5.6 **Check Status.** To check job status via condor: run command condor_q

which prints the total number of running jobs and remaining jobs. To print detailed job status information for each study: under $GEM5_ROOT/scripts/, run command

```bash
./status
```

It takes about 1 day to finish all jobs on a server with 80 cores.

C.5.7 **Collect Results.** After all jobs are finished, you can collect the experiment results. Each study has a script named collect under its directory, the script will read gem5 statistics and create plots for the study. Under $GEM5_ROOT/scripts/, run command

```bash
find . -name collect -type f -exec {} >/dev/null \;
```

to collect results for all studies (do not forget backslash and semicolon at the end of the command). After executing this command, there will be figures in PDF format under $GEM5_ROOT/scripts. Please refer to Section C.6 for expected results.

C.6 **Evaluation and Expected Result**

The collected plots for each study should match its corresponding figure in the directory $GEM5_ROOT/scripts/expectedResults.

C.7 **Methodology**

Submission, reviewing and badging methodology:

- [https://www.acm.org/publications/policies/artifact-review-badging](https://www.acm.org/publications/policies/artifact-review-badging)
- [http://cTuning.org/ae/submission-20201122.html](http://cTuning.org/ae/submission-20201122.html)
- [http://cTuning.org/ae/reviewing-20201122.html](http://cTuning.org/ae/reviewing-20201122.html)

REFERENCES


