Delay-on-Squash: Stopping Microarchitectural Replay Attacks in Their Tracks

CHRISTOS SAKALIS and STEFANOS KAXIRAS, Uppsala University, Sweden
MAGNUS SJÄLANDER, Norwegian University of Science and Technology, Norway

MicroScope and other similar microarchitectural replay attacks take advantage of the characteristics of speculative execution to trap the execution of the victim application in a loop, enabling the attacker to amplify a side-channel attack by executing it indefinitely. Due to the nature of the replay, it can be used to effectively attack software that are shielded against replay, even under conditions where a side-channel attack would not be possible (e.g., in secure enclaves). At the same time, unlike speculative side-channel attacks, microarchitectural replay attacks can be used to amplify the correct path of execution, rendering many existing speculative side-channel defenses ineffective.

In this work, we generalize microarchitectural replay attacks beyond MicroScope and present an efficient defense against them. We make the observation that such attacks rely on repeated squashes of so-called "replay handles" and that the instructions causing the side-channel must reside in the same reorder buffer window as the handles. We propose Delay-on-Squash, a hardware-only technique for tracking squashed instructions and preventing them from being replayed by speculative replay handles. Our evaluation shows that it is possible to achieve full security against microarchitectural replay attacks with very modest hardware requirements, while still maintaining 97% of the insecure baseline performance.

CCS Concepts: • **Computer systems organization → Superscalar architectures**; • **Security and privacy → Side-channel analysis and countermeasures**.

Additional Key Words and Phrases: microarchitecture, side-channels, security, replay attacks

ACM Reference Format:
Christos Sakalis, Stefanos Kaxiras, and Magnus Själander. 2021. Delay-on-Squash: Stopping Microarchitectural Replay Attacks in Their Tracks. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (January 2021), 24 pages. https://doi.org/TODO

1 INTRODUCTION

Due to the complexity of modern high performance processors, there exists a large amount of microarchitectural state that can be abused by a malicious application to create side-channels, with the intent of observing the behavior of security sensitive applications and leaking sensitive data [33, 53]. This is the case even for trusted execution environments (also referred to as secure enclaves) that are designed to protect particularly sensitive applications from outside interference, including interference from a privileged context (e.g., the operating system). Although there has been extensive research in the field [17, 33], typical modern processors delegate the responsibility for defending against such side-channels to the software, as general purpose hardware side-channel defenses usually incur large overheads [33]. Thankfully, side-channels often suffer from high

New Paper, Not an Extension of a Conference Paper.
Authors' addresses: Christos Sakalis, christos.sakalis@it.uu.se; Stefanos Kaxiras, stefanos.kaxiras@it.uu.se, Uppsala University, Box 337, Uppsala, Sweden, 75105; Magnus Själander, magnus.sjalander@ntnu.no, Norwegian University of Science and Technology, IT-bygget, Gløshaugen, Trondheim, Norway, 7034.

XXXX-XXXX/2021/1-ART1
https://doi.org/TODO

noise and low reliability, which has made designing secure software without introducing large overheads possible, as an attacker has to perform the same attack several times before they are able to reliably leak sensitive information. This can be difficult for the attacker to achieve, especially if the application is running in a secure enclave, as in such cases not even a compromised operating system can manipulate the execution of the application directly, making it impossible to repeat an attack. While it is not hard to imagine cases where the attacker is targeting specific immutable data and the code can be arbitrarily triggered (e.g., to encrypt some data), in a lot of cases (e.g., SGX implementations of Tor [1, 24], secure database implementations, or systems secured against rollbacks [34]) the attacker targets transient execution data (e.g., Tor traffic) and has only one single opportunity to perform the attack and leak information. In these cases, the majority of the available side-channels are not effective, as it is not possible to distinguish a single iteration of the attack from system noise [3, 33, 35].

MicroScope and *microarchitectural replay attacks* in general, as introduced by Skarlatos et al. [47], enable an attacker to "trap" the execution of an application and force it to re-execute specific regions of code ad infinitum. With MicroScope, which focuses on secure enclaves, this is achieved by abusing a combination of speculative execution and page fault handling, in cases where the latter is still delegated to the (malicious/compromised) operating system (OS). Under typical execution, if an address translation misses in the translation lookaside buffer (TLB), a page table walk is triggered. While the page walk is happening, the application is able to continue executing speculatively, as long as there exist instructions that do not depend on the faulting memory instruction. If during the page walk, it is determined that the page is not available and that the OS needs to be invoked, then the speculatively executed instructions are squashed and execution restarts from the faulting instruction. This time another page walk might be needed, as the translation still does not exist in the TLB, but since the operating system has now mapped the page, the page walk typically succeeds. MicroScope takes advantage of this behavior by having the OS signal that it has mapped the page without actually doing so. This traps the victim application in a loop where a memory instruction (referred to as **"the handle"**) misses in the TLB, triggers a page walk and continues executing speculatively, triggers a page fault, squashes the speculatively executed instructions, re-executes the faulting instruction and misses in the TLB again. By triggering these loops at specific parts of the code, just before the instructions that cause the side-channel information leakage, the attacker can repeat the side-channel until all the underlying noise is filtered out, making even the least reliable side-channels easy to exploit, regardless if the application is executing in a secure enclave or not.

While MicroScope focuses specifically on the OS abusing the page handling mechanism found on some enclaves, other sources of speculation and re-execution can be similarly abused, in some cases even from an unprivileged context. For this reason, in this work we aim to solve the problem of microarchitectural replay attacks in general, and not just MicroScope. Specifically:

- We generalize from re-execution brought on by page faults to re-execution brought on by any form of speculation in modern processors, some of which do not require a malicious OS.
- We also generalize the method developed by Skarlatos et al. to be applicable to cases where a single handle cannot by itself trigger a large number of re-executions, by introducing a method that utilizes multiple handles.
- Finally, we introduce **Delay-on-Squash**, a solution to this critical issue of microarchitectural replay attacks, which can transparently provide protection for MicroScope and future attacks, while avoiding significant impact on performance, energy, area, and implementation complexity.

A very simple but naïve solution would be to simply disallow speculative execution after a page walk miss, but this would only protect against MicroScope itself and not against microarchitectural replay attacks that utilize other handle types or multiple handles (Section 3.2). At the same time, we cannot just disable speculation every time a squash happens nor can we implement defenses for all possible side-channels, as that would be detrimental to the performance of the system (Section 7). Furthermore, speculative side-channel defenses that track the *data dependencies* of side-channel instructions, such as NDA [54], STT [58, 59], and others [5, 6, 15, 30], do not work in this context, because the side-channel instructions may actually be in the *correct path of execution* and *can also be fed with non-speculative data coming from before the point of misspeculation* that is used for replay. Broader defenses (not restricted to data dependencies) that could work in such a case, e.g., InvisiSpec [56], Delay-on-Miss [43, 44], and many others [2, 23, 25, 26, 41, 42, 49], not only focus on a small subset of side-channels but also incur much heavier penalties. Instead, Delay-on-Squash selectively delays the speculative execution of instructions when it detects that they might be used as part of a microarchitectural replay attack. We observe that if an attack requires microarchitectural replay to be successful, then we do not need to restrict all speculation but rather only repeating speculation interleaved with misspeculation. To achieve this, we use a lightweight mechanism, based on Bloom filters, that (i) tracks which instructions have been squashed and re-issued due to misspeculation, and (ii) prevents them from executing speculatively. We propose a method for clearing the Bloom filters that is based on the youngest seen handled (see Section 5.2), which assures that an instruction can only be squashed and re-executed once. Our evaluation shows that a fully secure configuration of Delay-on-Squash, requiring little storage and overall overhead, can achieve 97% of the performance of a baseline insecure out-of-order CPU.

The rest of the paper is structured as follows: First, we will discuss side-channels and why several iterations of an attack are necessary (Section 2), followed by a discussion on how MicroScope comes into play and why it is an important security threat (Section 3). Then, with the background information out of the way, we will discuss our expanded microarchitectural replay attacks (Section 3.2) and our overall threat model (Section 4). We will present Delay-on-Squash (Section 5) and evaluate its performance energy overheads (Section 7). Finally, we will discuss other related work, including existing speculative and non-speculative side-channel defenses and why they cannot be used to effectively prevent all microarchitectural replay attacks (Section 8), as well as how Delay-on-Squash differs from Jamais Vu [48].

2 SIDE-CHANNELS

In order to provide compatibility across different hardware implementations, modern CPU architectures separate the visible *architectural* behavior and state of the system from the underlying *microarchitectural* implementation. For each visible architectural state, there exist one or more corresponding hidden microarchitectural states (μ-states) [36]. The users/applications interact with the architectural state, but have limited or no access to the underlying μ-state. However, while it is usually not possible to observe the μ-state directly, it is possible to infer it based on observable side effects.

Microarchitectural side-channel attacks take advantage of the μ-state of modern CPUs to leak information under conditions where it is not possible to do so on the architectural level. For example, cache side-channels take advantage of the difference in timing between a hit or a miss in the cache to encode information [33, 39, 57], by indirectly manipulating and probing the state of the cache through normal memory operations. Similar timing side-channels can also be constructed in other parts of the system [17], such as by utilizing functional unit (FU) contention. Finally, non-timing side-channels are also possible, exploiting side effects of the execution such as power consumption [27] or EMF radiation [16]. As these side-channels are not purposely designed communication channels but rather side effects of the normal architectural and microarchitectural behavior of the system, they are inherently noisy and unreliable. For example, when using a cache-based side-channel, there is nothing that prevents the cache line(s) being used for the side-channel from being evicted by a third process in the system. Similarly, interrupts, context switches, and other interruptions in the application execution can also disrupt the side-channel. Since the system does not provide any architectural mechanisms for synchronizing the transmitter and the receiver during side-channel operations, these also have to be constructed using the side-channel itself or other mechanisms.

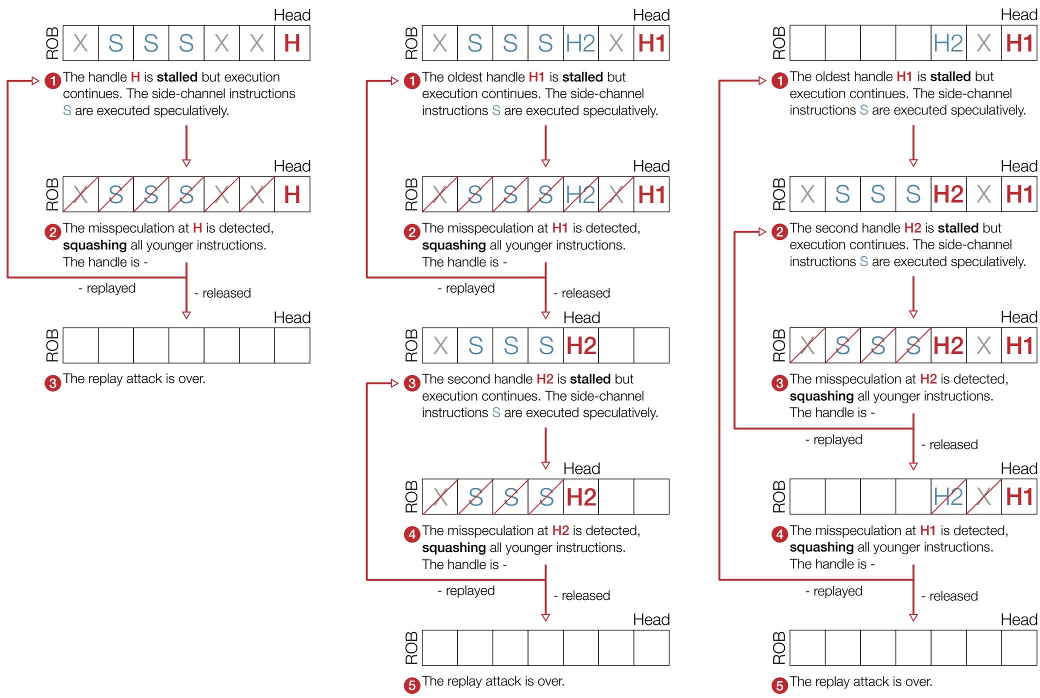
All these issues make exploiting side-channels harder, but not impossible. After all, these issues can be found in conventional communication channels as well, especially in the layers closest to the actual hardware and transmission mediums. Similarly to how modern communication protocols are designed with the underlying channel characteristics in mind, so can protocols for side-channels be designed. For example, noise on a side-channel can be filtered out using error detection and correction codes, combined with statistical methods [35]. The question then becomes *not if a side-channel can be exploited, but rather how easy, reliable, and fast the channel is.* As many of the underlying issues can be resolved by simply repeating the transmission of information through the side-channel until successful, whether a side-channel can be *practically* exploited becomes a function of the delay between each retransmission (i.e., how fast can the side-channel be repeated) and the average number of retransmissions necessary (i.e., how many times does the side-channel have to be repeated). Under the conditions described by Maurice et al. [35], both the transmitter and the receiver are under the full control of the attacker. This enables the attacker to not only control when the side-channel transmission takes place, to better synchronize the transmitter and the receiver, but to also repeat the transmission as many times as necessary. However, this is not always the case. For example, sometimes the transmitter is not a purposely designed application but a targeted victim, such as a cryptographic application running in a secure enclave. The attacker then uses side-channels to monitor the behavior of the application under normal execution and infer sensitive information, such as cryptographic keys. In some cases, the attacker is able to directly execute or otherwise trigger the execution of the victim application at will, repeating the execution as many times as necessary to extract the keys. This ability to *replay* the victim code multiple times is crucial in being able to reliably exploit the utilized side-channel, due to the issues we have already discussed. This is where MicroScope, a groundbreaking *microarchitectural replay attack* technique, comes into play.

3 MICROARCHITECTURAL REPLAY ATTACKS

Microarchitectural replay attacks, as introduced by Skarlatos et al. [47], are not by themselves a side-channel attack, speculative or otherwise. Instead, they can be seen as a tool to *amplify* the effects of side-channel attacks, enabling the attacker to mount a successful attack under conditions where it would not be possible otherwise. This is why microarchitectural replay can be so dangerous: even the smallest, most innocuous amount of information leakage can be amplified and abused. This is particularly dangerous when applied to secure enclaves, where the applications are security sensitive and the programmers are typically expected to manage the risk of side-channels. In addition, even though they exploit speculative execution, microarchitectural replay attacks *are not the same* as speculative side-channel attacks. Whereas the latter target the wrong execution path, effectively bypassing software and hardware barriers to access information illegally, microarchitectural replay attacks can amplify even the *correct* path of execution. This renders defenses that stop speculative data transmission [54, 58, 59] ineffective, since a replay attack can also amplify side-channel instructions that are on the correct path.

3.1 MicroScope

Many modern CPUs offer secure execution contexts referred to as *trusted execution environments* (sometimes also referred to as “secure enclaves”) that protect the executed code from outside



(a) Single Handle (b) Serial Handles (c) Nested Handles

Fig. 1. Three different microarchitectural replay attack patterns. The instructions in the reorder buffer are marked with an 'H' for the replay handles, 'S' for the side-channel instructions, and 'X' for other instructions not specific to the attack. The currently acquired handles are denoted in bold/red.

interference, including interference from the operating system (OS) or the hypervisor. The characteristics of these enclaves differ for each architecture, but they typically include encrypted memory for applications running in the enclave, code verification to prevent malicious code from being executed in the enclave, and hardware-enforced isolation of the enclaved execution context from any other execution context in the system, including the OS. These measures are meant to protect sensitive code and data, such as cryptographic functions and their keys, from any attacker that might have compromised other parts of the system, including the OS or the hypervisor. MicroScope targets exactly this case, focusing specifically on Intel's Secure Guard Extensions (SGX) enclave, although the underlying exploitable concept is not limited to SGX. MicroScope exploits the fact that under SGX the page management of the application is still delegated to the OS[1], to capture the execution of the application and force the application to be re-executed as many times as necessary for the side-channel attack to be successful.

Specifically, MicroScope [47] takes advantage of how page faults are handled during execution and of the out-of-order capabilities of modern CPUs. When a memory load (referred to as the **handle** in MicroScope) misses in the TLB, a (typically) hardware page-walker tries to resolve the miss by walking the page table residing in the main memory. While the page walker is trying to

[1]This is considered secure because the memory accessed under SGX execution is cryptographically encrypted and verified, preventing even the OS from accessing or manipulating it.

resolve the TLB miss, the victim application will continue its execution speculatively. By abusing this speculative execution mechanism and the squashing and re-execution caused by it, MicroScope is able to trap the execution of the victim application in a loop for an arbitrary number of *replay iterations*, until the attacker is able to reliably denoise the side-channel. A high level view of this process can be seen in Figure 1(a), with **H** representing the load/handle that keeps missing in the TLB, and **S** representing the information-leaking side-channel instructions that are being arbitrarily replayed by the attacker.

3.2 Replay Attacks Beyond MicroScope

MicroScope only exploits page faults, but that does not mean that this is the only behavior that can be exploited. Specifically, under MicroScope, a single instruction that page faults is used as a handle to replay a set of instructions indefinitely (Figure 1(a)). This is possible because (i) page faults are a specific type of misspeculation that can be repeated indefinitely, and (ii) there is nothing in the architecture that prevents an instruction from misspeculating several times in a row. However, it is conceivable that other forms of speculation can be used as handles. For example, in the short time since MicroScope has been released, Skarlatos et al. have already described a potential attack using speculative load re-ordering [48]. Even forms of speculation that cannot be repeated indefinitely, such as branch prediction, or even transactional memory[2], can be abused. Once a branch has been executed, the correct path is known and the incorrect path will not be mispredicted a second time. Similarly, transactions usually abort after a number of tries, at which point they follow a fallback path. By using multiple handles (**H1** and **H2** in Figure 1(b)), an attacker could extend the duration of the attack for each case [47, 48], especially if handles of different types are used together. In addition, if the system restricted the number of times each instruction is allowed to misspeculate and then be re-executed speculatively, e.g., as a simple but naïve solution to MicroScope, an attacker would still be able to use multiple handles to force a finite but tangible number of replays.

We can see that multiple handles acquired and released one after the other can be abused by an attacker to bypass some of the restrictions posed by different kinds of speculation and system restrictions. However, using handles *serially* like this only allows for a limited number of replays, with the total number being the sum of the replays of each handle. In contrast, Figure 1(c) shows a way of *nesting* handles, where each handle (**H1** in the figure) amplifies the number of replays of the next handle (**H2**). This works by (i) having the inner handle **H2** cause as many replays as possible before releasing it (② and ③ in Figure 1(c)), (ii) using the outer handle to squash and then re-execute the inner handle (① and ④), (iii) re-acquiring the new (as far as dynamic instructions are concerned) inner handle, and (iv) repeating indefinitely. With this technique, the total number of replays is not the sum of each individual handle's replays, but instead the product, growing exponentially with each handle. This can make a huge difference in the number of replays. For example, if the attacker is able to acquire five handles and use each only once (i.e., one replay each), with the serial handles the total number of attack iterations would amount to ten, while with the nested handles it would amount to 32. Of course, nesting handles has its own challenges, including the fact that not all forms of speculation can be used; in some cases (e.g., page faults) the misspeculation is not handled until after the instruction has reached the head of the reorder buffer (ROB). On the other hand, modern microarchitectures allow branch prediction to go several levels deep, resulting in the possibility of multiple outstanding (unresolved) predictions at any one time. By arranging for *older* branches to depend on *longer latency* operations (e.g., misses deeper in the memory hierarchy) and thus resolve slower than younger branches, an effective nested

[2]Transactional memory is not necessarily implemented as conventional speculative execution, confined within the reorder buffer, but it does have similar characteristics.

multiple-handle attack can be mounted. Similarly, if *closed nesting* of transactions is supported [22], the same effect can easily be achieved by controlling the abort of the inner versus outer transactions. In closed nesting, an abort of the inner transaction does not abort the outer transaction, which will then proceed to repeat the inner transaction.

Our goal is not to simply stop the currently known attacks (namely MicroScope) but to secure speculative execution from as many present and future microarchitectural replay attacks as possible. The serial and nested attacks that we have developed, currently have limited practical applications, but practical attacks with multiple handles and alternative handles have already been demonstrated in other works [47, 48]. Therefore, we claim that serial and nested handles cannot be safely ignored, and we have designed and evaluated our solution accordingly.

4 THREAT MODEL

MicroScope, and microarchitectural replay attacks in general, only make sense under certain conditions. For example, in the specific case of MicroScope, the assumption is that the OS is restricted from accessing the victim's execution state and memory. If this is not the case, then the OS can manipulate the victim directly, without the need for MicroScope. At the same time, even if direct manipulation is not possible, there are cases where it is possible to repeat the execution of the side-channel without using a microarchitectural replay attack. For example, a digital rights management (DRM) system that accepts encrypted data and decrypts them with a secret key could be fed the same data repeatedly by a malicious attacker, allowing for enough iterations to reliably leak the secret key. This is possible because, in this case, the secret is immutable, regardless of the input data. On the other hand, as a counter example, there exist applications where the secrets consist of transient data, such as SGX implementations of Tor [24] (where the sensitive data are arbitrary network packets) or applications where countermeasures against rollback attacks have been taken [34]. In such cases, the attacker has only a single opportunity to successfully leak the data. These are exactly the cases where microarchitectural replay attacks become useful, while also being the cases we expect to be the least protected against side-channel attacks, as, before MicroScope, they were very hard to exploit.

While MicroScope focuses on secure enclaves, one can imagine how a microarchitectural replay attack can be performed from a non-privileged context as well. For example, if the branch predictor is abused to create handles, it is possible to do a microarchitectural replay attack from an unprivileged SMT thread. We observe that for a microarchitectural replay attack to be necessary, two conditions have to apply:

(1) The side-channel used in the attack is ineffective unless it is repeated numerous times, as otherwise there would be no reason to replay the attack code.
(2) The information-leaking victim code is not repeated enough times, either by the program (e.g., in a loop) or by the attacker (e.g., by interfering with the victim's execution), to let the attacker reliably deduce the leaked information.

These two conditions combined mean that it is completely safe, under our current threat model, to allow a single execution of a side-channel attack while under speculation (Transient Leakage of 1, Section 6), as it will not be effective. Any additional attempts will be blocked by our replay-blocking defense mechanism, Delay-on-Squash.

5 DELAY-ON-SQUASH

The principle behind Delay-on-Squash is simple: If an instruction is issued, then squashed due to a replay handle, and then re-appears in the pipeline, it is not allowed to be re-issued, as it might constitute part of a microarchitectural replay attack. We have split the description of the Delay-on-Squash mechanism into two subsections: First, we discuss the concept on a high level, with no regard for any implementation constraints (Section 5.1), and then we discuss the practicalities of an actual implementation (Section 5.2).

5.1 Conceptual Description

To keep track of instructions that are issued, squashed, and then re-issued under the same handle, Delay-on-Squash needs to keep track of all the potential handles in the ROB. Due to the serial and nested cases that we described in Section 3.2, it is not enough to keep track of only the handle that caused the last squash, instead Delay-on-Squash takes into consideration all the handles that affect each squashed instruction.

To track all the handles, Delay-on-Squash utilizes a FIFO queue where all the dynamic instructions that can cause misspeculation and squashing are inserted during dispatch. While in the queue, these instructions are considered as potential handles and, by definition, are considered as "unsafe." Instructions can only be removed from the head of the queue and only if it can be determined that they are "safe," i.e., that they can no longer act as a handle, which happens only when they have *moved outside the window of speculation.* The instructions remain in the queue even if the instructions themself are squashed in the ROB. For an instruction to be classified as safe it first needs to appear at the head of the queue. Then, two scenarios are possible: (i) The instruction was on the wrong path of execution and is never seen again. For this case, the instruction sits at the head of the queue until a full worth of ROB instructions have been committed, thus assuring that all older handles have left the ROB. (ii) The instruction is dispatched again and the instruction will exit the head of the queue once it becomes non-speculative, for now lets assume that this happens once the instruction is committed. We will discuss the exact conditions as when an instruction is no longer speculative, and thus no longer a potential handle, in Section 5.2.2. As the queue is a FIFO, and handles can only be removed from the head of the queue, for a handle to become safe all the older handles need to also become safe. This is a very important condition, as it prevents the serial and nested replay patterns presented in Section 3.2.

With this queue of potential handles, it is now possible to keep track of squashed instructions on a per-handle basis. Specifically, every time a misspeculation occurs and instructions are squashed, Delay-on-Squash records the program counter (PC) for all the instructions that have been issued and are now being squashed. The *youngest* handle (in ROB order — in contrast, the Clear-on-Retire variant of Jamais Vu [48] tracks the oldest handle) is retrieved from the queue and is associated to the squashed PCs. These PCs then remain stored until their corresponding handle is determined to be safe, i.e., until the handle is removed from the queue. Using the youngest handle, instead of the actual misspeculating instruction that caused the squash, is important, once again, to prevent the serial and nested replay patterns. Essentially, by storing the PCs of the squashed instructions until the youngest (at the point of the squash) handle is safe, Delay-on-Squash ensures that the record of the squashed PCs will remain stored until all the handles that were present in the window of speculation during the squash have left the window and are thus safe. With this guarantee, all that remains is to check these records before issuing an instruction. If an instruction PC matches one of the stored records, it means that this instruction has previously been issued and squashed, and that *the handles that preceded it are still in the ROB and are still considered unsafe.* Such instructions are prevented from being issued until the relevant handles are deemed to be safe, and the records are removed. To prevent interference from other contexts, this information needs to be stored and restored on a context switch, much like the rest of the execution state (e.g., registers).

With this mechanism in place, we can instantly detect when an instruction has been issued, squashed, and then re-issued, even for complex cases, such as when the attacker might be utilizing nested handles. This enables us to detect and prevent replay iterations of the code, instantly stopping any microarchitectural replay attacks in their tracks. This mechanism does have one disadvantage: The pattern of "issue, squash, and re-issue" can also happen under normal speculative execution (i.e., when not under attack), for example, when a load is squashed due to a memory order violation, as in such cases the execution path remains the same. In addition, if we are executing a loop that is small enough to fit several loop iterations in the window of speculation, a squash in one of them will cause all the iterations that follow (within the window) to be delayed, as the instructions at each iteration all share the same PC. Fortunately, as we will see in the evaluation (Section 7), the actual number of cases affected by this are very small and come at a negligible performance cost (1%).

5.2 Efficient Implementation

While the Delay-on-Squash mechanism, as conceptually described in the previous section, can be implemented as an actual hardware solution, it would require large storage and expensive content addressable memories for keeping *exact sets* of squashed PCs. This would lead to prohibitively large area, latency, and energy overheads. Instead, we use Bloom filters [9] to represent the sets of PCs (of squashed instructions) that are temporarily prevented from being issued speculatively. With this approach, we can store the PCs of tens of squashed instructions with only a few bits of storage.

5.2.1 Bloom Filters. Bloom filters are hash-based, probabilistic data structures used to test if an element (in our case, a PC) is part of a set. A basic Bloom filter consists of a fixed-length bit-vector that is initially set to "zero". A new element is inserted into the filter by first hashing the element with a number of hash functions. Each hash is then used to index into the bit vector and the bit is set to "one". To check if an element is present in the filter, the same hash functions are used and the bit indicated by each hash is checked. If all bits, as indicated by the hashes, are set to "one", then the element is assumed to be present in the filter. While it is possible to get a false positive when checking the filter, it is not possible to get a false negative, which is an important property for us, as false negatives would lead to unsafe replay iterations. False positives, on the other hand, only manifest as reduced opportunities for speculation, in our case causing a negligible performance overhead. The Bloom filter's property of never resulting in a false negative and its relative simple implementation have inspired computer architects to use it for various cases where filtering is beneficial, e.g., to filter memory access in cache coherence protocols [38, 60] or set-associative caches [18].

Bloom Filter Implementation: In Delay-on-Squash we use the simplest, most efficient form of binary Bloom filters where the only way of erasing elements from the filter is to clear the whole filter by bulk-resetting it. Other approaches also exist [10, 13], but they come at increased overheads, and they would not offer significant performance benefits for our use case. For our implementation, we assume that all the hash functions of the PC of a particular dynamic instruction are precomputed during dispatch and kept in its ROB entry. Efficient hash functions can be created by only using Xor gates [51]. For each bit in the hash, a small set of bits from the PC are Xored together. The precomputed hash functions are then used (in the case of a squash) to index the Bloom filter and set the corresponding "ones," in parallel with multiple other ROB entries. We also assume that the whole process is hidden behind the back-end recovery latency following a squash [20].

Why Binary Bloom filters?: In addition to false positives, Bloom filters have another property that we need to account for when designing the Delay-on-Squash implementation: Removing individual elements (PCs) from a Bloom filter is expensive. To do that, we need to resort to *counting* Bloom filters (or one of their many varieties), significantly increasing our cost. Moreover, updating a counting Bloom filter entails several (as many as the hash functions) *read-increment-write* operations per PC. Parallelizing the insertion of multiple PCs in the Bloom filter becomes problematic as we have to watch out for conflicts and serialize conflicting PCs. That can be too expensive to do (either

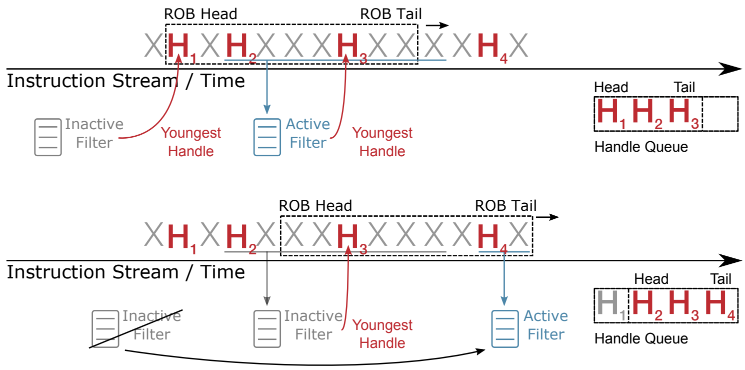


Fig. 2. An abstract visualization of the window of speculation and how it moves with the instruction stream. Handles are represented with 'H' while other instructions are represented with 'X'. Once the handle associated with the filter is resolved then the filter is marked as inactive, to be reused later.

in cycles or in read/write ports) when we need to store tens of PCs on a squash. In contrast, a simple binary Bloom filter can be updated much faster by just setting all the new "ones" in place.

Clearing the Bloom Filters: In the conceptual description, for each squash, we associate each set of squashed PCs with a handle. Observe, however, that in the case of a replay attack, the set of PCs would hardly change from squash to squash. Maintaining the same *redundant* information across multiple Bloom filters is, clearly, a waste of resources. Instead, we could consolidate the information in a single Bloom filter spanning several squashes, holding all the PCs of all the squashed instructions. For each squash, we insert the squashed PCs in the filter, and then we *associate the Bloom filter with the PC of the youngest handle in the handle queue, i.e., the entry of the tail.* While this is an effective way of keeping all the information we need to prevent replay attacks, it makes the clearing of the Bloom filter difficult. Recall that we have opted for a simple binary Bloom filter, where it is not possible to erase individual instructions. Also, we only associate the youngest seen handle with the Bloom filter, since it is not possible to remove individual PCs anyway. Finally, the condition for clearing the Bloom filter is when the associated handle leaves the window of speculation and becomes safe, i.e., all PCs in the Bloom filter are safe. As at each squash we re-associate the Bloom filter with the *youngest* handle during the squash, the lifetime of the filter can be extended an arbitrary number of times and the clearing can be deferred for an arbitrarily long time.

Instead, we use two Bloom filters and switch between them periodically. At each point in time, one of the filters is designated as *active*, while the other is inactive, *waiting to be cleared.* On a squash, the PCs of the squashed instructions are inserted in the currently active filter, and the filter is associated with the youngest handle. Meanwhile, the inactive filter is waiting for its associated handle to leave the window of speculation in order to be cleared with a bulk-reset. As soon as the inactive filter is cleared, then it can take over the role of the active filter, letting the previous active filter become inactive and eventually be cleared as well. To note here is that a squashed handle might never appear again as it might have appeared on a wrong path of execution. In such cases, the handle will remain at the head of the handle queue until as many instructions as can be stored in the ROB have been committed delaying the clearing of any Bloom filters. This is motivated by the fact that speculative replay attacks require the handle and side-channel transmitting instructions to be within the same ROB window of speculative instructions. Thus, once as many instructions as can be stored in the ROB have been committed it is ensured that all previous handles have also been committed. Of importance here is that instructions that are about to be issued need to check both filters, the active

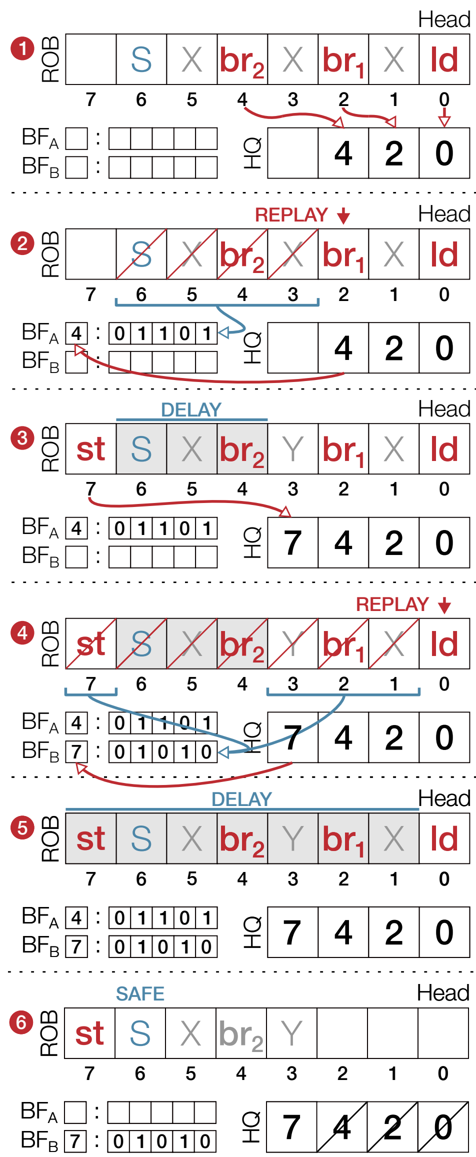


Fig. 3. Step-by-step example of the instruction tracking mechanism in Delay-on-Squash.

and the inactive, to see if they can be issued safely. The approach can naturally be extended to a cyclical list of several Bloom filters, out of which one is active, and the rest are inactive. A high level overview with just two filters can be seen in Figure 2. As the instructions at the head of the ROB are committed successfully and the ROB window moves in time, the older Bloom filter is cleared and moves to become the next active filter, with the old active filter no being marked as inactive.

The Bloom filters are context-specific, i.e., each execution context has its own set of filters, which is securely stored and reloaded on a context switch. This prevents other contexts, including the OS, from saturating or otherwise manipulating Delay-on-Squash.

A Working Example: Figure 3 contains a step-by-step example of how Delay-on-Squash tracks handles and how the Bloom filters (BF_A and BF_B in the figure) are used, when the following code is executed:

```
y = (secret & 0x1) * pi       // Not shown in the illustration
_ = load(x)                   // ld
if (cond1) { X } else { Y }   // br1
if (cond2) { X } else { Y }   // br2
sqrt(y)                       // S
store(z)                      // st
```

We have marked the potential handles in red, the side-channel instruction in blue, and the rest of the instructions in gray. Under this example, the attacker intends to first use br_1 and br_2 sequentially (one replay for each) by forcing each of them to mispredict once causing a squash and then use *ld* to squash everything and re-use br_1 and br_2 (nested attack). Branches cannot be used more than once sequentially, as once a branch is executed, the correct target becomes known. However, when the attacker triggers the *ld* handle, it is possible to re-train the branch predictor, as the victim will be stalled while the page fault is being handled. The *X* and *Y* for the *if*/*else* conditions are simply to show different paths of execution, i.e., first a mispredicted path and later the correct path. The *sqrt(y)* instruction is used to illustrate a simplified contention based attack, since *y* will either be *0* or *pi* with *pi* taking significant longer time to execute. A second thread can then perform square-root operations to determine the value of secret. The purpose of the attack is to execute the *sqrt(y)* operation as many times as possible. Note that handles based on page faults have to always be the outermost handle, as page faults are not handled speculatively, unlike branch-based handles, which are speculated early in the pipeline. For simplicity, we show the same instructions being squashed and fetched many times, as showing how instructions are tracked when the execution diverges between squashes (as is usually the case with branch mispredictions) would make the example overly complicated.

(1) The ROB contains three potential handle instructions, *ld*, br_1, and br_2, which are inserted in the handle queue (HQ) in order.

(2) The attacker has set up the microarchitectural state so that br_1 (which is mispredicted) is resolved first. This can be achieved by manipulating the branch predictor and other microarchitectural state that the branch condition (e.g., cached cache lines) depends on. When the misprediction has been detected, the instructions that follow br_1 will be squashed. The squashed instructions are inserted into the active Bloom filter BF_A, which in turn is tagged with the youngest potential handle (br_2).

(3) As execution restarts from the squashed branch, the instructions in the reconvergent path are once again dispatched in the ROB. The PCs that have been seen before, i.e., the ones that hit in the Bloom filters (marked with a gray background) will be delayed by Delay-on-Squash. As intended, Delay-on-Squash prevents *S* from being replayed. Here we also assume that a new instruction is dispatched, *st*, which is also a potential handle.

(4) For brevity, we do not show br_2 being used as a sequential handle. It is also worth to note though that br_2 has already been squashed once and is prevented from being issued and further used as a handle. Next, we consider the nested part of the attack. When the load page faults, execution will be squashed and all *issued* instructions will be added to the Bloom filter. Here we assume that we want to switch to a new filter, namely BF_B, which is tagged with the youngest potential handle, *st*.

(5) When execution resumes, all instructions that have been seen before are, once again, delayed by Delay-on-Squash. This includes the side-channel instruction *S*, which is still in BF_A.

(6) Once all the older handles (*ld*, br_1, and br_2) have been resolved, BF_A can be cleared and the side-channel instruction S can now finally be executed. Note how br_2 has not yet been retired, but it is considered as resolved once it has been executed and verified that it has not been mispredicted.

For simplicity, we show *ld* remaining in the ROB when squashed but in practice loads that page fault are typically squashed and re-dispatched. In some cases (e.g., on a context switch), it is possible that after squashing the handle queue is left empty, as all instructions are either squashed or otherwise deemed safe. In such cases, we can run into a corner case where the Bloom filters are cleared before the squashed handles (e.g., br_1) have been re-introduced into the window of speculation, enabling the attacker to perform several replay iterations. We handle such cases conservatively by delaying the clearing of the Bloom filters by the length of the dynamic instruction window, which is the longest window for any handles to be re-introduced. This happens very rarely, only affects instructions that hit in the Bloom filters, and has no measurable effect on performance.

5.2.2 Handles and the Window of Speculation. We have talked about handles that can cause misspeculation and squashing, and when such handles can be considered as safe. In the most naïve approach, we can consider handles to be safe when they reach the head of the ROB and are retired, but this is awfully pessimistic. Instead, we draw from the existing research on speculative execution [4, 7, 42–44, 56, 62] and consider handles as safe when they can no longer cause squashing, regardless of their position in the ROB. Specifically, we have adopted the approach of using speculative shadows by Sakalis et al. [42–44], a mechanism for detecting the earliest point at which an instruction is no longer speculative. While these shadows are designed to work with speculative side-channel defenses, which do not necessarily work against microarchitectural replay attacks (Section 8), the underlying principle can still be used.

According to Sakalis et al., any speculative instruction that can cause squashing is referred to as a "shadow-casting" instruction. Depending on the type of speculation, Sakalis et al. have defined different types of shadows [42, 43] (e.g., "E-Shadows" are cast by instructions that might raise an exception, as is the case with the page faults in MicroScope), but these can be extended to include other types of speculation as well, such as the transactional memory case described earlier. Once an instruction (i) is no longer shadowed by another instruction and (ii) no longer casts any shadows itself, i.e., when there is no reason for said instruction to be squashed, the instruction is considered non-speculative. At this point, the instruction has left the window of speculation and, assuming that the instruction was a potential handle, it can be considered as safe. The advantage of this approach is that it is possible for potential handles to reach the safe state a lot earlier than if we were waiting for them to retire. In addition, Sakalis et al. also describe a hardware implementation to track speculation based on a FIFO queue [43], where younger shadow-casting instructions are only resolved once all older shadow-casting instructions have also been resolved. This design fits well with Delay-on-Squash, as the handle queue has similar characteristics.

Note that while using the speculative shadows is not necessary and the use of alternative methods is possible (e.g., using the head and tail of the ROB), doing so in a naïve manner can lead to significant performance degradation [42, 62].

5.2.3 Software Support. Delay-on-Squash can operate transparently to the user and does not need any library, compiler, or operating system support to work. However, a mechanism to inform the program being executed of continuous replay attempts would enhance both security and usability, allowing the program to implement custom security routines for such cases. In addition, if an attack holds a handle for a long period of time, constantly preventing the application from making forward progress, the CPU should interrupt the program, once again allowing it to handle the security threat using custom security routines.

6 SECURITY ANALYSIS

Delay-on-Squash is not intended to protect against attacks where the sensitive code is replayed by the application itself, e.g., in cases where a loop accesses the exact same sensitive data in each iteration. In such cases, microarchitectural replay is not even necessary. As discussed in (Section 4), Delay-on-Squash only guarantees security with respect to the amplification of a side-channel and does not block the first use of a side-channel while under speculation. Sometimes, it is possible to leak information with a single iteration of an attack, although this is rare and not easy to achieve [3, 33, 35], especially when the attacker does not have full control of the victim. Delay-on-Squash does not protect against such attacks, as they are indistinguishable (and are, in fact, part of) normal execution. Instead, if such protections are required, Delay-on-Squash can be combined with other defense mechanisms, such as traditional defenses against side-channel attacks (Section 8). With Delay-on-Squash offering low overhead protection against many side-channel attacks (as most attacks require multiple iterations [3, 33, 35]), the additional defenses need only to focus on the remaining cases, lowering the complexity and overhead needed. Specifically:

- For Transient Leakage (TL) side-channels, as defined by Skarlatos et al. [48], that require microarchitectural amplification to leak information, Delay-on-Squash stops speculative execution of all instructions that have been squashed once in the instruction window. The Transient Leakage of Delay-on-Squash is always 1 (only one iteration is allowed), as Delay-on-Squash always waits until *all possible handles* in the ROB have been resolved. In comparison, the hardware-only version of Jamais Vu, Clear-on-Retire [48], can have a Transient Leakage of up to ROB-1, depending on the number of handles that can be found in the ROB.
- On the other hand, Non-Transient Leakage (NTL) side-channels that require no microarchitectural amplification to leak information fall outside the scope of Delay-on-Squash and information leakage remains the same with or without Delay-on-Squash. For security against such attacks, some other defense would be needed in parallel. The benefit of Delay-on-Squash in these cases is that it relieves us from having to defend against both TL and NTL side channels and allows us to concentrate on NTL, for which much more efficient defenses have been proposed (Section 8).

In addition to preventing microarchitectural replay attacks, we also want to ensure that Delay-on-Squash does not introduce any new side-channels into the system. As already discussed, the state kept by the mechanism (Bloom filters etc.), need to be kept isolated from other contexts and stored/restored on a context switch. This prevents the attacker from manipulating Delay-on-Squash either to “make it forget“ a replayed instruction or to introduce unnecessary overheads in the victim application. At the same time, the attacker cannot in any way probe the information that the Delay-on-Squash mechanism has of the victim, as that would allow the attacker to ascertain which instructions the victim has executed. These can be enforced by the hardware by isolating the mechanism between contexts and storing it in an encrypted manner on a context switch, much like the rest of the context (e.g., the register file) is already stored.

Finally, as Delay-on-Squash prevents instructions from executing under certain conditions, it raises whether it can be used to mount a Denial-of-Service attack on the victim. While Delay-on-Squash can affect the performance of the victim negatively, as long as instructions are being committed from the head of the reorder buffer (which is never delayed by Delay-on-Squash) then execution will not stall. Execution can only stall if the victim is caught into an execute-squash-replay loop (due to a microarchitectural replay attack), in which case no forward progress would have been made even

Table 1. The simulated system parameters.

Parameter	Value
Technology node	22 nm @ 3.4GHz
ROB size	192
Issue / Execute / Commit width	8
Cache line size	64 bytes
L1 private cache size	32 KiB, 8-way
L1 access latency	2 cycles
L2 shared cache size	1 MiB, 16-way
L2 access latency	20 cycles
Number of Bloom filters	2
Number of bits, hash functions per filter	128, 2
Hash width	lg(128) = 7 bits
Hash function	1 XOR-gate per hash bit

if Delay-on-Squash was not present. In practice, when execution is slowed down for any significant amount of time, due to an attack, the hardware should be designed to notify the software, allowing it to take appropriate measures to prevent the leakage and ensure that the system does not stall.

7 EVALUATION

We use the gem5 simulator [8] together with the SPEC CPU2006 benchmark suite [11] for the performance evaluation, combined with McPAT [31] and CACTI [32] for the energy evaluation. We generate simpoints with ScarPhase [46] and, for each simpoint, start with a one billion instruction warm up (990 million fast memory warm up and 10 million detailed pipeline warm up) followed by a 100 million instruction detailed simulation. The main parameters of the simulated system can be seen in Table 1. We have not performed an extensive evaluation of the most efficient hash function implementation. We have used a hash function that is based on one 2-input XOR gate per bit in the hash, i.e., two bits of the PC are XORed together. The delay of a single XOR gate is negligible and can easily be performed in parallel with other operations in the pipeline.

To avoid mixing different energy models, we estimate the energy expenditure of the additional handle and squash tracking structures as existing similar structures in McPAT that we size appropriately. Specifically, the handle/shadow tracking mechanism is modelled as an additional ROB structure, the Bloom filters as register files, and the hash functions for the Bloom filters as an additional integer arithmetic unit. Each hash function consists of seven bits, and with each bit requiring one XOR gate the total gate count for a single hash is seven. With an issue width of eight and two hash functions per instruction the total number of XOR gates for all the hash functions is 112. In comparison, a single 64-bit parallel adder, such as Kogge-Stone, has 128 XOR gates and significantly many more AND and OR gates. We have also tried to directly use bits from the PC to index into the Bloom filter achieving similar performance results. This indicates that it might be possible to completely eliminate the hash functions and instead use already available bits associated with each instruction to populate the Bloom filter. We leave such optimizations as future work.

We evaluate the following configurations in detail:

- **baseline:** The insecure out-of-order CPU in gem5.
- **Delay-All:** A lower-bound, non-speculative configuration where all instructions are delayed until no unsafe handles precede them. To make the comparison with the Delay-on-Squash

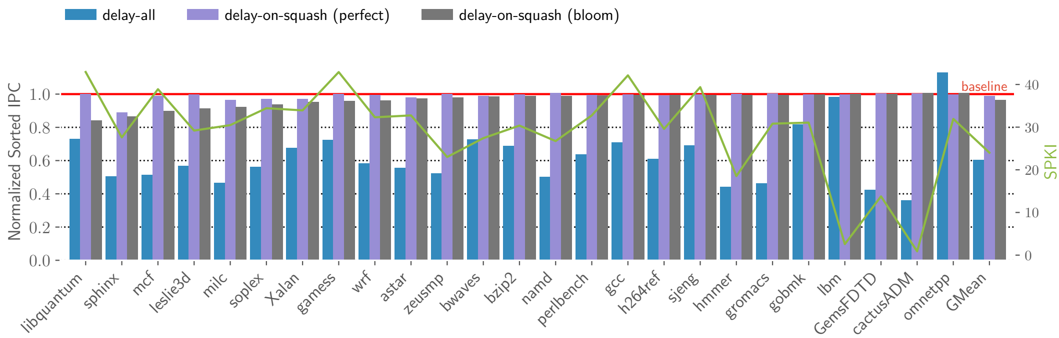


Fig. 4. Performance (instructions per cycle) normalized to the baseline (bars), overlaid with the number of squashes per 1000 instruction (SPKI – line)

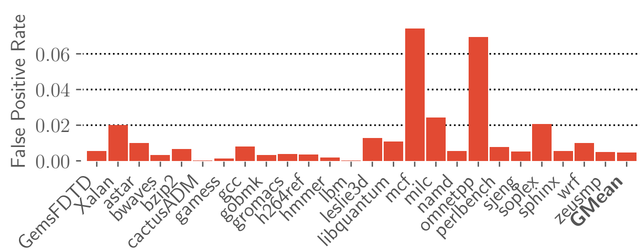


Fig. 5. Rate of Bloom filter false positives over the total number of *executed* (i.e., including squashed) instructions.

configurations fair, Delay-All utilizes the same handle-tracking mechanism as Delay-on-Squash to detect when a handle is safe (Section 5.2.2).

- **Delay-on-Squash (Perfect):** The ideal implementation of Delay-on-Squash, as described in Section 5.1. We assume that we have unlimited space for storing all the squashed instructions for as long as necessary.
- **Delay-on-Squash (Bloom):** The actual implementation of Delay-on-Squash with Bloom filters, as described in Section 5.2, with the Bloom filter parameters seen in Table 1. We chose these parameters empirically to balance performance, area, and energy usage.

In each case, the defense being evaluated is enabled for the whole duration of the execution.

7.1 Performance

Figure 4 contains the number of instructions per cycle (IPC), normalized to the insecure out-of-order baseline. In addition, it also contains the baseline number of squashes per 1000 instructions (SPKI). Overall, implementing Delay-All, which we are only including as a lower-bound configuration, would lead to 40% lower performance, when compared to the baseline. On the other hand, implementing Delay-on-Squash with a perfect filter would eliminate almost all the performance cost, reaching 99% of the baseline performance. Only one application has any perceptible performance degradation, `sphinx` (−11%), which misspeculates and squashes more often than average. In some cases, the same static instructions exist in the ROB in multiple dynamic instances, for example when a loop is dynamically unrolled in the ROB (Section 5.1). If a squash happens in such a situation, all instances of the static instructions will be conservatively treated by Delay-on-Squash as potential

Table 2. Techniques that (partially) mitigate microarchitectural replay attacks.

Mitigation	Speculative Side Channels					Reported
	Cache	Coherence	Memory-Contention	Functional-Unit	Implicit	Performance
SafeSpec [23]	✓	✗	✗	✗	✗	+3%
InvisiSpec [56]	✓	✓	✗	✗	✗	-40%
Ghost Loads [42]	✓	✓	✗	✗	✗	-20%
CleanupSpec [41]	✓	✓	✗	✗	✗	-5%
ReViCe [25]	✓	✓	✗	✗	✗	-6%
MuonTrap [2]	✓	✓	✗	✗	✗	-4%
DoM [44]	✓	✓	✓	✗	✗	-18%
CSF (CFENCE) [49]	✓	✓	✓	✗	✓	-21%
CSF (LFENCE) [49]	✓	✓	✓	✓	✓	-48%
Jamais Vu [48]	✓	✓	✓	✓	✓	-3%
Delay-on-Squash	✓	✓	✓	✓	✓	-3%

replay attacks and speculation will be restricted in a large part of the dynamic instruction window. In contrast, applications like `cactusADM` and `lbm`, which rarely misspeculate, are not affected at all. However, the amount of misspeculation in the application is not the only parameter that affects its performance with the perfect filter and, in fact, there is no correlation between the SPKI and the performance. Instead, the application's sensitivity to instruction and memory level parallelism (ILP and MLP) is also important. Restricting the speculative execution of an application which relies on ILP and MLP to achieve high performance can lead to more severe performance penalties, when compared with an application that does not rely on either [19, 44].

Finally, we have the realistic Delay-on-Squash implementation, based on the Bloom filters, with the parameters described in Table 1. Alongside the IPC, in Figure 5 we can also see the false positive rate of the applications, i.e., the number of false positives in the Bloom filter over the number of instructions executed. On average, we observe one false positive every 210 instructions (a rate of roughly 0.5%), but this number differs significantly between all the application. The application that has the highest false positive rate (`mcf`) also has lower performance, but that is not true for the application with the second highest rate (`omnetpp`). Once again, we see that the false positive rate of the filter does not necessarily affect the performance of the application. In fact, the application whose performance is affected the most is `libquantum` (−16%), which is not one of the applications with the highest false positive rates. Instead, `libquantum` is a streaming benchmark that relies heavily on MLP, so it is heavily affected by any restrictions to parallelism. At the same time, `libquantum` has the highest SPKI (42) out of all the applications, although once again there is overall no clear correlation between the SPKI and the benchmark behavior.

Overall, for Delay-on-Squash with the Bloom filters, we observe a mean performance of 97% of the baseline, two percentage points lower than the ideal version. Given that secure enclaves already exchanges performance for higher security guarantees, a 3% performance degradation is insignificant.

Table 2 compares Delay-on-Squash against several state-of-the-art speculative side channel mitigation techniques that can (partially) prevent microarchitectural replay attacks. To be noted, the table only shows the types of side-channels that they protect against and the reported performance, without considering the precise threat model nor the complexity of the proposed techniques, e.g., required modifications to the core, caches, coherence protocol, etc. Several proposed techniques hide [2, 23, 42, 56], delay [44, 49], or undo [25, 41] the microarchitectural state of speculative load instructions. Such techniques (partially) protect the memory hierarchy, but leave harder to exploit side-channels unprotected for improved performance. At the same time, side-channels that exploit

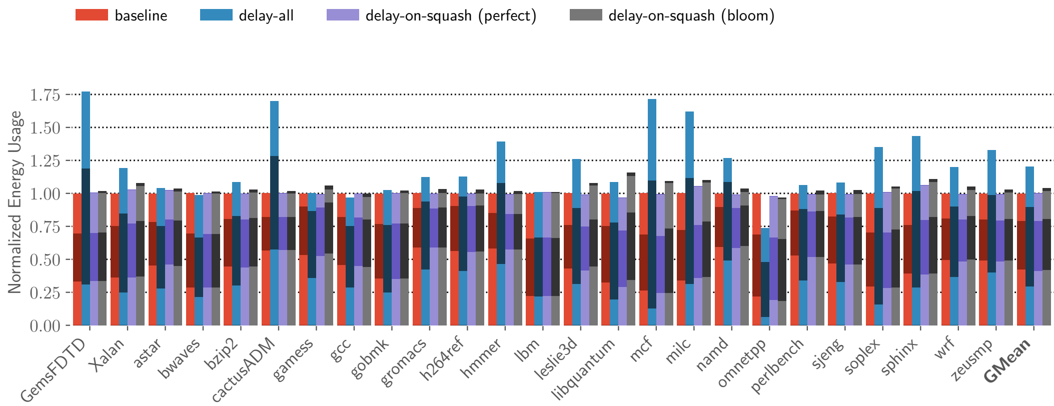


Fig. 6. Energy usage normalized to the baseline. Each bar considers of four different parts, representing the dynamic CPU energy usage (bottom, light), the static CPU energy usage (middle, shaded), the DRAM energy usage (middle, light), and the overhead of the Delay-on-Squash tracking structures (top, shaded).

the memory hierarchy often use persistent state that is only changed once evicted by other operations, resulting in relatively reliable side channels, hence microarchitectural replay attacks might not even be necessary to begin with. Techniques that protect against all speculative side-channels, such as the most restrictive version of context-sensitive fencing (CSF) [49], incur high performance overheads. Jamais Vu's [48] Clear-on-Retire has a similar performance as Delay-on-Squash, but allows more replays to be performed for serial and nested microarchitectural replay attacks.

7.2 Energy Usage

The energy usage of the different configurations can be seen in Figure 6. Each bar in the figure is split into four parts, representing (from bottom to top) dynamic, static, DRAM, and overhead energy. The overhead contains both the dynamic and static energy usage of the squash tracking mechanism and the Bloom filters, and is only included in the Bloom filter configuration, as that is the only configuration where we have a realistic hardware design.

Overall, there are two factors that affect the energy usage in the various configurations: Number of executed instructions and execution time. While we simulated all checkpoints to the same number of *committed* instructions, the number of (dynamically) executed instructions varies depending on the amount of speculation and squashing. Specifically, instructions that are executed speculatively and are squashed still require energy, but this energy is not used for useful work. For this reason, the dynamic energy usage of the applications is reduced in the strict Delay-All version.

However, while restricted speculation improves the dynamic energy usage, the static and DRAM energy usage are instead increased, because both depend on the execution time, which is increased. Specifically, unless power- or clock-gated, the CPU has the same amount of leakage regardless if instructions are executed or not. Then, the longer the CPU has to be active, the more the static energy increases. Similarly, DRAM cells need to be refreshed at regular intervals, so the DRAM energy increases with the execution time. Overall, these effects overshadow the energy gains provided by the reduced misspeculation, so the Delay-All version has an overall energy usage that is 20% higher than the baseline.

In contrast, Delay-on-Squash, with a perfect filter, is identical to the baseline. The performance difference between the two versions is very small, so there are no significant differences in the static and DRAM energy, and speculation is not restricted enough to lead to any energy usage reductions. As we are not modelling the filter mechanism, there is no overhead, but it is important to note that storing and accessing all the squashed instructions in the window of speculation would introduce very large overheads.

On the other hand, the Bloom filter version has a 4% energy overhead over the baseline, out of which half (2 percentage points) is due to the increase in execution time and half is due to the overhead of the mechanism. Increasing the size of the Bloom filters (to decrease the performance overhead) would lead to an overall increase in the energy overhead, as the gains (due to the small reduction in the execution time) would be overshadowed by the increased overhead due to the larger structures.

8 RELATED WORK

Jamais Vu: There exists only one other comprehensive solution to the problem of microarchitectural replay attacks, Jamais Vu [48], which requires compiler support to offer a good balance between security and performance. It comes with three main variants:

- **Clear-on-Retire**, a hardware only solution that is only effective against attacks utilizing a single handle, while having similar performance overheads to Delay-on-Squash.
- **Epoch**, a combined software and hardware solution that requires compiler support to protect against attacks that utilize multiple handles, or even attacks that go beyond the ROB, but at higher overhead. As we have explained in our threat model (Section 4) we consider such attacks outside the scope of microarchitectural replay. For comparison, instead of having compiler-defined epochs that can span whole loops, Delay-on-Squash can be thought of as having hardware autodetected epochs that span the window of speculation.
- **Counter**, a conservative approach where instructions are never removed from the replay filters, which has the worst performance overhead when compared against the other two variants (23%, as reported by Skarlatos et al.)

The critical difference between Delay-on-Squash and Jamais Vu is that Delay-on-Squash is designed to protect against attacks where replay is achieved through purely microarchitectural methods, while Jamais Vu also considers replay iterations due to the software architecture. We consider such attacks to not be microarchitectural replay attacks, as the replay is not done using microarchitectural methods (Section 4). Because of this insight, we are able to offer a solution that does not require any software modifications, does not introduce any additional overheads, and is more secure (lower Transient Leakage) than the equivalent hardware-only variant of Jamais Vu. However, it is also possible to combine the two methods, with Delay-on-Miss relieving some of the higher overhead mechanisms pressure of Jamais Vu, allowing for wider security with lower overheads.

Other Related Enclave Attacks and Defenses: Controlled-channel attacks [55] and Sneaky Page Monitoring (SPM) [52] use the fact that the page management is delegated to the OS to monitor the access patterns of the victim directly and leak information. As the OS has full control over the page system, such attacks can have low to zero noise and do not need to be repeated several times, so no microarchitectural replay is necessary. CacheZoom [37] is another attack that uses fine-grained control to perform a side-channel, but only targets cache side-channels. As per our threat model (Section 4), we consider such attacks outside the scope of our work, as when such attacks are possible there is no reason for an attacker to employ microarchitectural replay. We will, however, note that there exist solutions for such side-channel attacks, such as Klotski by Zhang et al. [61], but they come with steep performance and area overheads (up to 10× in execution time, depending on the configuration, for Klotski). There also exist speculative side-channel attacks that abuse speculative execution to gain access to SGX data, such as Zombieload [45] or TLBleed [21].

Such attacks are affected from the same noise issues as any other side-channel attack, and they can in fact benefit by microarchitectural replay.

Side-Channel Defenses: There are numerous works on side-channel attacks and defenses [17], particularly when it comes to cache side-channels [33]. Examples include disruptive prefetching or cache partitioning to hide the cache access patterns [12, 14, 28, 40] or oblivious RAM to hide memory access patterns [61]. Each of these solutions tries to prevent or limit the creation of specific side-channels and comes with different costs and limitations. To the best of our knowledge, there exists no published work evaluating the total cost of implementing all the different solutions that would have been necessary to prevent all the side-channels that microarchitectural replay attacks, such as MicroScope, can amplify. Also, only perfect solutions would be effective, as otherwise even the smallest observable side effects can be amplified by MicroScope. However, with Delay-on-Squash in place, a system would only need defenses against the limited range of attacks that do not require microarchitectural replay, making it possible to offer comprehensive security with reasonable complexity and overheads.

Speculative Side-Channel Defenses: There is a critical difference between microarchitectural replay and speculative side-channel attacks: The latter abuse speculative execution to *illegally* gains access to information during *the wrong execution path*, while, on the other hand, microarchitectural replay attacks can amplify *the correct path* and leak information that has been accessed *legally*. Defenses and optimizations that only try to block the wrong execution path [29, 50, 62] will not work against Microscope or other microarchitectural replay attacks. Furthermore, during a speculative side-channel attack, there is a *data dependence chain* between the illegal access and the information-leaking instructions, while in a microarchitectural replay attack the handle and the side-channel instructions have to be independent. State-of-the-art defenses that block the transmission of speculative accessed data to potential side-channel instructions, such as NDA [54], STT [58, 59], and others [5, 6, 15, 30], are ineffective in this context, because, under microarchitectural replay, the side-channel instructions may actually be in the *correct path of execution* and *can also be fed with non-speculative data*. At the same time, other defenses that are not restricted to data dependencies, such as InvisiSpec [56], Delay-on-Miss [43, 44], and many others [2, 23, 25, 26, 41, 42, 49], only focus on specific side-channels, under the assumption that not all side-channels can be exploited as easily. However, the effectiveness of microarchitectural replay attacks comes precisely from the fact that they can be used to amplify and successfully mount hard to exploit attacks, making these speculative side-channel defenses also unfit for our purposes. Even so, with Delay-on-Squash as the base, (similar with the non-speculative side-channels) some of these defenses can be modified to only cover the cases not covered by Delay-on-Squash, offering better overall security with lower overheads.

9 CONCLUSION

Microarchitectural side-channel attacks rely on observable μ-state changes caused by the victim application under attack. Such observations are commonly noisy, for example, due to the extremely short lifetime of transient μ-state or due to system interference that causes persistent μ-state to change. To successfully launch an attack, the side-channel has to be amplified and denoised by being repeated several times. This is the specific purpose of MicroScope and microarchitectural replay attacks, which trap the victim application in a loop, causing it to execute the side-channel over and over again. MicroScope specifically, has proven to be successful in leaking information from secure enclaves, but microarchitectural replay attacks, in general, may have broader applicability. At the same time, while these attacks do abuse speculative execution, existing speculative side-channel defenses are unable to mitigate them.

We make the observation that such replay attacks rely on repeated squashes of replay handles, and that the instructions causing the side-channel must reside in the same ROB window as the handles. Based on this observation, we propose Delay-on-Squash, a hardware-only technique for tracking squashed instructions and stopping them from being replayed. We further describe Delay-on-Squash can be efficiently realized, using Bloom filters and speculative shadow tracking [43].

We evaluate several configurations with different parameters, and we show that a fully secure system against microarchitectural replay attacks (according to the threat model) comes at a mere 3% performance degradation when compared to a baseline, insecure out-of-order, CPU, while also keeping the energy cost low, at 4% over the baseline.

ACKNOWLEDGMENTS

This work was supported by the Swedish Research Council grants no. 2015-05159 and 2018-05254 and Microsoft Research through its EMEA PhD Scholarship Programme grant no. 2021-020.

REFERENCES

[1] [n. d.]. The Tor Project | Privacy & Freedom Online. https://torproject.org

[2] Sam Ainsworth and Timothy M. Jones. 2020. MuonTrap: Preventing Cross-Domain Spectre-Like Attacks by Capturing Speculative State. In *Proceedings of the International Symposium on Computer Architecture*. 132–144. https://doi.org/10.1109/ISCA45697.2020.00022

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 870–887. https://doi.org/10.1109/SP.2019.00066

[4] Mehdi Alipour, Trevor E. Carlson, and Stefanos Kaxiras. 2017. Exploring the Performance Limits of Out-of-order Commit. In *Proceedings of the ACM International Conference on Computing Frontiers*. Association for Computing Machinery, 211–220. https://doi.org/10.1145/3075564.3075581

[5] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. Isolating Speculative Data to Prevent Transient Execution Attacks. *IEEE Computer Architecture Letters* 18 (July 2019), 178–181. Issue 2. https://doi.org/10.1109/LCA.2019.2916328

[6] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels. In *Proceedings of the International Conference on Parallel Architectural and Compilation Techniques*. IEEE Computer Society, 151–164. https://doi.org/10.1109/PACT.2019.00020

[7] G.B. Bell and M.H. Lipasti. 2004. Deconstructing commit. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, 68–77. https://doi.org/10.1109/ISPASS.2004.1291357

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39 (May 2011), 1–7. Issue 2. https://doi.org/10.1145/2024716.2024718

[9] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13 (July 1970), 422–426. Issue 7. https://doi.org/10.1145/362686.362692

[10] Saar Cohen and Yossi Matias. 2003. Spectral bloom filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 241–252. https://doi.org/10.1145/872757.872787

[11] Standard Performance Evaluation Corporation. 2006. SPEC CPU2006 Benchmark Suite. http://www.specbench.org/cpu2006/

[12] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization* 8 (Jan. 2012), 35:1–35:21. Issue 4. https://doi.org/10.1145/2086696.2086714

[13] Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. 2000. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking* 8 (June 2000), 281–293. Issue 3. https://doi.org/10.1109/90.851975

[14] Adi Fuchs and Ruby B. Lee. 2015. Disruptive prefetching: impact on side-channel attacks and cache designs. In *Proceedings of the ACM International Systems and Storage Conference*. Association for Computing Machinery, 1–12. https://doi.org/10.1145/2757667.2757672

[15] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. In *Proceedings of the ACM/IEEE Design Automation Conference*. Association for Computing Machinery, 1–6. https://doi.org/10.1145/3316781.3317914

[16] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic Analysis: Concrete Results. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*, Çetin K. Koç, David Naccache, and Christof Paar (Eds.). Springer, 251–261. https://doi.org/10.1007/3-540-44709-1_21

[17] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8 (April 2018), 1–27. Issue 1. https://doi.org/10.1007/s13389-016-0141-6

[18] Mrinmoy Ghosh, Emre Ozer, Simon Ford, Stuart Biles, and Hsien-Hsin S. Lee. 2009. Way guard: a segmented counting bloom filter approach to reducing energy for set-associative caches. In *Proceedings of the International Symposium on Low Power Electronics and Design*. Association for Computing Machinery, 165–170. https://doi.org/10.1145/1594233.1594276

[19] Andrew F. Glew. 1998. MLP yes! ILP no. In *undefined*. 1. https://www.semanticscholar.org/paper/MLP-yes!-ILP-no-Glew/b9b1144799183affa96c5becfd5920c039fb337e

[20] Antonio González, Fernando Latorre, and Grigorios Magklis. 2010. *Processor Microarchitecture: An Implementation Perspective*. MCP. https://doi.org/10.2200/S00309ED1V01Y201011CAC012

[21] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 955–972. https://www.usenix.org/conference/usenixsecurity18/presentation/gras

[22] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. 1994. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems* 16 (Nov. 1994), 1719–1736. Issue 6. https://doi.org/10.1145/197320.197346

[23] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *Proceedings of the ACM/IEEE Design Automation Conference*. 1–6.

[24] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. 2018. SGX-Tor: A Secure and Practical Tor Anonymity Network With SGX Enclaves. *IEEE/ACM Transactions on Networking* 26 (Oct. 2018), 2174–2187. Issue 5. https://doi.org/10.1109/TNET.2018.2868054

[25] Sungkeun Kim, Farabi Mahmud, Jiayi Huang, Pritam Majumder, Neophytos Christou, Abdullah Muzahid, Chia-Che Tsai, and Eun Jung Kim. 2020. ReViCe: Reusing Victim Cache to Prevent Speculative Cache Leakage. In *Proceedings of the IEEE Secure Development Conference*. IEEE Computer Society, 96–107. https://doi.org/10.1109/SecDev45635.2020.00029

[26] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 974–987. https://doi.org/10.1109/MICRO.2018.00083

[27] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Proceedings of the Annual International Cryptology Conference*, Michael Wiener (Ed.). Springer, 388–397. https://doi.org/10.1007/3-540-48405-1_25

[28] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. 2008. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the ACM Workshop on Computer Security Architectures*. Association for Computing Machinery, 25–34. https://doi.org/10.1145/1456508.1456514

[29] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2020. SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 39–53. https://doi.org/10.1109/SP40000.2020.00033

[30] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. 2019. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In *Proceedings of the International Symposium High-Performance Computer Architecture*. IEEE Computer Society, 264–276.

[31] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 469–480. https://doi.org/10.1145/1669112.1669172

[32] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 694–701. https://doi.org/10.1109/ICCAD.2011.6105405

[33] Yangdi Lyu and Prabhat Mishra. 2018. A Survey of Side-Channel Attacks on Caches and Countermeasures. *Journal of Hardware and Systems Security* 2 (March 2018), 33–50. Issue 1. https://doi.org/10.1007/s41635-017-0025-y

[34] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 1289–1306. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic

[35] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Boano Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Proceedings of the Network and Distributed System Security Symposium*. 1–15. https://doi.org/10.14722/ndss.2017.23294

[36] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178 [cs]* (Feb. 2019), 1–26. http://arxiv.org/abs/1902.05178

[37] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*, Wieland Fischer and Naofumi Homma (Eds.). Springer, 69–90. https://doi.org/10.1007/978-3-319-66787-4_4

[38] A. Moshovos. 2005. RegionScout: exploiting coarse grain sharing in snoop-based coherence. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 234–245. https://doi.org/10.1109/ISCA.2005.42

[39] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Proceedings of the RSA Conference*, David Pointcheval (Ed.). Springer, 1–20. https://doi.org/10.1007/11605805_1

[40] Daniel Page. 2005. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. *IACR Cryptology ePrint archive* (Aug. 2005), 1–14. https://eprint.iacr.org/2005/280

[41] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An "Undo" Approach to Safe Speculation. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 73–86. https://doi.org/10.1145/3352460.3358314

[42] Christos Sakalis, Mehdi Alipour, Alberto Ros, Alexandra Jimborean, Stefanos Kaxiras, and Magnus Själander. 2019. Ghost loads: What is the cost of invisible speculation?. In *Proceedings of the ACM International Conference on Computing Frontiers*. Association for Computing Machinery, 153–163. https://doi.org/10.1145/3310273.3321558

[43] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution through Selective Delay and Value Prediction. In *Proceedings of the International Symposium on Computer Architecture*. 723–735.

[44] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2020. Understanding Selective Delay as a Method for Efficient Secure Speculative Execution. *IEEE Trans. Comput.* 69 (Nov. 2020), 1584–1595. Issue 11. https://doi.org/10.1109/TC.2020.3014456

[45] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 753–768. https://doi.org/10.1145/3319535.3354252

[46] Andreas Sembrant, David Eklov, and Erik Hagersten. 2011. Efficient software-based online phase classification. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, 104–115. https://doi.org/10.1109/IISWC.2011.6114207

[47] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. 2019. MicroScope: Enabling Microarchitectural Replay Attacks. In *Proceedings of the International Symposium on Computer Architecture*. 318–331. https://doi.org/10.1145/3307650.3322228

[48] Dimitrios Skarlatos, Zirui Neil Zhao, Riccardo Paccagnella, Christopher W. Fletcher, and Josep Torrellas. 2021. Jamais vu: thwarting microarchitectural replay attacks. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 1061–1076. https://doi.org/10.1145/3445814.3446716

[49] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 395–410. https://doi.org/10.1145/3297858.3304060

[50] Kim-Anh Tran, Christos Sakalis, Magnus Själander, Alberto Ros, Stefanos Kaxiras, and Alexandra Jimborean. 2020. Clearing the Shadows: Recovering Lost Performance for Invisible Speculative Execution through HW/SW Co-Design. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. Association for Computing Machinery, 241–254. https://doi.org/10.1145/3410463.3414640

[51] H. Vandierendonck and K. De Bosschere. 2005. XOR-based hash functions. *IEEE Trans. Comput.* 54 (July 2005), 800–812. Issue 7. https://doi.org/10.1109/TC.2005.122

[52] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2421–2434. https://doi.org/10.1145/3133956.3134038

[53] Zhenghong Wang and Ruby B. Lee. 2006. Covert and Side Channels Due to Processor Architecture. In *Proceedings of the Annual Computer Security Applications Conference*. IEEE Computer Society, 473–482. https://doi.org/10.1109/ACSAC.2006.20

[54] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 572–586. https://doi.org/10.1145/3352460.3358306

[55] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 640–656. https://doi.org/10.1109/SP.2015.45

[56] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the IEEE/ACM International Symposium*

on Microarchitecture. 428–441. https://doi.org/10.1109/MICRO.2018.00042

[57] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the USENIX Security Symposium*. USENIX Association, 719–732.

[58] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. 2020. Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution. In *Proceedings of the International Symposium on Computer Architecture*. 707–720. https://doi.org/10.1109/ISCA45697.2020.00064

[59] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Association for Computing Machinery, 954–968. https://doi.org/10.1145/3352460.3358274

[60] Jason Zebchuk, Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Andreas Moshovos. 2009. A Tagless Coherence Directory. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 423–434. https://doi.org/10.1145/1669112.1669166

[61] Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. 2020. Klotski: Efficient Obfuscated Execution against Controlled-Channel Attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 1263–1276. https://doi.org/10.1145/3373376.3378487

[62] Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W. Fletcher, Adam Morrison, Darko Marinov, and Josep Torrellas. 2020. Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 1138–1152. https://doi.org/10.1109/MICRO50266.2020.00094