MicroScope: Enabling Microarchitectural Replay Attacks

Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher
University of Illinois at Urbana-Champaign

1 Introduction

It is now well understood that modern processors leak secrets over microarchitectural side and covert channels. These channels are seemingly everywhere—from the cache [7, 12, 11] to the branch predictor [4] and other structures [2, 1]—and are capable of leaking program secrets at many points in a program's execution.

Yet, a fundamental challenge for attackers exploiting these channels is that the channels are notoriously *noisy*. This means that multiple measurements of the same event often return widely different values. This occurs, for example, when attempting to glean secret-dependent control flow by measuring port contention inside the pipeline [1]. As a result, the attacker requires that the victim program run many times (e.g., thousands of times [1]) to increase the signal-to-noise ratio. This fact prevents attackers from learning secrets in many important scenarios, such as when the victim program runs only once.

To eliminate this limitation, this work introduces *Microarchitectural Replay Attacks*, a new class of attacks that enables an attacker to offset the measurement variation for (i.e., to *denoise*) potentially any microarchitectural side channel, even if the victim code is executed *only once*. The key observation is that, in modern out-of-order speculative cores, *a dynamic instruction may be forced to execute multiple times* due to pipeline squashes caused by page faults, exceptions, or other events. By forcing the squash and re-execution of an instruction multiple times, the attacker can repeatedly measure the execution characteristics of such instruction. We call this attack a microarchitectural replay attack.

This work also describes and implements a specific family of microarchitectural replay attacks that are applicable in the context of Intel's Software Guard Extensions (SGX) [5]. Specifically, in this environment, the attacker controls the Operating System (OS) and, while the attacker cannot see the victim's data directly, it controls the victim's demand paging. Now, suppose that there is an instruction *I* that, based on secret data, forms a noisy side- or covert-channel. In addition, suppose that the attacker finds a public-address load *L* that is older than *I* and is in the Reorder Buffer (ROB) at the same time as *I*. In this case, the attacker can arrange for *L* to page fault after a long page walk (e.g., by clearing the Present bit of the corresponding page table entry, and evicting the multi-level page table entries from the cache). While the page walk is underway, *I* executes and the attacker observes a noisy sample. Then, the OS pretends to service the page fault but keeps the Present bit cleared. As a result, *L* will go through the page walk again and *I* will execute again. This process is repeated many times, causing the replay of *I* an arbitrary number of times until the signal-to-noise ratio is reduced enough that the secret is leaked. All the while, the victim has logically run only once.

The paper performs a deep dive investigation of the capabilities of this attack, and provides a complete prototype tool called *MicroScope* that runs the attack on real hardware.[1] We investigate the attack's ability to leak secrets in straight-line code, branches, and loops. As a proof of concept, we demonstrate how the attack can denoise the notoriously-noisy side channel of pipeline port contention in a *single* run of the victim. Finally, we discuss how changing different parameters in the attack setup yields new flavors of the attack—e.g., enabling an attack to theoretically bias the output of a hardware instruction that generates true random numbers.

We released the full MicroScope framework as a kernel module, available at `https://github.com/dskarlatos/MicroScope`.

2 Brief Background

Secure enclaves [10], such as Intel's SGX [5] allow sensitive user-level code to run securely on a platform alongside an untrusted supervisor (i.e., an OS and/or hypervisor). Intel SGX uses the OS for TLB and page table management. Each page table entry contains a Present bit, which identifies if the physical page is present in memory or not. If the bit is cleared, then the translation process fails and a page fault exception is raised. The OS is then invoked to handle it. To maintain the TLB coherent while updating page table entries, the OS can selectively flush TLB entries through the INVLPG instruction.

3 Summary of the MicroScope Attack

Microarchitectural Replay Attacks (MRAs) are based on the key observation that modern hardware allows recently executed, but not retired, instructions to be rolled back and replayed if certain conditions are met. This behavior can be exploited to mount a variety of attacks (Section 6), such as denoising a side channel.

An MRA attack has three actors: Replayer, Victim, and Monitor. In a MicroScope attack, which is a type of MRA, the *Replayer* is a malicious OS or hypervisor that is responsible for page table management. The *Victim* is an application process that executes on some secret data that the attacker wishes to exfiltrate. The *Monitor* is a malicious process that performs auxiliary operations, such as causing contention and monitoring shared resources. Figure 1 shows the timeline of the interleaved execution of the Replayer, Victim, and Monitor for a MicroScope attack.

Attack Setup. MicroScope is enabled by what we call a *Replay Handle*. A replay handle can be any memory access instruction that occurs shortly before one or more security-sensitive instructions in program order.

In MicroScope, the Replayer sets up the attack by locating the page table entries required for virtual-to-physical transla-

[1] The name MicroScope comes from the attack's ability to peer inside nearly any microarchitectural side channel.

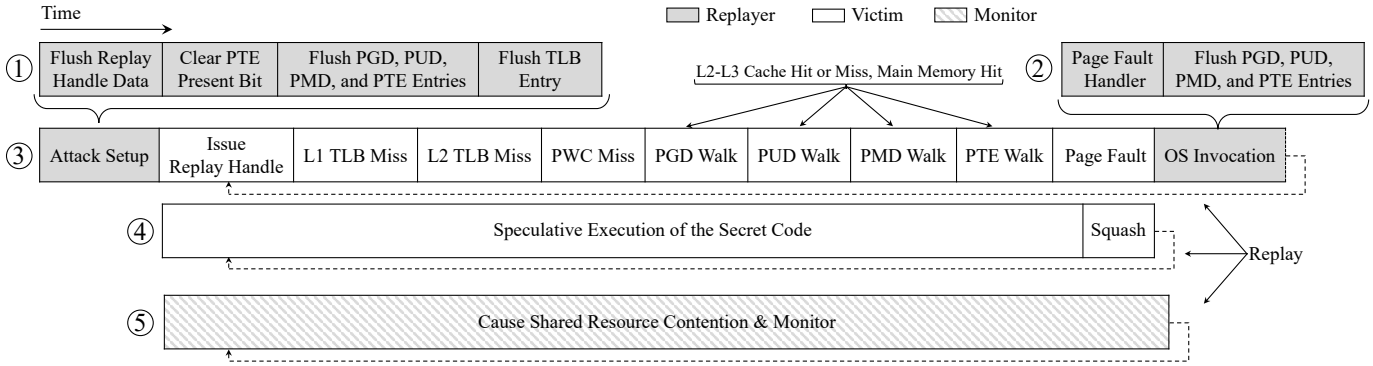


Figure 1: Timeline of a MicroScope attack. The *Replayer* is an untrusted OS or hypervisor process that forces the *Victim* code to replay, enabling the *Monitor* to denoise and extract the secret information.

tion of the replay handle. Then, it performs the following steps, shown in the timeline ① of Figure 1. First, it flushes the replay handle data from the cache. After that, it clears the Present bit of the leaf page table entry of the replay handle. After that, it flushes the translation page table entries from the caches. Finally, it flushes the TLB entry that stores the virtual-to-physical translation of the replay handle access. Together, these steps will cause the replay handle to miss in the TLB, induce a hardware page walk to locate the translations, and eventually suffer a page fault. In the meantime, instructions that are younger than the replay handle, including the sensitive instruction(s), can execute speculatively but not retire.

Speculative Execution in the Shadow of Page Walks. After the attack is set-up, the Replayer allows the Victim to resume execution and issue the replay handle as shown in timeline ③ of Figure 1. The replay handle access misses in the L1 TLB, L2 TLB, and Page Walk Cache (PWC), and initiates a page walk. The hardware page walker fetches the necessary page table entries sequentially, starting from PGD, then PUD, PMD, and finally PTE.

The Replayer can tune the duration of the speculative execution by choosing whether victim's page table entries are either present or absent from the cache hierarchy and page walk cache (shown in the arrows above timeline ③ of Figure 1). The speculative instructions executing in the shadow of the page walk may leave some state in the cache subsystem and/or create contention for hardware structures in the core. This allows the Monitor to perform a noisy measurement of the secret data. At the end of the page walk, the hardware raises a page fault exception and squashes the speculative instructions in the pipeline.

The Replayer is then invoked to handle the page fault. The operation is shown in timeline ② of Figure 1. The Replayer chooses to keep the Present bit cleared. Timeline ④ of Figure 1 shows the actions of the Victim. In this case, after the Victim resumes and re-issues the replay handle, the whole process repeats. This process can be repeated as many times as desired to denoise and extract the secret information.

Monitoring Execution. The Monitor extracts the secret information from the Victim. Depending on the side channel being exploited, the monitor can cause resource contention in parallel to the victim execution, or prime and inspect the cache state in between replays. This is shown in timeline ⑤ of Figure 1, where the Monitor executes in parallel with the Victim's speculative execution.

Summary of a MicroScope Attack. The attack has 6 steps:

1. The Replayer identifies a replay handle and prepares the attack—e.g., by priming microarchitectural state.
2. When the Victim executes the replay handle, it suffers a TLB miss followed by a page walk. The time taken by this step can be over one thousand cycles, and can be tuned as per the requirements of the attack.
3. In the shadow of the page walk and until the page fault is serviced, the Victim continues to execute speculatively past the replay handle into the sensitive region, potentially until the ROB is full.
4. The Monitor can cause and measure contention on shared hardware resources during the Victim's speculative execution, or inspect the hardware state left by the Victim's speculative execution.
5. When the replay handle triggers a page fault, the Replayer gains control and can optionally leave the Present bit cleared in the PTE entry. This will induce another replay cycle that the Monitor can leverage to collect more information. Before the replay, the attacker may also prime the processor state for the next measurement.
6. When sufficient measurements have been gathered, the Replayer sets the Present bit in the PTE entry. This enables the Victim to make forward progress.

With these steps, MicroScope can denoise a side channel formed by, potentially, any instruction(s)—even ones that expose a secret only once in straight-line code. Further, the Replayer can then clear the Presence bit of a later replay handle in the application and proceed to monitor a later section of the code.

4 Simple Attack Examples

Figure 2 shows several examples of codes that present opportunities for MicroScope attacks. Each example showcases a different use case.

4.1 Single-Secret Attack

Figure 2a shows a simple code that has a single secret. Line 2 accesses a public address (i.e., known to the OS). This access

```
1 //public address
2 handle(pub_addr);
3 ...
4 transmit(secret);
5 ...
```

(a) Single secret.

```
1 for i in ...
2    handle(pub_addrA);
3    ...
4    transmit(secret[i]);
5    ...
6    pivot(pub_addrB);
7    ...
```

(b) Loop secret.

```
1 handle(pub_addrA);
2 if (secret)
3    transmit(pub_addrB)
4 else
5    transmit(pub_addrC)
```

(c) Control flow secret.

Figure 2: Simple examples of codes that present opportunities for MicroScope attacks.

is the replay handle. After a few other instructions, sensitive code at Line 4 processes some secret data. We call this computation the *transmit* computation of the Victim, using terminology from [6]. The transmit computation may leave some state in the cache or may use specific functional units that create observable contention. The goal of the adversary is to extract the secret information. The adversary can obtain it by using MicroScope to repeatedly perform steps (2)–(5) from Section 3.

4.2 Loop-Secret Attack

We now consider the scenario where we want to monitor a given instruction in different iterations of a loop. We call this case `Loop Secret`, and show an example in Figure 2b. In the code, the loop body has a replay handle and a transmit operation. In each iteration, the transmit operation accesses a different secret. The adversary wants to obtain the secrets of all the iterations. The challenging case is when the replay handle maps to the same physical data page in all the iterations.

This scenario highlights a common problem in side channel attacks: `secret[i]` and `secret[i+1]` may induce similar effects, making it hard to disambiguate between the two. For example, both secrets may be co-located in the same cache line, or induce similar pressure on the execution units. This fact severely impedes the ability to distinguish the two accesses.

MicroScope addresses this challenge by using a *second* memory instruction to move between the replay handles in different iterations. This second instruction is located after the transmit instruction in program order, and we call it the *Pivot* instruction. For example, in Figure 2b, the instruction at Line 6 can act as the pivot.

MicroScope uses the pivot as follows. After the adversary infers `secret[i]` and is ready to proceed to extract `secret[i+1]`, the adversary performs one additional action during step 6 in Section 3. Specifically, after setting the Present bit in the PTE entry for the replay handle, it clears the Present bit in the PTE entry for the pivot, and resumes the Victim's execution. As a result, all the Victim instructions before the pivot are retired, and a new page fault is incurred for the pivot.

When the Replayer is invoked to handle the pivot's page fault, it sets the Present bit for the pivot and clears the Present bit for the replay handle. When the Victim resumes execution, it retires all the instructions of the current iteration and proceeds to the next iteration, suffering a page fault in the replay handle. Steps 2-5 repeat again, enabling the monitoring of `secret[i+1]`. The process is repeated for all the iterations.

4.3 Control Flow Secret Attack

A final scenario that is commonly exploited using side channels is a secret-dependent branch condition. We call this case `Control Flow Secret`, and show an example in Figure 2c. In the code, the direction of the branch is determined by a secret, which the adversary wants to extract.

As shown in the figure, the adversary uses a replay handle before the branch, and a transmit operation in both paths out of the branch. The adversary can extract the direction taken by the branch using at least two different types of side channels.

First, if Lines 3 and 5 in Figure 2c access different cache lines, then the Monitor can perform a cache based side-channel attack to identify the cache line accessed, and deduce the branch direction. A second case is when the two paths out of the branch perform different computations. In this scenario the Monitor can apply pressure on the functional units and, by monitoring contention, deduce the operation that the code performs and, hence, the branch direction.

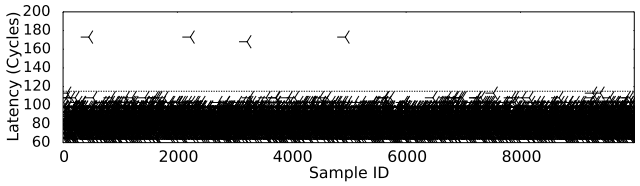
5 Evaluation

We validated microarchitectural replay attacks and MicroScope by denoising a notoriously noisy side channel: execution unit port contention [1]. For this attack we assume the SGX threat model. We use victim code similar to the one in Figure 2c, where one side of the branch executes two division operations, and the other side two multiplication operations. The Replayer forces the replay of the code. Concurrently, the Monitor executes a loop with one division operation in each iteration. We measure the time taken by each iteration of the Monitor loop. If the Victim executes the code with the two multiplications, the Monitor instructions execute fast and hence no contention is measured. Figure 3a shows the latency of each iteration of the Monitor. We see that all but 4 of the samples take less than 120 cycles, which we identify to be the contention threshold for our machine. If, instead, the Victim executes the code with the two divisions, the Monitor instructions execute slowly. Figure 3b shows the latency of each iteration. We see that 64 measurements are now above the threshold of 120 cycles. The two cases are clearly distinguishable. Overall, MicroScope is able to detect the presence or absence of two division instructions, outside of a program loop, using replays.

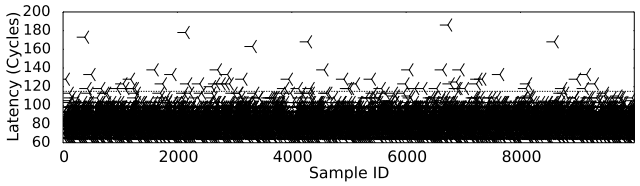
We further used MicroScope to perform a cache side-channel attack on AES [9]. MicroScope is able to extract the lines accessed in the AES tables without noise in a single run.

6 Generalizing Microarchitectural Replay Attacks

Figure 4 presents a generalized overview of MRAs. It has four parts: *replay handle*, *replayed code*, *side channel*, and *strategy*. In the attack that we described in the previous sections, the *replay handle* is a page fault-inducing load, the *replayed code* contains certain instructions that leak privacy, the *side channels* discussed are caches and functional unit ports, and the attacker's *strategy* is to unconditionally page fault until it



(a) Victim executes two multiply operations.



(b) Victim executes two division operations.

Figure 3: Latencies measured by performing a port contention attack.

has high confidence that it has extracted the secret. We now discuss how to create different attacks by changing some of these components.

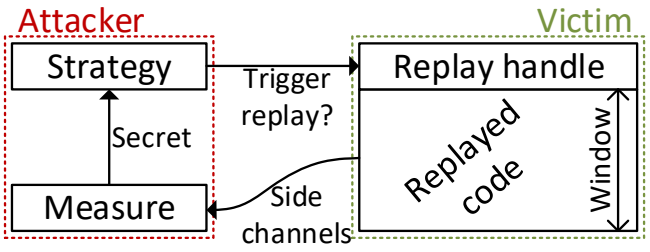


Figure 4: Generalized microarchitectural replay attacks.

Attacks on Program Integrity. MRAs can be used to violate program integrity, if the replayed instructions are non-deterministic. For example, suppose we replay the Intel true random number generator `RDRAND` instruction. If the attacker can read the return value through a side channel, it can selectively replay `RDRAND` until the value matches some desired criteria, effectively biasing the random number generation from the victim's perspective.[2]

Attacks Using Different Replay Handles. Microarchitectural replay attacks can exploit many different sources of replay beyond page faults. For example, they can exploit transaction aborts in transactional memory. Other instructions enable a limited number of replays. They include any event that can squash speculative execution [3], such as a branch misprediction or a load-to-store alias.

Amplifying Physical Side Channels. Microarchitectural replay attacks may also be an effective tool to amplify physical channels such as power and EM [8]. The idea is that a replay attack can turn non-repeating code into repeating code, which power meters, temperature sensors, and frequency analyzers can interpret better.

[2]This attack does not work on Intel machines due to a technicality. We verified that modulo this technicality, it should work.

7 Future Research Directions and Applications

The work presented in this paper can be extended in numerous ways. In this section, we describe some possible directions.

7.1 Potential Countermeasures Against MRAs

A future direction is to mitigate MRAs. We are actively working on this area. The root cause of MRAs is that individual dynamic instructions may execute more than once. This can be due to a variety of reasons (e.g., a page fault, a transaction abort, or a squash due to branch misprediction). Thus, it is clear that new, general security properties are required to comprehensively address these vulnerabilities.

The obvious defense against these attacks is for the hardware or the OS to insert a fence after each pipeline flush. However, corner cases such as multiple instructions in close proximity, individually causing replays, need to be considered. In these cases, even if a fence is introduced after every pipeline flush, the adversary can extract information from the resulting multiple replays.

MicroScope relies on speculative execution to replay Victim instructions. Therefore, a defense solution that holistically blocks side effects caused by speculative execution can effectively block MicroScope. However, existing defense solutions have limited defense coverage, introduce substantial performance overhead, or require costly hardware.

Page fault-oriented defense mechanisms could be effective to defeat MicroScope. Unfortunately, solutions that rely on Intel TSX are not sufficient, since TSX itself creates a new mechanism with which to create replays, through transaction aborts. Thus, we believe further research is needed before applying either of the above defenses to any variant of MRA.

7.2 MRAs as a Speculative Defense Mechanism

While MicroScope is presented as an attack, its operation can be used to improve security defenses. This is because it provides a window into what speculative attacks such as Spectre and Meltdown can do. For example, a Spectre attack on a given branch cannot affect subsequent instructions that are more than a ROB-long distance away from the branch dynamically. Consequently, MicroScope can be useful to bound the defenses that are inserted against speculative execution attacks.

Furthermore, MicroScope can be used to perform black-box analysis of microarchitectural structures such as the ROB, LSQs, and others. Controlled fine-grain microarchitectural replay capabilities can enable the reverse engineering of hardware structures. This approach can reveal timing, number of ports, and interconnect information and, more importantly, uncover previously-unknown behavior of hardware units under speculative execution. Such information is not only useful for discovering previously-unknown vulnerabilities, but it can further provide a foundation for defense mechanisms.

7.3 Parallel Application Debugging

The mechanism that enables our current MicroScope prototype, namely the capture and re-execution of a ROB-sized set of instructions, is a very useful primitive in software development and debugging. Capturing these instructions can provide insights into the program state in a way that no current tool can. Replaying these instructions deterministically may provide a way to debug hard-to-reproduce software bugs such as

data races. Small enhancements may enable single-stepping the code, or the ability to change the direction of branches on the fly. Overall, with a good interface, MicroScope may become a unique debugging tool for sequential and parallel code.

REFERENCES

[1] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit." IACR'18, 2018.

[2] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *S&P'15*, 2015.

[3] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security'19*, 2019.

[4] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *ASPLOS'18*.

[5] Intel, "Intel Software Guard Extensions Programming Reference." `https://software.intel.com/sites/default/files/329298-001.pdf`, 2013.

[6] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *MICRO'18*, 2018.

[7] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, May 2015.

[8] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "EDDIE: EM-Based Detection of Deviations in Program Execution," in *ISCA'17*, 2017.

[9] OpenSSL, "Open source cryptography and SSL/TLS toolkit." `www.openssl.org`, 2019.

[10] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *CCS'17*, 2017.

[11] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security'14*, 2014.

[12] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA," *IACR'16*, 2016.