Observing the L1 dTLB with JavaScript Code

Woosun Song

***Abstract*— Timer granularity reduction is a popular approach taken by browser developers in order to mitigate microarchitectural side channel attacks. However, leaky.page, a proof-of-concept for Spectre, proved that it is possible to overcome the absence of fine-grained timing primitives. They devised a novel technique to observe the L1 cache even with a coarse-grained timer by constructing a sequence of addresses that conditionally results in frequent misses. However, when leaky.page was ran on Intel i7-8086k, it failed to observe the L1 cache correctly.**

In this work, I first analyze the cause of failure of leaky.page's L1 Timer implementation. Through multiple experiments, I conclude that the primary cause of failure is high load processes running on other cores. Based on this analysis, I present an alternative to leaky.page's L1 Timer that leverages the L1 dTLB instead of the L1d cache. I construct a functioning proof of concept code written in JavaScript and show that the L1 dTLB based timer is not susceptible to cross-core interference. Additionally, the disparities of dTLB behavior among different microarchitectures imply the possibility of microarchitectural fingerprinting through the web.

I. Introduction

Multiple prior works[1][2] have showcased successful microarchitectural attacks on JavaScript and web browsers. As a response to these publications, mainstream browser vendors reduced the granularity of their embedded timer APIs. Schwarz et al.[3] presented methods that abuse browser components such as the `SharedArrayBuffer` in order to construct timers in nanosecond precision, but their methods are limited to Chromium based browsers. On the other hand, leaky.page, a proof-of-concept published by Google Project Zero, uses a more generic approach they call the tree-pLRU Oscillator. By using the tree-pLRU oscillator, an attacker can determine if a JavaScript callback has accessed a specific set within the L1 cache. The tree-pLRU oscillator does not depend on browser components and is only based on the assumption that the L1d cache uses tree-pLRU for its eviction algorithm.

However, when I tested the tree-pLRU Oscillator on Intel i7-8086k, Ubuntu 20.04 and Chrome 102, it failed to read the L1d side effects of the callback correctly. I analyzed the tree-pLRU Oscillator to determine the cause of failure, based on two hypotheses: First, the L1 eviction algorithm may not be tree-pLRU. Second, the co-locating hyperthread may interfere with the attacker's L1 cache. However, through experiments, both hypotheses were rejected and instead, additional observations lead to the conclusion that high-load processes running on other cores were the primary cause of error. I speculated that L3 evictions triggered by other cores were causing L1 evictions on the victim's core. Therefore, I decided to create an alternative implementation of leaky.page's L1 Timer that uses the L1 dTLB instead of the L1d cache The decision of using the L1 dTLB was based on the fact that it cannot be altered by processes running on other cores.

By using a heuristic algorithm I constructed an adversarial sequence similar to that of leaky.page's. By using this adversarial sequence, I created a proof-of-concept implementation with WebAssembly and showed that the L1 dTLB can be observed via timing differences in the order of multiple milliseconds. Also, the proof-of-concept works exclusively for Intel Skylake, which implies that it can be used for microarchitectural fingerprinting through the web.

II. Background

A. Timers in JavaScript

The timer with the highest precision that JavaScript provides is the `performance.now()` API. It returns the `DOMHighResolutionTimestamp`, which should guarantee a minimum of 5 μs precision by standard. However, to mitigate Spectre attacks, mainstream browser vendors reduced its granularity to varying degrees.

Browser	Timestamp Precision
Chrome	200μs + 200μs jitter
Safari	1ms
Firefox	2ms
Edge	200μs
Tor	100ms

Fig. 1. `performance.now()` precision of mainstream browsers after Spectre reception

B. Eviction Sets

Modern caches employ set associativity in microarchitectural caches. In a N-way set associative cache, there can be at most N entries that map to the same cache index. An **eviction set** is a set of addresses constituent of N different addresses that map to the same cache index. By accessing all of the addresses in an eviction set, an attacker can flush out all preexisting entries in that cache set. Thus, eviction sets construction is crucial in environments where the `clflush` instruction is unavailable.

In order to construct an eviction set, the mapping function between the entry address and the cache index must be known. For Intel Skylake, the L1 cache is in total 4KiB, where each entry is 64 bytes and there are 16 8-way associative sets. The mapping function is linear: the cache index is equal to the mod-16 of address[6:12]. Because the cache index is contained within a page offset, the attacker does not require knowledge of physical memory layout in order to construct an eviction set.

For the L3 cache, the bits passed to the mapping function include bits that determine the page number. Thus, prior knowledge about physical memory layout is required for constructing an eviction set. Also, the L3 cache is split into multiple slices and distributed via an undocumented hash function. Thus, the straightforward approach used for the L1 cache is infeasible for L3. Vila et al.[4] proposed a $O(n^2)$ time algorithm for constructing eviction sets using reduction from a large set of addresses.

C. TLB Organization of Intel Skylake

Gras et al.[5] reverse engineered the organization of TLBs existing in various microarchitectures.

According to their discoveries, Intel Skylake has a 2-level hierarchy for TLBs, where the 2-level TLBs are all contained within a single CPU core. The first level, the L1 TLB, is divided into L1 iTLB and L1 dTLB which stores page translations for code pages and data pages respectively. The second level, the L2 TLB is called the sTLB (shared TLB) because it embraces both code and data pages.

Additionally, they showed that the L1 dTLB is 4-way associative with 16 sets. The mapping function between the page number and the index is simply a linear one, where the TLB index is equal to the mod-16 of the page number. This simplicity allows the construction of a L1 dTLB eviction set to be straightforward: any addresses whose distance is a multiple of 0x10000 maps to the same set.

They did not discuss the replacement policy of the TLBs.

D. Intel PMC

Intel PMC(Performance Monitor Counter) is a hardware feature implemented to count microarchitectural events with minimal overhead. Its usage is described in Intel SDM Volume 3B. Its intended use is to allow performance optimization and high-precision benchmarking. The linux `perf` tool makes heavy use of Intel PMC.

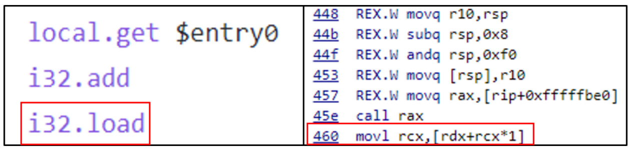
For Intel CoffeeLake, there are a total of 12 counters, where 4 of them are FF(Fixed-Function) counters and the remaining 8 are GP(General-Purpose) counters. FF counters count predefined events, while GP counters are programmable: the event to catch is specified by the user. Programming the GP counters require the `rdmsr` and `wrmsr` which requires ring0 privileges.

Once a counter is programmed, it can be read in either two ways. First, it can be read by reading from a dedicated MSR that holds its value. Second, it can be read using the `rdpmc` instruction. The `rdpmc` instruction takes `rcx` as an index to the counter. Executing the `rdpmc` instruction in userspace requires modifying the `cr4` register and is only allowed in ring0 by default.

E. WebAssembly

WebAssembly is a binary instruction format for web browsers. It was designed so that compiled langauges, such as C, C++, or Rust can be ran on browsers. It is also possible to write WebAssembly code in `wat`, which is a human readable disassembly of WebAssembly's binary format. WebAssembly binary code can be compiled to native code by passing the bytes to JavaScript's `WebAssembly.Module` function.

WebAssembly has support for typed load/store instructions. The address space of WebAssembly starts from 0x0 and is partitioned into 64KiB pages. However, for a single load/store WebAssembly instruction, multiple load/store instructions are inserted in the corresponding native code. These instructions are generated in order to prevent out-of-bounds memory read-/write, runtime type checks, and deoptimization(the process of bailing out the interpreter in JIT code). In an attacker's point of view, such instructions are a source of noise because they cause inadvertent memory accesses.



```
local.get $entry0
i32.add
i32.load
```

```
448  REX.W movq r10,rsp
44b  REX.W subq rsp,0x8
44f  REX.W andq rsp,0xf0
453  REX.W movq [rsp],r10
457  REX.W movq rax,[rip+0xfffffbe0]
45e  call rax
460  movl rcx,[rdx+rcx*1]
```

Fig. 2. Assembly code generated from the WASM `i32.load` instruction. The typed load instruction is lowered to a x64 `movl` instruction in line 460. Assembly code in 448 to 45e are boilerplate code that causes inadvertent memory accesses.

F. tree-pLRU Oscillator

The tree-pLRU Oscillator is a technique used in leaky.page to observe the contents of the L1d cache with a coarse-grained timer.

An attacker first determines the cache set index he wishes to observe, denoted by I. Afterwards, he constructs an eviction set for L1 consisting of addresses $E = \{E_1, E_2, \cdots E_n\}$. For Intel Skylake, which has 16 8-way associative sets, $0 \leq I < 16$ and $n = 8$. An additional address X, which also maps to set I, is prepared as well. After constructing an eviction set, the attacker fills cache set I with elements in E.

Afterwards, the attacker executes the speculative gadget. The speculative gadget is a JIT compiled JavaScript callback. The attacker crafts the speculative gadget so that it accesses X if and only if the secret bit is 1. Therefore, as depicted in Figure.3, the cache state at this point will be dependent on the secret bit.

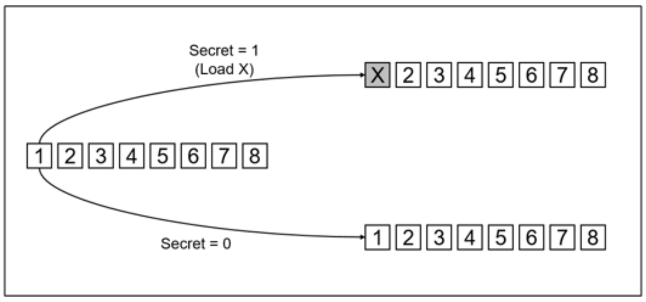


Fig. 3. Cache state after speculative gadget execution. If secret data is 1, X is accessed, causing one of the elements in the eviction set ($\{1, 2, \cdots, 8\}$) to be replaced by X.

Finally, the attacker observes the L1 cache by measuring the execution time for an adversarial load sequence. An adversarial load sequence is a sequence consisting of entries in E such that its behavior depends on the presence of X. For example, the length 1 sequence $\{1\}$ is an adversarial sequence because it causes a miss only if X was present, as depicted in

Figure.4. However, it is not long enough to cause a large timing difference. Thus, a sequence with sufficient length is necessitated.

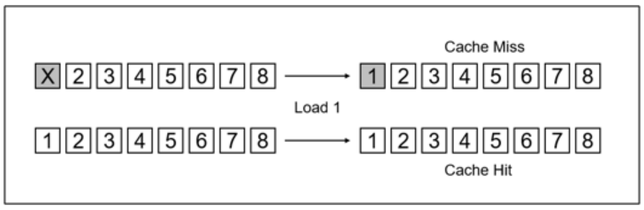


Fig. 4. The different effects of load 1 depending on the presence of X. If X was in the cache, load 1 causes a cache miss. On the other hand, if X was not in the cache, it results in a hit.

An adversarial sequence must satisfy an invariant: X must be present in the cache during the entire sequence execution. If X is evicted, set I will be filled up exclusively with entries in E at some point, and all subsequent loads will result in a hit. Thus, to preserve X in set I, the attacker must prevent the replacement policy from marking X as the least frequently used element.

In tree-pLRU, a binary search tree points to the next-to-be-evicted element. When an entry within the cache is accessed, the tree is rotated so that edges point away from that entry. Specifically, all edges in the path from the root to the target entry are negated. Thus, if a sibling leaf of some entry e is accessed, e will be safe from eviction for at least the next one subsequent load. Under this assumption, the authors of leaky.page constructed a sequence that preserves X in the cache until the end, resulting in consistent cache misses.

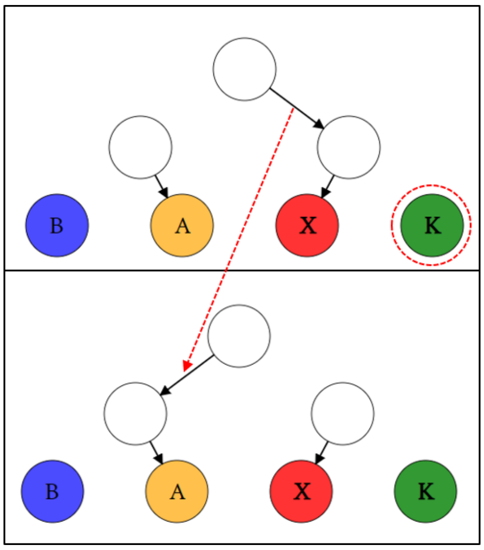


Fig. 5. Strategy for Preventing X from Being Evicted. When X's sibling leaf K, the edge pointing to X's parent is negated, pointing away from X and K.

Their proposed sequence is as follows:

$$\{E_0, E_1, E_2, E_4, E_3, E_5, E_6, E_4, E_7, E_0,\\ E_1, E_4, E_2, E_3, E_5, E_4, E_6, E_7, E_0, E_4,\\ E_1, E_2, E_3, E_4, E_5, E_6, E_7, E_4\} \times N$$

By configuring the number N, an attacker can amplify or decrease the timing difference. Increasing N is a trade-off between leakage accuracy and leakage bandwidth. Also, N must be at least large enough so that the timing difference is larger than `performance.now()`'s precision.

III. Threat Model

A. Attacker

The attacker is assumed to be a JavaScript process that can run arbitrary JavaScript code. This condition imposes restrictions that does not exist for attackers running native code. First, the attacker cannot execute the `clflush` instruction. Second, the `rdtsc` instruction is unavailable, and attackers are left with coarse-grained timing primitives.

An attacker is capable of allocating page aligned memory via the `WebAssembly.Memory` constructor. Also, an attacker can freely load/store to page aligned memory. Due to these primitives, it becomes possible to construct a L1 cache eviction set in a straightforward manner, because for Intel Skylake the L1 cache is contained within a 4KiB page.

B. Victim

The victim is the same process as the attacker. Thus, the victim and the attacker share the CPU core and memory. This threat model is atypical in the sense that side channel attacks usually aim to spy on processes running on other CPU threads or cores. Such threat models exist when speculative execution attacks are used to break security boundaries set by software. A similar threat model exists for attacking the kernel[6] from userspace by using Spectre.

C. What the Attacker Gains

With Spectre-v1, the attacker can freely read the memory of the victim. If the attacker is capable of executing native code, attacking oneself with Spectre-v1 is a meaningless task because it can read its own memory without a side channel. However, in the case of JavaScript, reading one's own address space is a critical security violation.

Via out-of-bounds read of JavaScript arrays, the attacker can leak pointers. Attackers can use such leakages to bypass ASLR in conjunction with other memory corruption attacks. Furthermore, attackers can use Spectre-v1 to read `StructureID` in JavaScriptCore, the JavaScript engine used in Safari. `StructureID` is a randomized magic value embedded in the header of in-memory JavaScript objects. It exists to mitigate in-memory object corruption exploits.

In conclusion, the implications of Spectre-v1 on JavaScript is critical, and thus the threat model chosen for this work is practical.

IV. Topic Statement and Hypothesis

A. Failure of Leaky.page's L1 Timer

Leaky.page provides a separate page for showing the timing distribution of the tree-pLRU Oscillator.

In Project Zero's demo video[7], the plot showed a clear distinction between the two cases(when the gadget alters the cache and when it doesn't), where the green KDE plot represents the timing distribution for when the gadget altered the cache, and the purple KDE plot represents the timing distribution for when the gadget did not alter the cache.

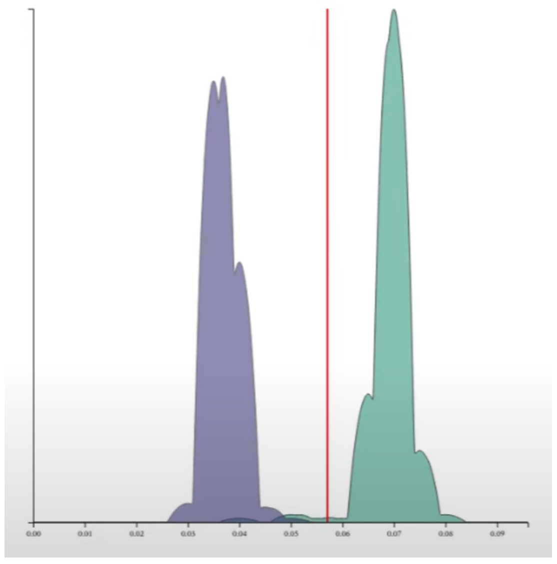


Fig. 6. Timing Diagram Obtained from Leaky.page's Official Demo Video, $N = 2000$.

The results obtained on Intel i7-8086k @4.00GHz, Chrome 102, Ubuntu 20.04 is shown in Figure.7.

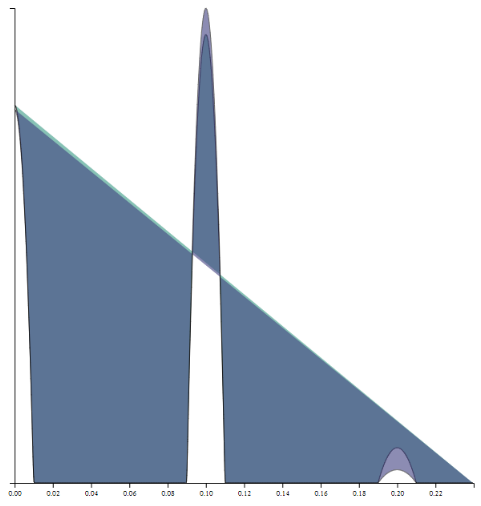


Fig. 7. Timing Diagram Obtained on i7-8086k @ 4.00GHz, Chrome 102, Ubuntu 20.0 with $N = 2000$

The timing distribution obtained on the authors' computer show no distinction between green and purple. In other words, the L1 Timer does not have any capability of observing the L1d cache under the given parameters. This may be due to additional timer granularity reduction that occurred between the release of Project Zero's demo video and current time. Thus, I increased N(discussed previously in II-F). so that the timing difference is at least larger than the timer precision. The results are shown in 8.

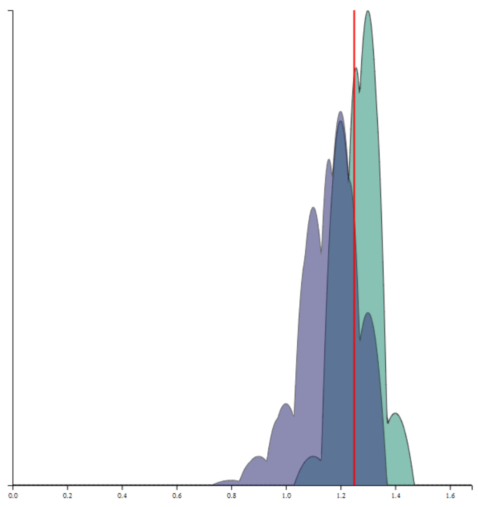


Fig. 8. Timing Diagram Obtained on i7-8086k @ 4.00GHz, Chrome 102, Ubuntu 20.0 with $N = 40000$

It can be seen that there is some distinction between green and purple, as each distribution resembles a Bell-curve where the mean of the green distribution is higher than that of the purple distribution. However, the large overlapping region of the two distributions imply high rates of false negatives and positives. Thus, I concluded that using the tree-pLRU Oscillator without any improvements is impractical.

B. *Topic Statement*

- Analyze the cause of failure of leaky.page's tree-pLRU Oscillator method.
- Devise an improvement for leaky.page's tree-pLRU Oscillator method so that noise is reduced.

C. *Hypothesis*

1) *What is the cause of failure of the tree-pLRU Oscillator?:*

- The tree-pLRU assumption is invalid. In other words, the L1d cache on Intel Skylake does not employ tree-pLRU and uses some other replacement policy.
  $\rightarrow$ Reverse engineer the replacement policy of the L1d cache, and construct a new adversarial sequence targeting that replacement policy.

- Cache penalties due to SMT causes noise. If a co-locating thread takes over control in the middle of sequence execution, the cache set the sequence was operating on may be altered, and the invariant that X is kept in the cache may break.
  $\rightarrow$ Devise methods so that the attacker thread is co-located with an idle thread (another attacker created thread that does not incur critical cache penalties) with high probability.

- Processes running on other cores may cause noise. Due to the inclusiveness of the cache orgnization in Intel CPUs,

if an entry is evicted from the L3 cache, it is evicted from the L1 cache as well. Thus, a high-load process running on other cores can incur cache evictions on the attacker's core by accessing the L3 cache.
$\rightarrow$ The memory cache is a component that exhibits some cross-core interference, due to inclusiveness. Construct a timer using CPU components that have no cross-core interference.

V. Methods

A. *Benchmarking*

Let $S = \{S_1, S_2, \cdots S_n\}$ denote the sequence of loads. We first construct a primitive that determines whether the load of S_n causes a L1d cache miss or a hit. Assume that for any S_i, it maps to set 0 of the L1d cache. Thus, its page offset must be in the range [0,0x40].

1) Baseline: Load all the addresses from $S_1 \cdots S_{n-1}$, and measure the timing of the load of S_n. There are multiple caveats.

- Data structures holding $\{S_1 \cdots S_{n-1}\}$ must not affect the cache set of interest. For example, if a 1-D `void *` array is used to hold the addresses, lookups to the array may affect the cache set of interest if n is large to span over a page. Also, even if n is not large, sequential access to an array may trigger stride prefetching. Thus, I used singly-linked lists to hold $\{S_1 \cdots S_{n-1}\}$. Also, I created a wrapper over `malloc` so that all pointers whose page offset is in the range [0,0x40] are rejected. By doing so, it prevents the list entry structures from being mapped to set 0 of the L1d cache.
- Loads must be strictly ordered, because OoO execution may reorder the loads, causing the LRU state of the cache to be updated in a different order. I used two methods to impose strict ordering between loads. First, I imposed a data dependency between loads. Not only does this allow loads to be serialized, but it also effectively prevents compiler optimizations from eliminating the loads. Second, I inserted the `lfence` instruction between all loads.

```
char load(char *ptr1, char *ptr2) {
    register char trash;
    trash = 0;
    trash = ptr1[trash];
    _mm_lfence();
    trash = ptr2[trash];
    _mm_lfence();
    return trash;
}
```

Fig. 9. Code snippet of two loads serialized by imposing data dependency and inserting lfence

- Local stack variables may affect cache state. Thus, I eliminated usages of all local stack variables by using the `register` prefix for all local variables.

2) Using Intel PMC: The timing of the last load proved to be uninformative when deciding if it was a cache miss or a hit. In other words, the timing distribution for a L1 cache miss and a hit had low disambiguation. Xiong et al.[8] state that serialization noise due to `lfence` is a cause of noise when measuring load timings. Thus, they use a pointer chasing technique to reduce the proportion of timing caused by serialization. However, in this work, instead of using pointer chasing, Intel PMC was used to count the number of cache misses explicitly. The GP0 counter is programmed to catch all `l1d.replacement` events. Afterwards the difference of the GP0 counter after and before the load is measured. If the difference was a 1, it implies that a L1d cache miss occurred.

```
void load(char *ptr) {
    register uint64_t ctr;
    register char trash;
    trash = 0;
    ctr = rdpmc(0);
    trash = ptr[trash];
    ctr = rdpmc(0) - ctr;
    return ctr | trash;
}
```

Fig. 10. Usage of Intel PMC to explicitly count L1d cache misses

B. *Model of the Cache Replacement Policy*

A DFA consists of states and state transitions. I will show how the replacement policy of a cache can be modeled to DFA states and transitions.

1) State: A single set of the L1d cache can be modeled as a tuple (S_{DATA}, S_{LRU}) where S_{DATA} represents the data in the cache and S_{LRU} represents the internal LRU state. Because Intel Skylake has a 8-way associative L1 cache, S_{DATA} can be modeled as a set such that $|S_{DATA}| \leq 8$. If we assume a sequence of addresses constituent of 9 addresses (8 for the eviction set + 1 for X), the number of configurations for S_{DATA} is $2^9 - 1$. The internal LRU state is used to keep track of the least recently used entry. If tree-pLRU is used, the internal LRU state is a binary search tree. Because I do not make any assumptions about the replacement policy, the number of states that S_{LRU} can take is arbitrary. Thus, the number of states for a L1d cache set is $511 \times |S_{LRU}|$.

2) State Transition: State transitions occur when a memory load occurs. There are two types of state transitions: cache misses and cache hits. Cache misses are state transitions that result in the change of S_{DATA}. Although cache hits do not alter S_{DATA} it may alter S_{LRU}. From now on, I will denote cache misses as red edges and cache hits as black edges.

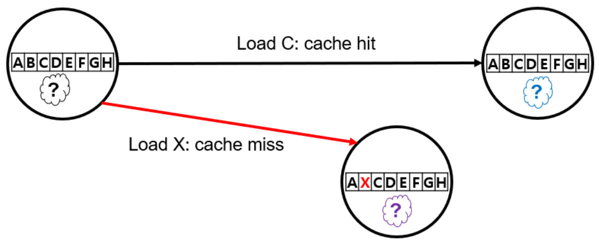


Fig. 11. Modeling of the L1d cache into a DFA. Even if a cache hit occurs, S_{LRU} may be altered, which is denoted by the change of color of the cloud in the diagram.

C. *Validating the tree-pLRU Assumption*

By using the primitives described in V-A, we can use the algorithm below to determine how many misses occur in leaky.page's proposed sequence.

Algorithm 1 Color-Edges($S_1, S_2 \cdots, S_n$)

```
1: colors ← $[0 \times n]$
2: for $i = 1$ to $n$ do
3:     if IsLastEntryMiss($S_1, \cdots, S_i$) then
4:         colors[$i$] ← red
5:     else
6:         colors[$i$] ← black
7:     end if
8: end for
9: print($t$)
```

The number of red entries in the `colors` array is equal to the number of cache misses that occurred during the entire load sequence. The reason why the number of cache misses is not measured collectively for an entire load sequence is because loads irrelevant to the sequence will be counted as well. For example, lookups to the list that stores the pointers will increment the counter as well.

I test a sequence with $N = 100$. If the number of misses is commensurate to the length of the sequence, the tree-pLRU assumption can be accepted. If the tree-pLRU assumption is invalid, misses will only occur for the first few loads and all subsequent loads will result in hits.

D. *Testing the Effects of SMT*

Testing if SMT introduces noise is straightforward: disable SMT and compare the outcome. If there are no differences, the hypothesis that SMT is a source of noise can be rejected.

E. *Constructing an Adversarial Sequence for the L1 dTLB*

The DFA (discussed in V-B) of the replacement policy of the L1 dTLB is useful when constructing an adversarial sequence. The DFA for the L1 dTLB can be obtained in a similar manner to the process for obtaining the DFA for the L1d cache. Instead of capturing `l1d.replacement` events, `dtlb_load_misses.stlb_hit` events are captured. Also, the eviction set is constructed differently.

Now, I describe a heuristic to discover adversarial sequences. An adversarial sequence has a one-to-one correspondence with a path in the DFA. Thus, the task of finding an adversarial sequence is equivalent to finding a path in the DFA with a sufficient number of red edges.

Due to its vast size, it is infeasible to obtain a representation of the entire DFA. Thus, I use the approach of 'n-depth traversal'. We only collect DFA information until the root has a height of n. In Figure.12, $n = 2$.

After a n depth traversal is made, a subgraph of the DFA is given. The subgraph is a tree with height n. The algorithm then collects all paths from the root to a leaf that has at least one read edge. For the first n depth traversal, a red edge is guaranteed to exist.

Afterwards, the algorithm is recursively applied on the leaves of the selected paths. If there are no red edges in the n depth traversal of the leaves, the algorithm backtracks.

In Figure.12, the algorithm is described step by step. The initial state is A. There are a total of 2 paths from the root node to the leaf node that contains at least one red edge. Thus, the heuristic recursively traverses with D and E. The depth-2 traversal starting from E does not have any red edges. Thus, the path A-C-E is discarded. The depth-2 traversal starting from D reveals 3 paths that has a red edge. Thus, the 3 paths are collected and the algorithm runs recursively on G, H, and I. By running this algorithm, the sequence A-C-D-F-H was obtained. By increasing the traversal depth and the number of recursions, it is possible to create an arbitrary length sequence.

However, in a practical point of view the adversarial sequence is likely to have a cycle. Thus, it is reasonable to terminate the heuristic if a repeated subsequence is found.

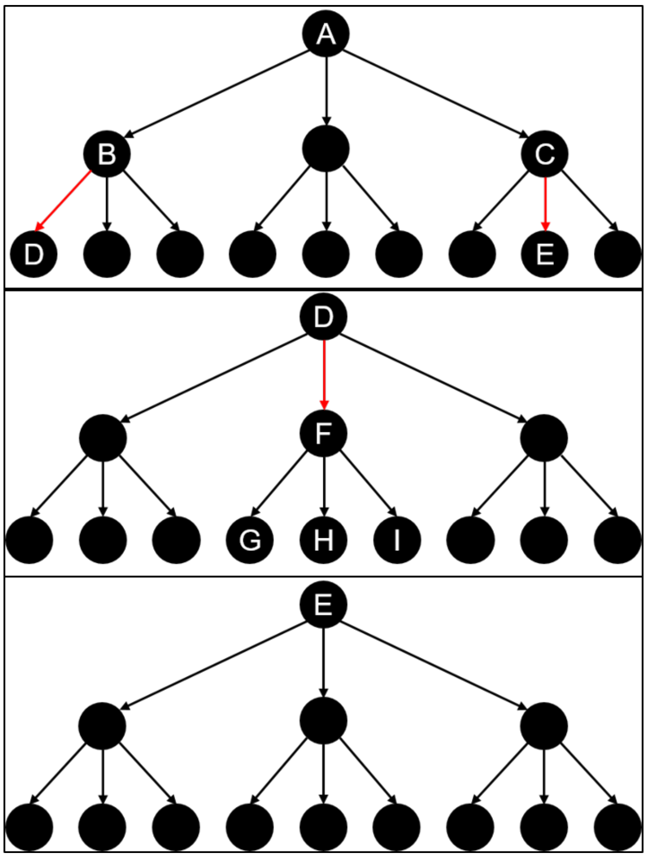


Fig. 12. Heuristic to discover adversarial sequences given a cache description in the form of a DFA.

VI. Results

A. *Correctness of the tree-pLRU Assumption*

The results of V-C lead to the conclusion that the tree-pLRU assumption is valid. In other words, the adversarial sequence proposed by leaky.page results in consistent misses under ideal conditions.

B. *Effects of SMT*

The tree-pLRU Oscillator did not function properly even with SMT disabled. Thus, SMT is not a significant source of noise. This conclusion is counterintuitive, as SMT is expected to incur periodic cache penalties to the victim thread.

C. *Cross-Core Interference*

By turning off background processes running default on Ubuntu, I could make leaky.page's tree-pLRU oscillator function properly. Thus, it can be concluded that processes running on other cores is the primary cause of noise.

Although the L1d cache is a per-core data structure, it is possible for processes running on other cores to affect its state via inclusiveness. If a process triggers a L3 eviction, the corresponding L1 entry will be evicted as well. If the evicted L3 entry belonged to the victim thread, its L1 cache will be subject to eviction consequently.

To mitigate effects of processes running on other cores, I decided to leverage components with no cross-core interference. The L1 dTLB is a component that satisfies this property. There are no global TLBs; in other words, there does not exist a concept analogous to the LLC. Thus, a process running on another core cannot affect the TLB of the victim thread's core.

Also, using the L1 dTLB gives the benefit of a higher timing difference between hits and misses. The L1 dTLB miss penalty is approximately 9-10 cycles, whereas the L1d cache miss penalty is 4-5 cycles. If the L1 dTLB miss results in a PWC lookup or a page walk, it results in an even larger timing difference.

D. *Constructing an Adversarial Sequence for the L1 dTLB*

Let E denote an eviction set for the L1 dTLB. An example of a functioning eviction set is as follows:

$$E = \{0x10000, 0x20040, 0x30080, 0x300c0\}$$

The page offsets are set so that all of the entries are mapped to a different L1d cache set. An adversarial sequence consisting of E_1, E_2, E_3, E_4 is as follows:

$$\begin{aligned} S = \{&E_3, E_1, E_0, E_2\} + \\ &\{E_0, E_3, E_0, E_1, E_0, E_2, E_0, E_3, E_0, E_1, E_0, E_2\} \times N \end{aligned}$$

The total length of the sequence is adjustable via configuring N. For example, one can create a sequence such that length is at least 10000 by setting N to 833, for $10000 \geq 4 + 833 \times 12$.

VII. JavaScript Proof-of-Concept

A. *Proof-of-Concept with C*

Writing a PoC with JavaScript+WebAssembly is harder than writing a PoC with C, because as discussed in II-E, native code created from WebAssebmly contains boilerplate code which introduces unsolicited memory accesses. Thus, as a preliminary step I first wrote a PoC with C. To impose JavaScript-like constraints, I used the `gettimeofday` system call to measure time, which has a precision of 1ms. Also, I simulated the speculative gadget by inserting load X after the cache initialization and before the adversarial sequence beginning. Thus, if load X is included, it can be considered as a simulation of such that the secret bit is 1. On the other hand, if load X is not inserted, it is a simulation of the secret bit being 0. The results are shown in Figure.13.

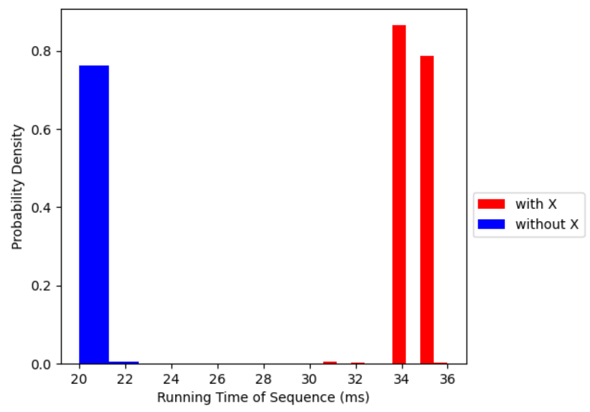


Fig. 13. Results of PoC written in C

The timings for when load X is included is larger than when load X is not included. The difference is approximately 10ms, which can be considered practical regarding the timer precision of `performance.now()` in mainstream browsers (Figure1).

The timing distribution for a load with X present is wider than the timing distribution for a load without X. This can be explained by the varying degrees of L1 dTLB miss latency. When the L1 dTLB results in a miss, the CPU will try to look up the entry in the following order: L2 sTLB, PWC, DRAM (page walk). Thus, the L1 dTLB miss latency is not constant and is a random variable such that the minimum value is the sTLB latency and the maximum value is the page walk latency.

B. *Proof-of-Concept with JavaScript*

To minimize the introduction of boilerplate code, I wrote the PoC with `wat` instead of writing it in C and compiling it to WebAssembly. Figure. is a screenshot of a webpage that hosts the PoC. It displays the timing distribution in a graphical fashion via `d3.js`, a JavaScript library for plotting chars and graphs on HTML.

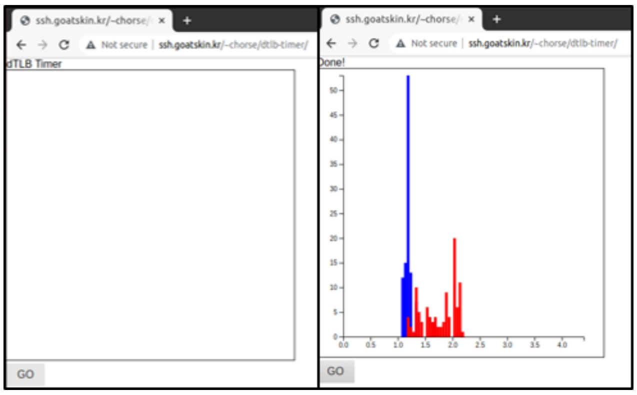
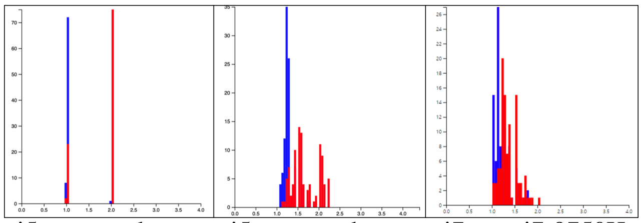

ssh.goatskin.kr/~chorse/
Not secure | ssh.goatskin.kr/~chorse/dtlb-timer/
dTLB Timer
GO
ssh.goatskin.kr/~chorse/
Not secure | ssh.goatskin.kr/~chorse/dtlb-timer/
Done!
GO

Fig. 14. Webpage hosting PoC. The webpage displays the timing differences in a graphical fashion.

The red distribution represents the timings for when load X is inserted. The blue distribution represetns the timings for when load X is not inserted.

I distributed the link to people with various devices and collected the results by asking them to provide a screenshot. In conclusion, the PoC showed a clear timing distribution difference on Intel Skylake CPUs. Otherwise, it failed to observe the L1 dTLB correctly.

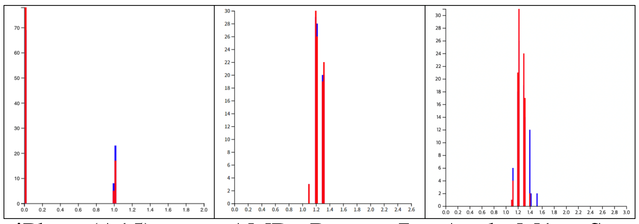
1) Intel Skylake: For all Skylake CPUs tested, the timing distribution showed a clear distinction between the two cases. The shape of the distribution differed by browser. For example, Safari has a timer precision of 1ms. Thus, it results in a discrete distribution rather than a continuous one.



i5 quad-core @2.3GHz + Safari | i5 quad-core @2.3GHz + Chrome | i7 i7-9750H @2.6GHz + Chrome

Fig. 15. PoC results on Intel Skylake CPUs

2) Non Intel Skylake: For all non Skylake CPUs tested, the timing distribution showed no differences between the two cases. It implies that the L1 dTLB organization, including the number of sets, set associativity, and replacement policy is unique to Skylake.



iPhone(A15) + Safari | AMD Ryzen 7 1700 + Chrome | Apple M1 + Safari

Fig. 16. PoC results on non-Skylake CPUs

VIII. Implications and Contributions

A. Spectre using the L1 dTLB

The L1 dTLB timer presented in this work can be used for Spectre attacks on JavaScript engines.

B. Microarchitectural Fingerprinting Through the Web

The disparities between PoC results on Intel Skylake CPUs and non Intel Skylake CPUs imply that the L1 dTLB organization is unique to Skylake. This observation can be generalized to other microarchitectures. An attacker can prepare a sequence that is only adversarial to a certain microarchitecture, and measure its running time. If this process is enumerated for all commercial microarchitectures, it would be possible for a malicious website to fingerprint its clients' microarchitecture.

C. Modeling of the Cache

This work presents a generic method to reverse engineer the replacement policy of microarchitectural caches by modeling it to a DFA. While leaky.page constructed an adversarial sequence based on prior knowledge that the L1d cache employs tree-pLRU, the methods used for this work only makes the assumption that the replacement policy is deterministic.

IX. Limitations

A. Disadvantages of Leveraging the L1 dTLB

When compared to side channels based on the L1d cache, the L1 dTLB has a disadvantage that its observation granularity is 4KiB. However, this is not a significant limitation when conducting Spectre attacks on JavaScript, because the speculative gadget is written by the attacker. Thus, the manner in which it leaves traces is determined by the attacker and observation granularity is not a limitation.

B. Applications to Non-Intel Architectures

The benchmark relies heavily on Intel PMC, which was used in order to avoid the usage of pointer chasing. Thus, it would be desirable to improve the benchmark by using pointer chasing so that it becomes possible to reliably distinguish a cache miss from a hit only with `rdtsc`.

C. Considerations on SMT Noise

In VI-B, I conclude that co-locating threads are not a significant source of noise. However, this is a shallow conclusion because it is only based on empirical results and lacks proper reasoning.

References

[1] Artur Janc Stephen Rotteger. *Leaky.page*. 2020. URL: `https://leaky.page` (visited on 05/18/2022).

[2] Ayush Agarwal et al. "Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution". In: *43rd IEEE Symposium on Security and Privacy (S&P'22)*. 2022.

[3] Michael Schwarz et al. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript". In: *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*. Ed. by Aggelos Kiayias. Vol. 10322. Lecture Notes in Computer Science. Springer, 2017, pp. 247–267. DOI: `10.1007/978-3-319-70972-7\_13`. URL: `https://doi.org/10.1007/978-3-319-70972-7\_13`.

[4] Pepe Vila, Boris Köpf, and Jose Morales. "Theory and Practice of Finding Eviction Sets". In: *40th IEEE Symposium on Security and Privacy (S& P '19)*. IEEE, 2019, pp. 695–710.

[5] Ben Gras et al. "TLBleed: When Protecting Your CPU Caches is not Enough". In: *Black Hat USA*. Aug. 2018. URL: `Slides=https://i.blackhat.com/us-18/Thu-August-9/us-18-Gras-TLBleed-When-Protecting-Your-CPU-Caches-is-Not-Enough.pdfWeb=https://vusec.net/projects/tlbleed`.

[6] Jann Horn. 2018. URL: `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`.

[7] Stephen Rotteger. *Spectre JS demo*. 2021. URL: `https://www.youtube.com/watch?v=V_9cQP60ZGI`.

[8] Wenjie Xiong, Stefan Katzenbeisser, and Jakub Szefer. "Leaking Information Through Cache LRU States in Commercial Processors and Secure Caches". In: *IEEE Transactions on Computers* 70.4 (2021), pp. 511–523. DOI: `10.1109/TC.2021.3059531`.