SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon

Jason Kim
Georgia Tech
nosajmik@gatech.edu

Daniel Genkin
Georgia Tech
genkin@gatech.edu

Yuval Yarom
Ruhr University Bochum
yuval.yarom@rub.de

***Abstract*—Since Spectre's initial disclosure in 2018, the difficulty of mitigating speculative execution attacks completely in hardware has led to the proliferation of several new variants and attack surfaces in the past six years. Most of the progeny build on top of the original Spectre attack's key insight, namely that CPUs can execute the wrong control flow transiently and disclose secrets through side-channel traces when attempting to alleviate control hazards, such as conditional or indirect branches and return statements.**

In this paper we go beyond (speculatively) affecting control flow, and present a new data speculation primitive that stems from microarchitectural optimizations designed to alleviate data hazards. More specifically, we show that Apple CPUs are equipped with a Load Address Predictor (LAP). The LAP monitors past addresses from the same load instruction to speculatively load a predicted address, which may incorrectly point to secrets at rest (i.e., never architecturally read by the CPU). Once the secret is retrieved, the LAP allows for a large speculation window that suffices for an adversary to compute on the secret, such as leaking it over a covert channel.

We demonstrate the LAP's presence on recent Apple CPUs, such as the M2, A15, and newer models. We then evaluate the LAP's implications on security by showing its capabilities to read out-of-bounds, speculatively invoke rogue functions, break ASLR, and compromise the Safari web browser. Here, we leverage the LAP to disclose sensitive cross-site data (such as inbox content from Gmail) to a remote web-based adversary.

1. Introduction

From the turn of the decade, there is a pivotal change in the desktop computing market. Whereas x86 CPUs from Intel and AMD had dominated the heavyweight CPU market in the past, new lineages from Apple and Qualcomm using the ARM architecture are emerging in market share. Over the past few years, they have become formidable competitors to the x86-based landscape, bringing equivalent or often better performance at a fraction of the power consumption.

Another change brought about in recent computer technology is the rise of transient execution attacks (e.g., Spectre [33] and Meltdown [36]), which exploit speculative and out-of-order execution to leak information across security domains [16, 31, 33, 34, 36, 40, 54, 61, 62, 64, 65, 66, 70, 71]. That is, nearly all currently known Spectre variants rely on the CPU speculating on control hazards, encompassing if-statements, indirect jumps, returns, and loops, speculatively diverting the CPU's control flow into code gadgets benefitting the attacker. Finally, Spectre has impacted nearly all modern heavyweight CPU designs, including recent generations released by Intel, AMD, Apple and other vendors.

Next, to further increase performance, computer architects proposed speculating on data flows in addition to control flow, aiming to alleviate data hazards encountered during software execution [35, 47, 48, 56, 57, 69]. Here, rather then waiting for the hazard to resolve, the CPU attempts to predict the value of the data being accessed and proceeds execution of younger instructions using the predicted value. While some forms of such data speculation have been observed in the wild, mainly through data-dependent hardware prefetchers [18, 67], store-to-load forwarding prediction [26], or floating point issues [52, 58], much remains to be done to fully characterize all forms of data speculation present on modern CPU platforms. Thus, in this paper, we ask the following questions.

Are there additional data speculation mechanisms present on modern CPUs? If so, what are the security implications of such speculation?

1.1. Our Contributions

In this paper, we answer the first question in the affirmative. More specifically, to the best of our knowledge, we document the first existence in the wild of a Load Address Predictor (LAP), where the CPU predicts addresses of load instructions based on prior addresses it had observed. In case the predicted address is cached, the CPU speculatively loads from the predicted address as opposed to the address originally intended by the program. Once the value from the predicted address is fetched, the CPU computes on it transiently using arbitrary instructions that are downstream in program code. Moreover, the speculation window is large enough for the value to be transmitted using microarchitectural side channels. Finally, we show that the LAP is present on Apple's M2-M4 CPUs for Macs and iPads, and the closely related A15-A17 CPUs for iPhones.

Next, tackling the second question, we show the LAP is exploitable and results in grave security consequences. We first introduce Spectre-LAP, a transient 64-bit out-of-bounds read primitive that founds a new lineage of speculative execution attacks on data flows. Building on this, we introduce

SLAP, an end-to-end attack on Apple's Safari web browser capable of disclosing email content and browsing behavior from an arbitrary target webpage to a remote adversary.

Discovering and Reverse Engineering the LAP. We start by analyzing the CPU's behavior on load instructions that cannot be reordered, due to them having a read-after-write dependency. On the M1 CPU, we observe no difference in runtime whether the load addresses increment in strides or not. However, we observe a drastic speedup on the M2 and M3 CPUs only when the load's addresses are striding, whereas their runtime to execute the non-striding loads is similar to the M1. From this, we show that the LAP mechanism is present on recent Apple CPUs.

Next, to reverse engineer the LAP, we design a primitive that lets us observe its speculative behavior by causing a misprediction and then recovering side-channel traces of the resulting transient execution. Using this primitive, we identify that the LAP needs to be trained on 500 or more striding loads to activate reliably. When the LAP does activate, it speculatively loads from its predicted address, rather than the address dictated by the program. In turn, when the value from the predicted address arrives, we observe that the CPU opens a deep speculation window (up to 600 cycles), during which arbitrary computation can be performed on the value. This window allows us to leak the contents of LAP-predicted addresses via microarchitectural covert channels. Finally, we also observe that the stride must be at most 255 bytes for the LAP to generate predictions, limiting the scope of the LAP's reach to this range.

Weaponizing the LAP. Now, we assume there is a secret in the address space which the adversary cannot read. While the speculation window is sufficient to leak the secret value over a cache covert channel in principle, our primitive from before would require the secret to be at most 255 bytes from the last training address in order for the LAP to transiently load it. Therefore, we modify our primitive to read from anywhere in the address space by adding another layer of indirection (i.e., an additional pointer dereference). More specifically, we write the address of the secret at a memory location reachable by the LAP. Then, we trigger an LAP prediction on this memory location, obtaining the secret's address under speculation. We then speculatively dereference this address and leak the secret's value using a microarchitectural covert channel. This forms the basis for Spectre-LAP. In addition, we show that Spectre-LAP can not only divert data flow, but also control flow by branching to functions which never get invoked architecturally. Finally, observing that Spectre-LAP only runs to completion on mapped addresses, we use this phenomenon to defeat Address Space Layout Randomization (ASLR) on macOS.

Orchestrating an Attack on Safari. We culminate our findings by investigating the implications of data speculation on web browser security. We port our primitive to mistrain the LAP to JavaScript, where we discover that Safari's JavaScript engine exhibits behaviors that are favorable for mistraining when dealing with string objects. This results in a gadget that can disclose the content of out-of-bounds JavaScript strings. However, as JavaScript's lack of pointers precludes us from using another level of indirection, the reach of our LAP gadget is again limited to read 255 bytes. We sidestep this limitation by devising a new memory massaging technique against Safari's allocator, which lands cross-origin DOM strings from the target webpage into the attacker's window. Finally, we orchestrate SLAP end-to-end, causing the LAP to disclose sensitive content from Gmail, Amazon, and Reddit when the target is authenticated.

Summary of Contributions. We contribute the following:

- We investigate the data speculation mechanisms on Apple CPUs, discovering the LAP's presence on recent generations. Next, we reverse engineer the LAP's training and activation criteria (Section 4).
- We demonstrate that speculation using the LAP can be weaponized to achieve out-of-bounds reads anywhere in the 64-bit address space, divert control flow under speculation, and break ASLR on macOS (Section 5).
- We build an out-of-bounds read primitive in the Safari web browser which subverts Safari's sandboxing and side-channel countermeasures, leaking cross-origin content from sensitive websites (Section 6).

1.2. Responsible Disclosure

We disclosed our results to Apple on May 24, 2024. Apple's Product Security Team have acknowledged our report and proof-of-concept code, requesting an extended embargo beyond the 90-day window. At the time of writing, Apple did not share any schedule regarding mitigation plans concerning the results presented in this paper.

2. Background

Cache Organization on Apple Silicon. Like most vendors, Apple CPUs use small on-chip buffers named caches to reduce the disparity in speed between the core and memory subsystem. In addition, they feature a heterogeneous core design with Performance (P) cores and Efficiency (E) cores [5]. Both types of cores have private L1 caches and shared L2 caches within a cluster of the same core type. The caches are set-associative: they are partitioned into multiple cache sets using part of the memory address, and data from that address can fit into any of the cache ways in that set, where each way contains a cache line.

Side-Channel Attacks on Caches. As the cache is shared across all processes, an adversary on the same system can measure the latency to certain data to gain inferences about the secret-dependent activity of a target process. Broadly, the plethora of prior work can be bifurcated into FLUSH+RELOAD [24, 25, 73], where the adversary times the access to shared data, and PRIME+PROBE [21, 27, 30, 38, 44, 49, 51, 63, 68], where they time accesses to their own data which shares a cache set with the target's data.

Control Hazards and Speculative Execution. To improve performance, nearly all modern CPUs execute code out of program order, especially in terms of control flow. To avoid stalling when the correct control flow from a control hazard is not yet available, CPUs predict the control flow and execute instructions at the predicted execution path speculatively. If the prediction matches the control flow from program order, the changes in CPU state are committed and made visible to software. Conversely, if the prediction is incorrect, the CPU rolls back the modified state and resumes execution from the ground-truth control flow. However, microarchitectural changes, such as that to the cache, are not reverted. This leads to an abundance of control-flow speculation attacks, wherein an adversary can transiently access and recover secrets on the same system as the target [1, 13, 15, 16, 17, 22, 23, 28, 32, 33, 34, 36, 37, 39, 40, 53, 61, 62, 64, 65, 66, 70, 71].

Data Hazards and Out-of-Order Execution. In addition to predicting control flows, modern CPUs execute instructions in the same basic block out of program order and oftentimes in parallel when their operands are made available. However, this entails three types of data hazards that must be handled: ① Read-After-Write (RAW), where the source operand of a younger instruction is the destination of an older one, ② Write-After-Read (WAR), where a register is read by an older instruction and modified by a younger one, and ③ Write-After-Write (WAW), where a register is modified twice by older and younger instructions.

Modern CPUs can resolve WAR and WAW hazards by register renaming [60], a technique that duplicates architectural registers into several microarchitectural registers to obviate the temporal dependency caused by the overwrite. In contrast, the RAW dependency is also known as a 'true dependency' that cannot be resolved and thus reordered. Here, the instructions must run serially due to the operand of the younger instruction being unknown.

Data Speculation on Load Instructions. However, recent industry patents and works in computer architecture propound a novel mechanism named data speculation to improve instruction-level parallelism on RAW dependencies. Here, instead of speculatively executing instructions based on predicted control flow, the CPU predicts values of data based on past execution and attempts to reorder the RAW dependency by executing younger instructions based on that speculative value [14, 19, 35, 45, 47, 48, 56, 57, 69]. Load instructions are the most common targets for data speculation since they frequently occur and highly vary in latency, thus forcing the CPU to stall for RAW hazards for more than 100 cycles on cache misses.

Load Address Prediction. One common way to speculate on load instructions is to predict the memory addresses they will access. We show an overview of a Load Address Predictor (LAP) in Figure 1. Here, we focus on one ARM load instruction at address `0xabcd`, which takes the data inside the `x1` register as the load address and returns the data at that address in the `x0` register.

LAPs typically keep track of the load address in `x1` each time the instruction at address `0xabcd` is executed. If the stream of addresses is predictable, such as constants or striding values (Figure 1 (Left)), the LAP activates by speculatively issuing a load to the predicted address (`0x40`) and waits for it to resolve in `x0`. When the load resolves, its data

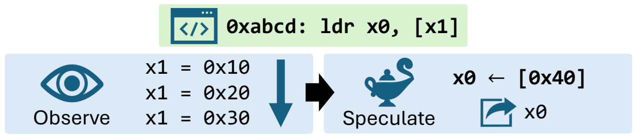


Figure 1: An overview of load address prediction.

can be forwarded transiently to younger instructions that use `x0`, speculating on the correctness of the LAP's prediction. See Figure 1 (Right). Finally, when the load address for `x1` resolves, speculation terminates. If the predicted address is wrong, the CPU flushes its pipeline and resumes execution from the correct load address, fetching the value to `x1`.

3. Threat Model

For the attack scenarios in Sections 5 and 6, we target recent Apple CPUs. We assume that the target systems run macOS 14.4, which are up-to-date at the time of writing, and do not leverage software vulnerabilities. Furthermore, we assume the systems operate in their default configurations, especially with respect to side-channel countermeasures.

In Section 5, we model the attacker as a typical native adversary with unprivileged code execution capabilities on the target system. Next, in Section 6, we adopt the typical web-based adversary model, wherein the target visits the attacker's webpage. Here, we assume the target uses the Safari 17.4 browser, also up-to-date at the time of writing.

4. Reverse Engineering the LAP

In this section, we first confirm the existence of a LAP on Apple's M2, A15, and newer CPUs, and also rule out the possibility that our results were due to prefetching. We then reverse engineer the LAP's activation criteria, and measure the CPU's transient behavior under LAP-induced speculation. Finally, we discuses limitations, such as address checks and instruction tagging.

4.1. Observing Load Data Speculation

We start with the following experiment outlined by the C code in Listing 1. Overall, we run a loop of read-after-write (RAW) dependent loads twice: first as a 'dry run' to bring all the load addresses into the cache to rule out the effects of classical prefetching, then second as a 'wet run' where we measure the runtime of the loop.

Line 2 starts the dry run, where we have filled an array with different contents depending on the experiment. In one experiment which we label as 'Striding', we fill the buffer in a pointer-chasing manner with stride S. That is, we write S to index 0, $2S$ to index S, $3S$ to index $2S$, and so on. In the other experiment labeled 'Random', we fill the buffer with randomly generated in-bounds indices. We illustrate this filling of the array in Figure 2.

Continuing the dry run in Line 3, we zero-initialize a variable `dep` to act as the RAW dependency between loads

```
1  // Dry run to cache addresses
2  volatile int array[]; // See Fig. 2
3  volatile int dep = 0;
4  for (int i = 0; i < ITERS; ++i)
5      dep = array[dep];
6  // Wet run to measure runtime
7  dep = 0;
8  uint64_t start = get_timestamp();
9  for (int i = 0; i < ITERS; ++i)
10     dep = array[dep];
11 uint64_t end = get_timestamp();
12 return end - start;
```

Listing 1: C representation of our gadget to measure the runtime of a loop of read-after-write (RAW) dependent loads.

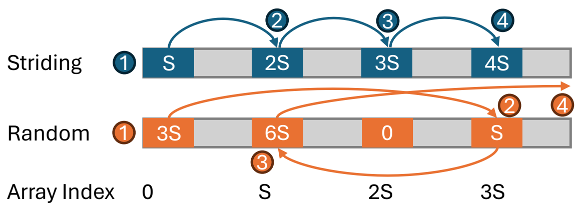


Figure 2: The contents of the array in Listing 1 for each experiment. The numbered bulletpoints indicate the order of memory accesses, with the first memory access starting at the beginning of the array.

to the array. We declare the array and dep as volatile to prevent the compiler from optimizing out Lines 4-5, as dep is zeroed again in Line 7. Lines 4-5 show how we use dep to create RAW-dependent loads in a manner similar to traversing a singly-linked list. The load address into the array for an arbitrary iteration i cannot be determined until the load of iteration $i-1$ resolves, and its load value is copied into dep. As a RAW dependency is a true dependency that cannot be resolved by register renaming (cf. Section 2), we expect the loads to be serialized on typical CPU microarchitectures, regardless of the load addresses or values.

We now move on to the wet run, where Line 8 obtains a timestamp. Lines 9-10 are identical to Lines 4-5 during the dry run, looping through the RAW-dependent loads. Finally, Line 11 obtains a second timestamp, and the difference from the first timestamp is returned in Line 12.

Experimental Setup. We run Listing 1 on the P- and E-cores of the Apple M1, M2, and M3 CPUs, using the pthread_set_qos_class_self_np API in macOS to schedule the Constant and Random experiments on each core type. For both experiments, we report the median runtime from 100 invocations of Listing 1. We set $S = 32$ bytes, and increase the ITERS variable (in Lines 4 and 9) from 10 to 1,000 in increments of 10 iterations. Given these parameters, the maximum size for the array is $1000 * S <$ 32 KiB. The L1 data cache size of the M-series CPUs is 128 KiB for the P-cores and 64 KiB for the E-cores [5], and thus all array elements can remain cached. Finally, for the get_timestamp calls in Lines 8 and 11, we use macOS's kperf API to count CPU cycles.

Results. We show the resulting plots in Figure 3. On both core types of the M1 (top left and bottom left), we observe a consistent linear increase in latency, which does not differ based on the load addresses or values. On the E-cores of the M2 (bottom center), we observe an identical trend. Thus, we conclude these cores lack load data speculation mechanisms.

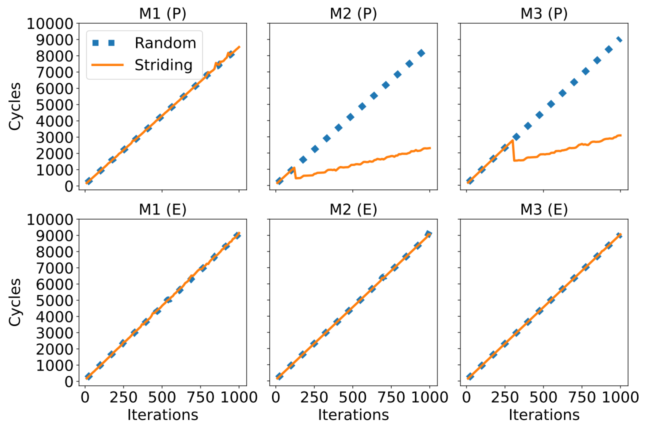


Figure 3: Striding (solid) and Random (dotted) experiment plots for the P- and E-cores of the Apple M1, M2, and M3 CPUs.

However, on the P-cores of the M2 (top center), the plot is vastly different: while the latency for the Random experiment continues to increase linearly, the latency for the Striding experiment diverges around 120 iterations, where we observe a notable speedup. As the number of iterations increases, the latency for the Striding experiment grows much slower compared to that of the Random experiment. Given that this speedup occurs even though the loads are RAW-dependent on each other and cannot be parallelized, and occurs only on loads whose addresses and values exhibit a pattern, we confirm the presence of a data speculation mechanism for loads on the P-cores of the M2.

Next, we attribute the obtained speedups to instruction-level parallelism. That is, without load data speculation, younger RAW-dependent loads must stall until older loads are resolved. Conversely, a CPU with data speculation can transiently execute the loads (using predictions) in parallel with verifying these predictions from resolved loads.

Finally, we observe similar behavior on the M3, with the E-cores showing no notable difference between the Striding and Random experiments (bottom right), but the P-cores showing a speedup on the former (top right). Moreover, whereas 120 striding loads suffice to activate the prediction mechanism on the M2, we observe this threshold is higher on the M3 where the speedup occurs at 320 iterations.

4.2. Confirming Load Address Prediction

Having observed speedups from load data speculation, we seek to confirm that the mechanism is a LAP. To test for one, we fix the load values to be random. Then, we introduce the SA+RV (Striding Addresses from Random Values) experiment where the load addresses are striding nonetheless. The top half of Figure 4 shows the array contents and memory access pattern that we desire.

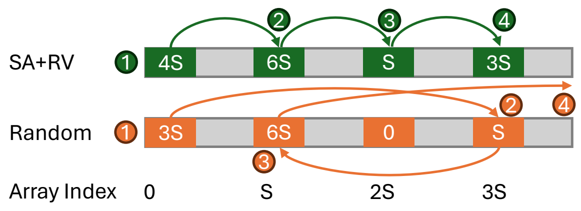


Figure 4: Memory contents and access pattern for the SA+RV and Random experiments. The latter is identical to the bottom half of Figure 2.

Despite the load values being random multiples of S from our array fill, the array accesses always stride by S in the SA+RV experiment. We compare it to the Random experiment from Section 4.1 at the bottom half of Figure 4, where both the load addresses and values are random. That is, we confirm the existence of a LAP if we observe a speedup on the SA+RV experiment compared to the Random experiment. To achieve the memory access pattern of the SA+RV experiment, we modify the for-loop of Listing 1 (Lines 3-4 and 9-10) slightly, as shown in Listing 2.

```
1  for (int i = 0; i < ITERS; ++i)
2      dep += min(array[dep], S);
```

Listing 2: Modified for-loop for making load addresses stride, despite the contents of the array being random.

Instead of assigning the load value directly to the `dep` variable to be used as a RAW dependency for the next load address, we increment `dep` by the smaller of the load value and the stride S. Furthermore, we populate the array with randomly generated nonzero multiples of S, such that regardless of the load values the array accesses will be `array[0]`, `array[S]`, `array[2S]`, and so on. We run the Random and SA+RV experiments on the P-cores of the M2 and M3 CPUs. For the SA+RV experiment, we use the same parameters (median of 100, S = 32B, and cycle-counting) as the Random experiment. Figure 5 shows the resulting latency plots.

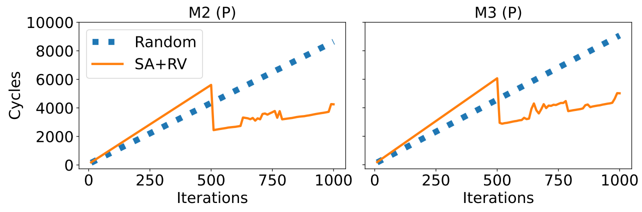


Figure 5: SA+RV (solid) and Random (dotted) experiment plots for the P-cores on the Apple M2 and M3 CPUs.

On both the M2 (left) and M3 (right) P-cores, until 500 iterations, the latency for the SA+RV experiment grows faster than the Random experiment due to the additional overhead of the `min` function in Line 2 of Listing 2. At 500 iterations and beyond, though, the SA+RV latency sharply drops below that of Random and stays below. Moreover, it grows at a notably slower rate.

From this, we arrive at an important conclusion: the M2 and M3 CPUs observe load addresses originating from the same instruction address, predicting subsequent load addresses when the previous ones are striding. Thus, we determine that their data speculation mechanism is a LAP.

Benchmarking More Apple Silicon. Next, we extend our experiments to Apple's recent mobile CPUs (A-series), which share core designs with the M-series [5]. However, A-series CPUs are available only on iOS devices, where we do not have access to CPU cycle counters. As such, we develop a portable version of the Random, Striding, and SA+RV experiments by compiling them to WebAssembly. We then compile the WebAssembly binary to machine code using WebKit, the engine underlying the Safari web browser. Since the timer resolution in WebKit is coarse (1 ms), we fix the number of iterations to one million to amplify the timing difference, while keeping the stride unchanged at 32 bytes. We indicate the presence or absence of the LAP across several devices in Table 1.

Device	CPU	LAP
MacBook Pro (A2338)	M1	✗
MacBook Air (A2681)	M2	✓
MacBook Pro (A2918)	M3	✓
iPhone 11 (A2111)	A13 Bionic	✗
iPhone 12 (A2172)	A14 Bionic	✗
iPhone 13 Mini (A2481)	A15 Bionic	✓
iPhone 13 (A2482)	A15 Bionic	✗
iPhone 14 Pro Max (A2651)	A16 Bionic	✓
iPhone 15 Pro Max (A2849)	A17 Pro	✓
iPad Pro (7th Gen.) (A2925)	M4	✓

Table 1: LAP presence/absence on recent Apple devices.

We note that the core design of the M1 is similar to the A14 Bionic, M2 to A15 Bionic, and M3 to A16 Bionic [5]. Thus, we observe results on the A-series devices that are mostly consistent with our previous observations on M-series devices. However, we observe contrasting results on two devices that were released simultaneously (iPhone 13 Mini and iPhone 13) and have the same A15 Bionic CPU. Given that the A15 is manufactured by only one firm (TSMC), to the best of our knowledge, this is the first observance of Apple CPUs with the same product code differing in microarchitectural behavior in a similar manner to the stepping level in Intel and AMD CPUs. Finally, we note that the M4 CPU was just released at the time of writing and also contains the LAP like its predecessors.

4.3. Confirming Speculative Execution via LAP

We now recall from Section 2 that LAPs speculatively execute the load on the predicted address, bring the load's value into the register file, and then use the value for younger dependent instructions until the ground-truth load address becomes known. While the drastic speedups from the Striding and SA+RV experiments in Sections 4.1 and 4.2 strongly point at such behavior, the loop of RAW-dependent loads causes difficulties for reverse engineering because in the case the LAP activates on striding addresses, the speculative execution is always correct. This causes it (and its microarchitectural traces) to be 'masked' by architectural execution, precluding us from precisely measuring behavior within the LAP's speculation window.

Therefore, in this subsection, we aim to collect stronger evidence for LAP-induced speculation and lay the groundwork for subsequent reverse engineering experiments by causing the predicted load address to differ from the ground-truth load address, resulting in misspeculation that leaves different microarchitectural traces.

Gadget Setup. We first implement a singly-linked list where each node has a pointer to the next node and a pointer to data in Figure 6. Here, we aim to mistrain the LAP using the loads to the data elements.

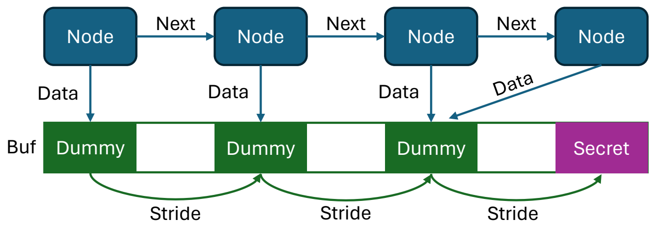


Figure 6: A graphical overview of the linked list and buffer data structures to cause misprediction by the LAP. These data structures are also used by the code shown in Listing 3.

As shown in all the nodes but the last, the nodes' data pointers hold striding offsets of a writable buffer in memory. In each of the offsets, we write a dummy value. However, we break this striding behavior for the last node. That is, to the next striding offset of the buffer, we write a secret value this time. But the last node's data pointer does not point there, instead pointing to the same memory address as the second-to-last node. As such, all architectural dereferences to data only point to dummy values.

Inducing Misprediction. Next, using the C code in Listing 3, we show how to train the LAP such that it predicts a load address that is never architecturally accessed and transmits its value over a covert channel.

```
1  cache_flush(last_node, FR_buf);
2  struct Node *n = first_node;
3  while (n != NULL) {
4      uint8_t LAP_load = *(n->data);
5      FR_buf[LAP_load * PAGE_SZ];
6      n = n->next;
7  }
8  return FR_recv(FR_buf);
```

Listing 3: C representation of our misprediction gadget. The gadget performs a linked list traversal while dereferencing the data pointer of each node and encoding the value into a cache channel.

We use FLUSH+RELOAD [73] as our covert channel for experimental expediency, as it supports transmitting several different byte values. Line 1 performs its flushing step, but also flushes the last linked list node from the cache. By doing so, the value of the last node's data pointer is not quickly retrievable, forcing the CPU to use the predicted address from the LAP. We show the effects of running Listing 3 on Figure 6's data structures in Figure 7. First, we illustrate Line 1's effect at the top right corner, with the last node being colored orange to indicate it is uncached.

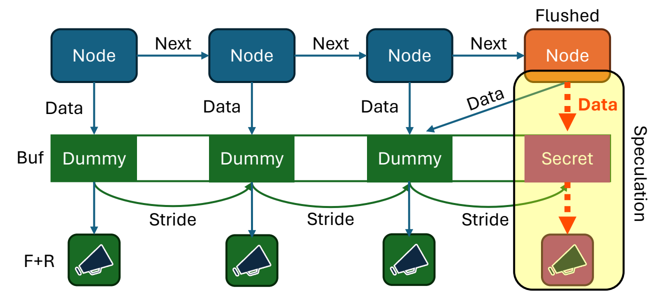


Figure 7: Overview of the outcome of running Listing 3 on Figure 6. There is a divergence between architectural execution (blue arrows) and speculative execution (red arrows, highlighted area).

Next, Line 2 declares a pointer to the first linked list node, and Line 3 starts a while-loop for the traversal. In this loop, Line 4 dereferences the data pointer of the current node, storing its value in the `LAP_load` variable. It is this load instruction which we coerce the LAP to predict the address of, as for all nodes but the last, the load addresses become B, $B + S$, $B + 2S$ and so on for a given stride S and buffer address B. Hence, at the last node, the LAP's predicted load address becomes $B + nS$ for a linked list of length $n+1$, whereas the actual load address is $B+(n-1)S$. The right side of Figure 7 depicts this, where the speculative dereferences (red arrows) differ from architectural, putting the secret value into `LAP_load`.

Finally, Line 5 transmits the load value, and Line 6 advances to the next linked list node. We then receive over the covert channel on Line 8. Without any LAP activation, we expect to receive the dummy value only. On the other hand, with LAP activation (and misprediction), we expect to receive both the secret and dummy values. This is shown at the bottom of Figure 7, where the bullhorn icons represent covert channel transmissions. The area highlighted in yellow marks the speculative execution from the mispredicted load address, which leads to the secret value being transmitted.

Experimental Setup for Reverse Engineering. We now introduce the setup we use to collect data from the workload in Listing 3, and for further reverse engineering experiments in Sections 4.4 to 4.6. We focus on the LAP of the Apple M2 CPU. For accurate measurements, we require the ability to count cycles and flush cache lines from userspace programs. Furthermore, to reduce noise and variation, we require the ability to manually control CPU frequency, isolate CPU cores from being used by operating system processes, and pin programs to the isolated cores.

To that aim, we perform our experiments on Fedora Linux's Asahi Remix with kernel version 6.6.3-414, since macOS lacks support for all of our requirements except cycle counting. Here, we set all cores to their maximum frequency using the `cpufreq` interface in `sysfs` and the userspace governor. We exclude one P-core using the `isolcpus` kernel parameter, and pin our experiment to it with the `sched_setaffinity` system call.

Initial Results for Measuring Speculation. Following prior results on Figure 5 where we observed LAP activation at 500 iterations, we use a linked list of 501 nodes to train the LAP on 500 striding data pointer loads, which are made RAW-dependent by the linked list traversal. Likewise, we keep the stride between data pointer addresses at 32 bytes. Next, we test our gadget 1,000 times, where in each trial we repeat Listing 3 for 20 runs. Over the FLUSH+RELOAD covert channel, we receive the dummy value on all 1,000 runs and the secret value on 721 runs. We show a histogram of the latency to reload the secret value in Figure 8.

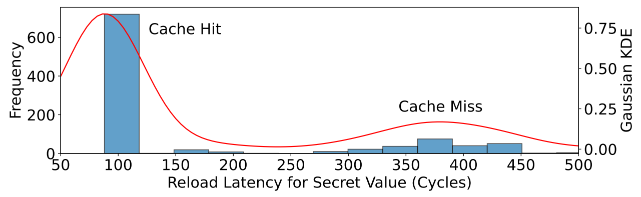


Figure 8: Histogram of latencies to the FLUSH+RELOAD memory address that corresponds to the secret value.

Finally, we ascertain our results by not writing the secret value into the striding buffer offset, and confirming that it is no longer received over the covert channel. Hence, we demonstrate that we can dereference pointers under speculation and make the CPU operate on data at rest by the LAP. That is, after being written, the secret value is never architecturally read in by the core.

4.4. Spatial Conditions for Speculation

Having observed LAP-induced speculation from a mispredicted load address, we now seek to identify the spatial conditions to reliably activate the LAP, that is, in terms of virtual addresses the LAP interacts with and their topology. We observe that the training length (i.e., number of linked list nodes) and the stride for the load addresses are both parameters, and analyze how they affect the probability of the LAP activating. Then, we ask questions about the paging of addresses and their effect on LAP training or activation, as most classical prefetching occurs within a page.

Training Length and Stride: Experimental Setup. Our setup here is largely identical to the experiment in Section 4.3, where we run our misprediction gadget 1,000 times on the same set of parameters. However, we vary the linked list length from 20 to 1,000 in increments of 20, as well as the stride between pointers to the buffer from -320 bytes to 320 bytes in increments of 8 bytes. That is, for a negative stride value, we start writing the first dummy value at the end of the buffer, and traverse it backwards with the given stride to train the LAP.

Training Length and Stride: Results. We plot our results in Figure 9 as a heatmap. Here, training length and load address stride are the X- and Y-axes, and we represent the number of observed LAP activations (out of 1,000) as the heat, where the closer the color to dark blue, the more the LAP activates. Also, we highlight zero values in yellow.

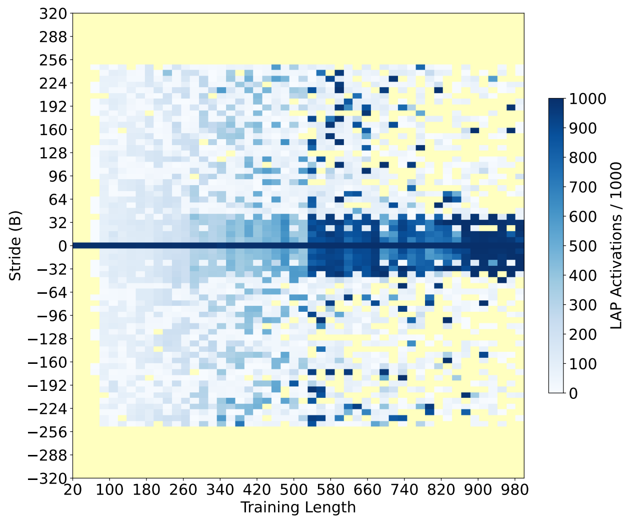


Figure 9: Heatmap showing the effects of the linked list length and the stride between training load addresses on the likelihood of LAP activation. Values of zero are highlighted in yellow.

Firstly, at Stride = 0, the LAP seems to always activate regardless of training length. However, this is expected, as a zero stride leads the data pointers of all nodes to point to the start of the buffer. In turn, this causes the secret value to be transmitted architecturally. Hence, this is not speculation, and we disregard the result (horizontal blue line).

Dead Zones for LAP Activation. Secondly, we observe three 'dead zones' in Figure 9 with no LAP activations (highlighted in yellow). The first is to the left side of the heatmap, when the training length is less than 80. As we can now measure LAP activity much more precisely by checking for misprediction, we regard this as the minimum training threshold for RAW-dependent striding load addresses from which we can observe LAP activity (albeit at a low rate). Furthermore, the second and third dead zones are at the top and bottom of the plot, where the absolute value of the stride is 256 bytes and above. From this, we confirm that the LAP can keep track of both positive and negatively striding load addresses, and conjecture that its internal state for the stride is one sign bit with eight ($\log_2 256$) magnitude bits.

Optimal Conditions for LAP Activation. Thirdly, we observe regions of both training length and stride wherein the LAP has a proclivity to activate. We note a sharp increase in activations across most strides with magnitude less than 256 bytes when the training length exceeds 500. This threshold coincides with when we observed an abrupt speedup while testing for a LAP in Section 4.2, hence we regard this as the practical training threshold to regularly observe LAP activations. On the other hand, across all training lengths above 80 (and especially past 500), we emphasize a 'block' of frequent activations when the stride has a magnitude below 64 bytes. Coincidentally, this is the size of an L1 cache line on Apple CPUs: we conjecture that small strides that access a cache line more than once during training increase the LAP's inclination to activate.

Measuring LAP Training and Prediction Across Page Boundaries. We also demonstrate that the LAP on the M2 maintains training state across page boundaries, but will not generate a prediction on a new page. See Appendix A.

4.5. Temporal Conditions for Speculation

Moving away from spatial conditions, we focus on identifying the temporal conditions for starting LAP-induced speculative execution and prolonging the speculation window. First, we measure the window for which the LAP keeps state during training when given extraneous instructions. Then, we measure how deeply the LAP speculates conditioned on the caching status of the variables that are crucial to the linked list traversal.

Gadget Overview. We modify the loop of Listing 3 (Lines 3-7) into Listing 4 to better separate (at the source code level) the training iterations from the loop iteration where the LAP misprediction happens.

```
1 uint8_t LAP_load;
2 for (int i = 0; i < LL_SIZE; ++i) {
3     // Insert MULs for training window.
4     LAP_load = *(n->data);
5     n = n->next;
6 }
7 // Insert MULs for speculation window.
8 FR_buf[LAP_load * PAGE_SZ];
```

Listing 4: Rewritten linked list traversal loop from Listing 3 that allows us to insert dummy instructions at two different points in the code to measure the training and speculation windows.

This loop is functionally equivalent for traversing the linked list and dereferencing each node's data pointer, thereby training the LAP. However, on Line 2, we use a for-loop spanning the linked list's length instead of a while-loop to make the loop unrollable by the compiler. Finally, we move the FLUSH+RELOAD transmission outside the loop to Line 8, having declared the `LAP_load` variable in Line 1 to have it in scope. As the loop terminates immediately after LAP activation on the last node, the covert channel reception is identical to before: LAP activation results in the secret and dummy values being transmitted, while no activation transmits just the dummy value.

However, the most notable difference is that we introduce two lines where we can add extraneous `mul` instructions. If we insert them in Line 3, they are architecturally executed between each load that trains the LAP in Line 4. This lets us measure the effect of extraneous instructions during LAP training. On the other hand, if we insert them in Line 7, they are also speculatively executed before the covert channel transmission when the LAP activates. Hence, here we can measure the speculation window's length.

Measuring the Training Window. We run our misprediction gadget for 1,000 runs on a stride of 32 bytes and 1,000 training addresses, since we have identified these parameters to be optimal for activating the LAP from Section 4.4. We continue using these parameters for all experiments in this subsection. We use Listing 4 for the traversal loop, and insert `mul` instructions into Line 3 from zero to ten of them in increments of one. We plot the effect of the extraneous `mul` instructions on the number of LAP activations in Figure 10.

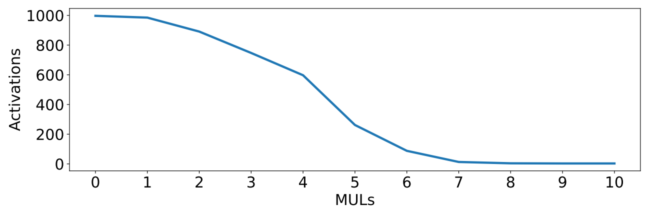


Figure 10: Number of LAP activations out of 1,000 runs when extraneous `mul` instructions are inserted during training.

Surprisingly, we observe a sharp decrease in activations past 2 `mul` instructions, to no activations at 8 `muls` and more. Hence, we conclude that the LAP employs a short training window, wherein the striding loads must be executed quickly enough to train the LAP. With Apple documentation stating that each `mul` requires 3 cycles [5, Appendix A], this window is at most 24 cycles.

Measuring the Speculation Window. We repeat the setup for measuring the training window. Rather than inserting the `mul` instructions into Line 3 of Listing 4, we insert them into Line 7 instead (such that they will execute speculatively) from 0 to 300 of them in increments of 10. We measure the speculation window conditioned on the caching status of the last linked list node and the predicted address. Firstly, we keep the predicted address cached, and measure how many `mul` instructions can fit when the last node is cached or flushed in Figure 11 (Left). Then, we flush the predicted address and repeat the experiments in Figure 11 (Right).

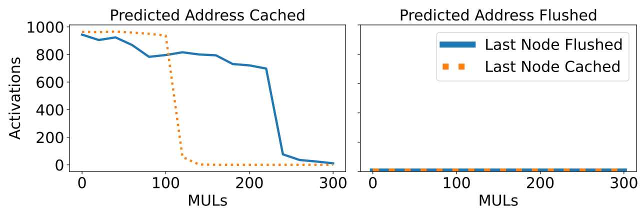


Figure 11: Number of LAP activations when `mul` instructions are placed in the speculation window. (Left) The LAP's predicted address is cached. (Right) The predicted address is flushed.

When the predicted address is cached, we observe that the window length when the last node is cached is similar to that of a speculation window opened by a branch predictor when the predicate is evicted, at around 100 `muls` [31, Section 4.2]. Contrarily, when the last node is flushed, we show a speculation window twice as deep, exceeding 200 `muls` (or 600 cycles). However, with the experiments where the predicted address is flushed, we observe 0 activations regardless of the caching status of the last node. Hence, we conclude that another precondition for the LAP to activate is for the predicted address to be cached, and that the LAP does not prefetch its predicted address.

Comparison with Data-Dependent Prefetching. Now that we have understood how the LAP behaves, we revisit the Data Memory-dependent Prefetcher (DMP) that was shown in prior work [18, 67] to be present also on recent Apple CPUs. If data being returned from a load resembles a valid pointer, the DMP treats the data as an address and dereferences it, anticipating that the address will be loaded from at some later point. Although the LAP and DMP are both present on Apple CPUs and activate on load instructions, we note some important distinctions. Unlike the LAP, the DMP does not speculate past RAW dependencies, since it is a hardware prefetcher. Conversely, the LAP is not a prefetcher: from the speculation window experiments in this subsection, we observed that the LAP terminates if the predicted address is not cached, instead of prefetching it and continuing. Finally, only the LAP opens a speculation window wherein arbitrary computations can be performed.

4.6. Confirming Instruction Address Tagging

We recall that typical LAPs proposed in literature keep state per instruction address (cf. Section 2). Using the experiments in Appendix B, we first observe that unrolling Listing 4 causes the LAP not to activate, confirming that it uses the instruction address as a tag. Subsequently, we test if the LAP uses all or part of the instruction address for such tagging. We begin by observing that the LAP's training state persists even if training is interrupted at the 30th-to-last node. Next, we traverse the last 30 nodes using a clone of Listing 4 whose instruction address aliases the original for the lowest 6 to 47 bits. Finally, we confirm that the LAP uses all canonical address bits of the training loads as a tag.

5. Weaponizing the LAP for Attacks

Having confirmed the LAP's existence and described techniques to use it for speculative execution in Section 4, we now weaponize these techniques to serve as proof-of-concepts on the danger of LAPs. We show how the linked list traversal which activates the LAP can be modified to speculatively hijack both data and control flow, just with one more load in the traversal pattern. We then use this new technique, which we name Spectre-LAP, to break Address Space Layout Randomization (ASLR) on macOS.

5.1. Spectre-LAP

With our misprediction experiment in Section 4.3, we recall from Figure 7 that with a load address stride of S, the LAP gets trained to jump S bytes beyond the last dummy value in the buffer. However, from an adversarial perspective, the out-of-bounds reach becomes limited to S which we show in Section 4.4 is at most 255 bytes. As such, we now aim to augment our primitive with the LAP's deep speculation window shown in Section 4.5 to transiently reach anywhere in the address space.

Gadget Overview. We revise the experiment in Section 4.3 to use the data structures shown in Figure 12. We assume there is a secret at address `addr` that we wish to read. For this, we retain the linked list with data pointers with striding data pointers, and we flush the last node to force the CPU to use the LAP's predicted address as before.

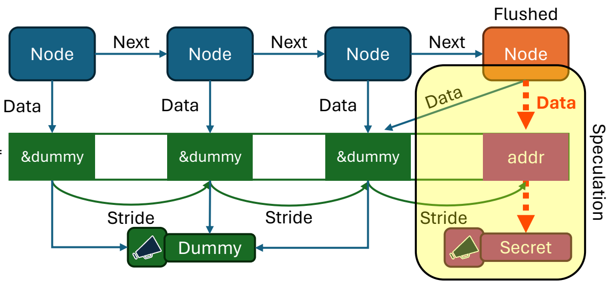


Figure 12: Our modified misprediction primitive that speculatively dereferences double pointers, allowing for a 64-bit out-of-bounds read. The bullhorns indicate that both the dummy and secret data will be transmitted over the covert channel.

However, our goal is to hijack data flow further beyond S bytes of the last dummy in the buffer. To that aim, we add a level of indirection by treating the contents of the buffer as addresses, as opposed to values. We declare each node's data pointer to be a double-pointer, and write the dummy element's memory address at the buffer's striding offsets. Next, we write `addr` at the last striding offset (i.e., the LAP's predicted address), but make the last node's data pointer point to the last dummy address written into the buffer. In turn, for every node we traverse, we dereference the double-pointer twice and transmit the resulting value over the covert channel, which we depict with bullhorns. Hence, LAP misprediction on this linked list results in the LAP first loading `addr`, then loading the secret, then finally transmitting the secret as we show in Figure 12's highlighted box. Lastly, we show the traversal code corresponding to our new linked list in Listing 5.

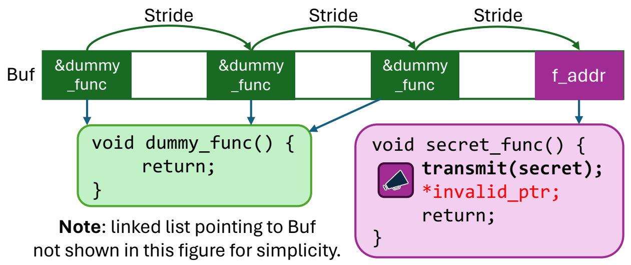
```
1 while (n != NULL) {
2     uint8_t LAP_load = **(n->data);
3     transmit(LAP_load);
4     n = n->next;
5 }
```

Listing 5: C representation of the linked list traversal code for Figure 12, i.e., a gadget for Spectre-LAP.

The sole change we make is Line 2 (highlighted). Here, the first dereference into the buffer is the speculative load by the LAP, and retrieves `addr`. When the LAP speculatively forwards `addr`, the second dereference loads the secret into `LAP_load`, which gets leaked in Line 3.

5.2. Hijacking Control Flow

Now, we show a variant of Spectre-LAP that diverts control flow under speculation in addition to data flow. Previously in Section 5.1, the dummy and secret elements in Figure 12 were buffers containing data. Now, we assume there is a function we wish to call during speculation named `secret_func`, whose entry point is at address `f_addr`. To divert control flow, we replace the dummy buffer with a dummy function that shares a signature (arguments and return type) with `secret_func`. In turn, we write the address of the functions to the buffer, making each linked list node contain a double function pointer. Graphically, we keep the linked list structure at the top half of Figure 12, but replace the contents of the buffer and the memory pages for the secret and data elements with Figure 13.


Stride
Stride
Stride
Buf
&dummy_func
&dummy_func
&dummy_func
f_addr
Note: linked list pointing to Buf not shown in this figure for simplicity.

```
void dummy_func() {
    return;
}
```

```
void secret_func() {
    transmit(secret);
    *invalid_ptr;
    return;
}
```

Figure 13: Our changes to the lower half of Figure 12 for the pointer values in the buffer to point to dummy and secret functions.

Invoking Rogue Functions Under Speculation. In this modified setup, the dummy function does nothing and returns. Conversely, `secret_func` contains the covert channel transmission, followed by a load to an invalid address. This ensures that our program will segfault if the secret function is ever invoked architecturally. Next, we replace Listing 5 with Listing 6 to handle function pointers.

```
1 while (n != NULL) {
2     **(n->data)(); // Call pointed function
3     n = n->next;
4 }
```

Listing 6: C representation of the modified Spectre-LAP gadget to dereference function pointers.

Comparing Line 2 (highlighted) with that of Listing 5, the first dereference is still the load which the LAP activates on. However, at that point, the second dereference now brings the entry point of the secret function (`f_addr`) into the CPU's register file. As the CPU continues to speculate, it branches to the entry point and executes the function body, which contains the covert channel transmission (instead of within Listing 6). As such, if the LAP activates and diverts control flow under speculation, we will receive a value over the covert channel. In contrast, if the LAP fails to speculatively call functions, we will not receive any value.

5.3. Evaluation and Breaking ASLR

We now evaluate the accuracy and throughput for both variants of Spectre-LAP on the M2 and M3 CPUs. Moreover, when using our attack on an invalid address, we find that speculation continues to the covert channel transmission only when we dereference mapped memory under speculation. This serves as our motivation for breaking ASLR.

Attack Setup. For all three attacks in this section, we report the median value of 100 runs. Moreover, as macOS disables cache flush instructions, we use EVICT+RELOAD as the covert channel instead. We continue to use the optimal LAP training parameters of 1,000 addresses striding 32 bytes apart from Sections 4.5 and 4.6.

Out-of-Bounds Reads. We first measure the out-of-bounds read primitive from Section 5.1 by reading a secret string. Here, we report the leak rate and bitwise accuracy when recovering the string. The top row of Table 2 shows our results, where we demonstrate that we can read out of bounds robustly on both CPUs. Though, we observe the LAP is more difficult to activate on the M3, indicated by its significantly lower throughput.

Attack	M2 Acc.	M2 Rate	M3 Acc.	M3 Rate
OOB Reads	1.00	9,140 b/s	1.00	5,795 b/s
Control Flow	0.99	9,481 runs/s	0	0 runs/s
ASLR Break	0.72	11.39 ms	0.44	1,299.26 ms

Table 2: Results for Spectre-LAP and ASLR break attacks on the M2 and M3 CPUs, shown as the median of 100 trials. The baseline accuracy, i.e., randomly guessing the ASLR slide, is 1/5120 ≈ 0.0002.

Hijacking Control Flow. Then, we measure the control flow hijacking primitive from Section 5.2. In this case, we measure throughput as how quickly we can execute the misprediction routine, and accuracy as the fraction of executions that resulted in a covert channel transmission. In the middle row of Table 2, we observe similar results as reading out-of-bounds on the M2. However, we do not receive anything over the covert channel on the M3. Thus, we conjecture that newer Apple CPUs contain measures that prevent function calls from executing under speculation.

Breaking ASLR on macOS. XNU (the kernel underlying macOS) employs a maximum slide of 80 MiB, placing the entry point of a Mach-O binary at the start of any 16KiB page between virtual address `0x100,000,000` and `0x105,000,000` [8, 9, 10]. The 'magic' byte string denoting the start of every 64-bit Mach-O binary is `0xfeedfacf` [7], hence we repeatedly search for this sequence using Listing 5 amidst 5,120 possible pages. We measure the time to find a match, and check our guess against the ground-truth slide value using the `vmmap` utility.

We succeed in breaking ASLR on both the M2 and M3, and show the results in the bottom row of Table 2. We find the M2's more amenable LAP beneficial, recovering the correct slide on 72 runs in a median time of less than 12 ms. In contrast, on the M3 the search must be repeated several times until we can output a guess, slowing the runtime by two orders of magnitude and also affecting accuracy.

6. Attacking Safari With the LAP

We now reason about reading out-of-bounds in the JavaScript sandbox of the WebKit engine underlying the Safari web browser. Firstly, we describe how to overcome the restrictions set by JavaScript sandboxing such as the lack of pointers, memory management, and high-resolution timing, resulting in a gadget that retains the linked list structure but is able to transiently read JavaScript string objects which do not belong to the attacker's webpage. Secondly, we further discuss and leverage WebKit's memory allocation techniques, which enable us to place an arbitrary JavaScript string from any target webpage within the out-of-bounds reach of our browser-based gadget.

These two parts complete our attack, which we name SLAP, a portmanteau of speculative execution on data, Safari, and the LAP. Finally, we show how SLAP can compromise the security of the Safari web browser, reading login-protected data from the DOM of real websites.

6.1. Timer-Resilient Covert Channel

As a side-channel countermeasure, WebKit restricts the timer resolution in JavaScript to 1 ms [31, 43]. Hence, for an end-to-end attack, we must first be able to distinguish cache hits from misses with extremely coarse timing, about seven orders of magnitude coarser than the timing difference between a single cache hit and miss. We achieve this with an amplifier gadget which we detail in Appendix C.

6.2. SLAP Gadget Overview

We recall that the out-of-bounds read primitive in Listing 5 uses two data structures: a linked list with data attached, and a buffer that has pointer values written into it at a fixed stride. The former is possible in JavaScript, but pointers do not exist in the language. After observing that the metadata object of JavaScript strings in WebKit is 16 bytes wide, and that hundreds of them can be contiguously allocated, we replace the buffer with a region of WebKit's heap for JavaScript string objects full of dummy strings.

Training the LAP on String Objects. We illustrate the changes for training the LAP on JavaScript strings in Figure 14. Similarly to the native version in Section 4.3, we adopt a construction where the last linked list node is evicted to prolong the LAP's speculation window (cf. Section 4.5), and where each node's data reference is a training address.

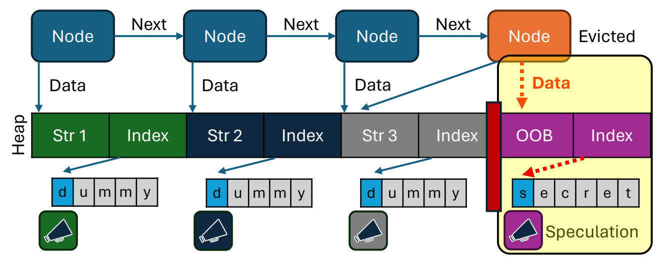


Figure 14: Graphical overview of the browser-based version of our LAP-training gadget. Architectural execution is shown in blue arrows, while speculative execution is shown in red arrows and the highlighted region.

In this case, the data are JavaScript strings owned by the attacker's webpage, labeled Str 1, 2, and 3 in the figure. This lets our training code obtain architecturally valid references to them in the linked list nodes. For this subsection, we assume that adjacent to these strings is another JavaScript string that the attacker's webpage does not own. As such, JavaScript from the attacker's webpage cannot reference this string, making it architecturally out-of-bounds.

After making the last node's data variable reference Str 3, we traverse the linked list with the JavaScript code shown in Listing 7. While similar to the native linked list traversal, Line 3 is the key difference. As we cannot dereference pointers in JavaScript, we retrieve `n.data` instead. By doing so, under the hood, Safari's JavaScript engine dereferences the addresses of the string objects which we have allocated in strides, thereby training the LAP. Then, to access the contents of the out-of-bounds string, we call the `charCodeAt` method, which returns the ASCII value of the character at the given index. Finally, for the `transmit` function in Line 4, we encode the bits of the ASCII value into the cache state, such that we can recover them subsequently using the amplifier from Section 6.1.

```
1 let n = first_node;
2 while (n) {
3     const secret = n.data.charCodeAt(index);
4     transmit(secret);
5     n = n.next;
6 }
```

Listing 7: Our browser-based gadget to mistrain the LAP using the data structures shown in Figure 14.

6.3. Exploiting Safari's Memory Model

Thus far, we use the linked list training method on contiguously allocated JavaScript strings for an out-of-bounds memory access, which will proceed through speculation as long as the memory layout is also that of a string. We also amplify cache hits and misses to a point where they can be distinguished with coarse timers. Now, we recall from Section 4.5 that the LAP's maximum stride is 255 bytes. We indicate this as the red zone beyond our training strings in Figure 15: in contrast, data placed further than that in the black zone are not exploitable. Hence, our goal is to place sensitive data from a target website into this span.

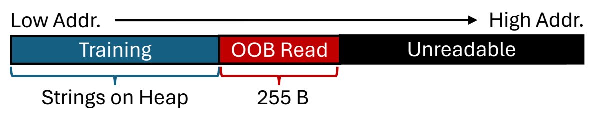


Figure 15: Diagram of the regions of WebKit heap memory that are and are not reachable with our attack primitive.

Consolidation and Object Heaps. WebKit uses its own memory allocator, `libpas`, separately from the operating system's `malloc` API [50]. With `libpas`, WebKit partitions its virtual address space into several heaps, where each heap can be multiple disjoint areas of addresses but always contains JavaScript objects of the same type [6]. On the other hand, the iLeakage attack observed that WebKit does not isolate webpages into separate rendering processes when using the `window.open` API call [31, Section 5.1]. We confirm this observation. Moreover, we observe that JavaScript objects from different webpages but of the same type can share a heap (in addition to a process).

Inspecting WebKit's Heap Allocations. Now, we inspect what data are allocated within the 255-byte reach of our training strings in the JavaScript string heap. We first spawn a new WebKit rendering process to handle the attacker webpage. Then, we use the `window.open` call to render the target webpage in the same address space. Lastly, we allocate our training strings and inspect the address space using a debugger. However, we observe that strings from the target webpage's DOM are not within the reach. While we do find reachable memory where WebKit has allocated a JavaScript string, the strings do not contain any user data. Instead, they are internal to WebKit's built-in JavaScript APIs, such as error messages. We depict this case in the memory diagram of Figure 16 (Top).

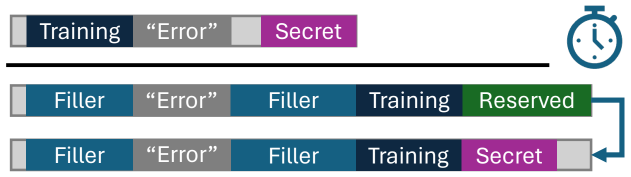


Figure 16: Simplified memory layouts of WebKit's address space. (Top) No memory pressure is applied. (Middle) Memory pressure is applied with the filler strings, and the target page has not loaded yet. (Bottom) Memory pressure is applied, and the target page has completely loaded.

Massaging with Memory Pressure. Next, we observe that memory pressure causes `libpas` to behave differently, allocating more virtual address ranges for the string heap. Hence, we allocate 'filler' JavaScript strings in the attacker webpage first to exert memory pressure. Then, we allocate our training strings. At this point, we observe that the memory contents of the 255 bytes adjacent to the last training string are vastly different. Instead of containing internal WebKit strings, the region becomes reserved for future string allocations. See Figure 16 (Middle).

Next, we load the target page using `window.open`. As the page renders and executes JavaScript, the reserved region becomes populated with strings as the target page's scripts interact with its DOM. Accordingly, we can probabilistically induce the allocation of strings holding sensitive information within the 255-byte reach of the LAP, as shown in Figure 16 (Bottom). Finally, we note that memory pressure can be applied discreetly and without attracting the user's attention, as the attacker's website only needs to allocate a few megabytes of filler strings to trigger `libpas` into creating these reserved areas.

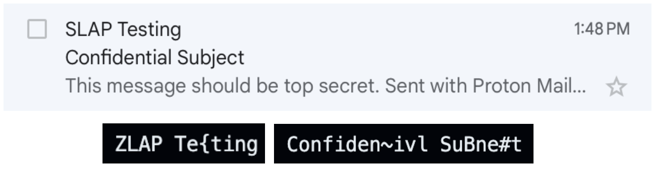
6.4. Reading Data Across Websites

With the setup for SLAP now being complete, we now demonstrate its application to real-world targets. We first set up a proof-of-concept target page that continuously queries a server for the current time and displays it to the user when the response arrives. In order to display the updated time, our website must modify the DOM. This causes WebKit to scan all DOM nodes and store the resulting HTML as a string, reliably putting it into the LAP's reach when the attacker webpage `window.opens` our target page.

Initial Benchmarks. Using the M2 CPU, we run SLAP end to end on macOS and Safari. Firstly, our attacker page allocates 100,000 filler strings. Each string's inline size on the heap is 16 bytes, resulting in 1.6 MB of pressure on `libpas`'s heap for JavaScript strings. Secondly, we allocate the training strings for the SLAP gadget, observing that about 500 of them can be contiguously allocated. Thirdly, we open the target page described above and observe that the DOM is usually 64 bytes away from the last training string. Hence, to train the LAP on a 64-byte stride, we construct the linked list from every fourth string, resulting in 125 linked list nodes. Finally, we repeat reading the DOM of the target webpage 10 times, with a median bitwise accuracy of 87.9% and throughput of 0.384 bits per second. Here, we observe that most noise is from single-bit errors, and can be further filtered with repeated sampling.

Reading Inbox Data from Gmail. We now target real-world websites. JavaScript runs on virtually every website nowadays, interfacing with the `document` API to read and write parts of the DOM. As such, we now investigate which parts of the DOM of one of the largest email services can be allocated adjacently to our training strings.

We assume the target is authenticated to Gmail, and visits the attacker webpage. The attacker webpage allocates 1.7 MB of filler and training strings, and then calls `window.open` on Gmail's inbox page when the mouse cursor is placed over itself. As Gmail loads, JavaScript in the page starts rendering the inbox, whose content is personalized to the target. Over repeated trials, we show that the subject line and the sender's identity can land in the reachable out-of-bounds region of the LAP, allowing for recovery by the adversary in Figure 17.

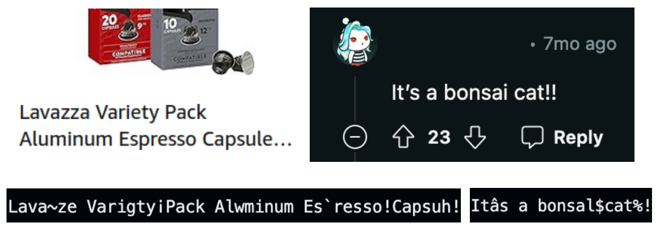

SLAP Testing 1:48 PM
Confidential Subject
This message should be top secret. Sent with Proton Mail...

```
ZLAP Te{ting   Confiden~ivl SuBne#t
```

Figure 17: (Top) Email subject and sender name shown as part of Gmail's DOM. (Bottom) Recovered strings from this page.

Fingerprinting Amazon and Reddit Activity. The main pages of web services that recommend content to authenticated users act as fingerprints for their past activity. As such, we turn our attention to Amazon's 'Buy Again' page and Reddit's home page, as both are major e-commerce and forum platforms where users can express interest for certain product categories or discussion topics ('subreddits').

Similarly to the Gmail scenario, we assume the target is authenticated to Amazon and Reddit and visits our webpage. We repeat the attack procedure from before, but cause the `window.open` calls to open Amazon's 'Buy Again' page and Reddit's home page. Then, we observe what personalized data appears within reach of the LAP. On Amazon, we recover a product description of coffee pods, which the user has purchased previously, as we show in Figure 18 (Top Left, Bottom Left). Furthermore, on Reddit, we recover text from a comment on a post which belongs to a subreddit that the user subscribes to: see Figure 18 (Top Right, Bottom Right). Here, we note we can use Reddit's search feature to recover the original post and subreddit.


Lavazza Variety Pack
Aluminum Espresso Capsule...

• 7mo ago

It's a bonsai cat!!

```
Lava~ze VarigtyiPack Alwminum Es`resso!Capsuh!
```

```
Itâs a bonsal$cat%!
```

Figure 18: (Top Left) A listing for coffee pods from Amazon's 'Buy Again' page. (Bottom Left) Recovered item name from Amazon. (Top Right) A comment on a post on Reddit, and (Bottom Right) the recovered text.

7. Mitigations

Having demonstrated new attack surfaces opened by the LAP, we now reason about strategies to mitigate them. As a first step, in Section 4.1, we observed that E-cores on the M2 and M3 CPUs do not have the LAP. On MacOS and iOS, this entails a simple user code change in the form of adding a call to the `pthread_set_qos_class_self_np` API with the `QOS_CLASS_BACKGROUND` argument [2]. However, this approach is not universal: it may be undesirable to computation-intensive applications, and E-cores in future Apple CPUs may or may not have the LAP.

Attempting to Disable the LAP. Next, we seek to disable the LAP altogether by writing to several candidate system registers. On the M2 and M3 CPUs, our first candidate is the Data Independent Timing (DIT) bit of the Armv8.4-A ISA [11], which specifies that certain instructions should have a latency independent of the operands when this bit is set. Furthermore, the DIT bit can be set from userspace (EL0) and per-process on macOS, and has been shown to disable the Data Memory-dependent Prefetcher (DMP) on the M3 CPU [18]. Hence, we hypothesize that it may also disable other unconventional performance optimizations such as the LAP. For the same reason, our second candidate is bit 30 of the `HID11_EL1` system register, which has been shown to disable the DMP on the M1 and M2 CPUs [18, 41]. However, this bit can only be set on Linux, which has no M3 support at the time of writing.

Testing More System Registers. On the M2 CPU and Linux, we reference the Asahi project's documentation on Apple Silicon's system registers [12] for bit descriptions pertaining to prefetching. From this, we test the following five bits: Bit 13 of `HID2_EL1` "Disable MMU MTLB Prefetch", Bits 44-45 of `HID5_EL1` "Disable HWP Load/Store", Bit 0 of `HID10_EL1` "Disable HWP Gups", and Bit 3 of `ACTLR_EL1` "Disable HWP". Here, we hypothesize that 'HWP' is an acronym for hardware prefetcher.

Results. We repeat the experiments with a loop of RAW-dependent loads from Section 4.1 with one candidate bit set at a time, with all others at their default values. The speedup on the M2 and M3 CPU's P-cores is still present with the DIT bit set, indicating it does not disable the LAP. For the system registers, the `HID5_EL1` bits cause Linux to crash immediately. All other bits do not affect operation, but fail to eliminate the speedup likewise.

The 'Kill Switch' Bit. Another system register bit of interest on Apple CPUs is bit 4 of `HID4_EL1`. However, its description is "Force CPU Oldest In Order" [41], resembling a kill switch to disable all out-of-order execution. Indeed, we do not observe a speedup from the experiments in Sections 4.1 and 4.2 after setting this bit. Repeating the misprediction experiment from Section 4.3, we observe only dummy values over the covert channel and no secret values, confirming again that this bit does disable the LAP.

However, we cannot recommend this as a mitigation due to macOS currently lacking support for setting this bit even for privileged users, necessitating Apple-released software updates. Moreover, the performance penalty is dire: we measure this with PassMark's PerformanceTest version 11.0.1002. On default settings, the M2 scores 8,098 points. However, with the 'kill switch' bit set, the score drops to 2,873, a slowdown of 2.81x. For comparison, this is just below the 2,929 points scored by an Intel Core i5-2500T, a consumer desktop CPU released in 2011 [46]. We leave the task of finding a method to disable the LAP at the CPU level without significant performance impact to future work.

Considerations for Safari. We emphasize the importance of site isolation [55], a mechanism preventing webpages of different domains from sharing rendering processes. Site isolation is already present in Chrome and Firefox [42, 55], preventing sensitive information from other webpages from being allocated in the attacker's address space. While its implementation is an ongoing effort by Apple [3, 4], site isolation is not currently on production releases of Safari. On the contrary, we also reflect on `libpas`'s heap layout from Section 6.3, allowing sites to not only share processes, but also heaps. Partitioning JavaScript heaps by at least several memory pages per-webpage would prevent JavaScript strings from the target webpage from being allocated within the 255-byte reach of the LAP.

Future Mitigations. In the short term, a practical mitigation from Apple would be to disable the LAP when the DIT bit is set, following its precedent with the DMP. Doing so would allow developers to disable the LAP in sections of code handling secrets without affecting other programs (since its state is kept per-process), and subsequently re-enable it for non-sensitive code regions to leverage the LAP's performance benefits on data dependencies.

For future microarchitectures, prior work in side-channel defense literature [20, 59, 72] proposes hardware support for marking memory regions or operands as holding secrets, and for speculation to abort if the CPU ever accesses marked data. We acknowledge this would make exploitation more challenging, since speculation would terminate before our gadgets can retrieve a marked secret and encode it over microarchitectural covert channels. However, this method requires vendors to release new hardware with extensive additions to circuitry, while requiring developers to update their software to explicitly mark sensitive data.

8. Conclusion

In this paper, we discover a new mechanism for data speculation in the form of load address predictors in recent Apple CPUs. The LAP can issue loads to addresses that have never been accessed architecturally and transiently forward the values to younger instructions in an unprecedentedly large window. We demonstrate that, despite their benefits to performance, LAPs open new attack surfaces that are exploitable in the real world by an adversary. That is, they allow broad out-of-bounds reads, disrupt control flow under speculation, disclose the ASLR slide, and even compromise the security of Safari. In the landscape of the decline of Moore's Law birthing more exotic microarchitectural optimizations, we believe that LAPs may not be an Apple exclusive, either now or soon. As such, we emphasize the need for novel hardware and software countermeasures against LAPs in future work.

Acknowledgments

This research was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-24-1-0079; the Alfred P Sloan Research Fellowship; an ARC Discovery Project number DP210102670; the Defense Advanced Research Projects Agency (DARPA) under contract numbers W912CG-23-C-0022, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972; and gifts from Cisco and Qualcomm.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking chrome strict site isolation via speculative execution. In *IEEE SP*, 2022.

[2] Apple. Energy efficiency guide for mac apps: Prioritize work at the task level. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx/PrioritizeWorkAtTheTaskLevel.html, 2016.

[3] Apple. Webkit contributor meeting 2022. https://docs.webkit.org/Other/Contributor%20Meetings/ContributorMeeting2022.html, 2022.

[4] Apple. Process swap on cross-site window.open behind a flag. https://github.com/WebKit/WebKit/pull/10169, 2023.

[5] Apple. Apple silicon cpu optimization guide: 3.0. https://developer.apple.com/documentation/apple-silicon/cpu-optimization-guide?changes=_9, 2024.

[6] Apple. Internals - webkit documentation. https://docs.webkit.org/Deep%20Dive/Libpas/Internals.html, 2024.

[7] Apple. xnu/external_headers/mach-o/loader.h at main. https://github.com/apple-oss-distributions/xnu/blob/94d3b452840153a99b38a3a9659680b2a006908e/EXTERNAL_HEADERS/mach-o/loader.h#L84, 2024.

[8] Apple. xnu/bsd/kern/mach_loader.c at main. https://github.com/apple-oss-distributions/xnu/blob/main/bsd/kern/mach_loader.c#L550, 2024.

[9] Apple. xnu/osfmk/arm64/proc_reg.h at main. https://github.com/apple-oss-distributions/xnu/blob/main/osfmk/arm64/proc_reg.h#L1286, 2024.

[10] Apple. xnu/osfmk/vm/vm_map.c at main. https://github.com/apple-oss-distributions/xnu/blob/94d3b452840153a99b38a3a9659680b2a006908e/osfmk/vm/vm_map.c#L21090, 2024.

[11] ARM. How is instruction timing affected by the feat_dit architectural feature? https://developer.arm.com/documentation/ka005181/latest/, 2024.

[12] Asahi. Hw:arm system registers. https://github.com/AsahiLinux/docs/wiki/HW%3AARM-System-Registers, 2023.

[13] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *USENIX Security*, 2022.

[14] Harry Stefan Barowski and Rolf Hilgendorf. Universal load address/value prediction using stride-based pattern history and last-value prediction in a two-level table scheme, January 10 2006. US Patent 6,986,027.

[15] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *ACM CCS*, 2019.

[16] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *ACM CCS*, 2019.

[17] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, 2019.

[18] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. Gofetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers. In *USENIX Security*, 2024.

[19] Yuan C Chou, Viney Gautam, Wei-Han Lien, Kulin N Kothari, and Mridul Agarwal. Early load execution via constant address and stride prediction, November 28 2023. US Patent 11,829,763.

[20] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. ProSpeCT: Provably secure speculation for the constant-time policy. In *USENIX Security*, 2023.

[21] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime + Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *USENIX Security*, 2017.

[22] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *ACM ASPLOS*, 2018.

[23] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the Spectre era. In *ACM CCS*, 2020.

[24] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *DIMVA*, 2016.

[25] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *IEEE SP*, 2011.

[26] Jann Horn. Speculative execution, variant 4: Speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[27] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *IEEE SP*, 2015.

[28] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. SPOILER: Speculative load hazards boost Rowhammer and cache attacks. In *USENIX Security*, 2019.

[29] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The gates of time: Improving cache attacks with transient execution. In *USENIX Security*, 2023.

[30] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *DAC*, 2016.

[31] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. iLeakage: browser-based timerless speculative execution attacks on apple devices. In *ACM CCS*, 2023.

[32] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[33] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, 2019.

[34] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT*, 2018.

[35] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. Value locality and load value prediction. In *ACM ASPLOS*, 1996.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[37] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of AMD's cache way predictors. In *ACM AsiaCCS*, 2020.

[38] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, 2015.

[39] Andrei Lutas and Dan Lutas. Security implications of speculatively executing segmentation related instructions on Intel CPUs. https://businessresources.bitdefender.com/hubfs/noindex/Bitdefender-WhitePaper-INTEL-CPUs.pdf, Aug 2019.

[40] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *ACM CCS*, 2018.

[41] Hector Martin. Found the dmp disable chicken bit. https://social.treehouse.systems/@marcan/112238385679496096, 2024.

[42] Mozilla. Project Fission. https://wiki.mozilla.org/Project_Fission, 2021.

[43] Ryosuke Niwa. Reduce the precision of "high" resolution time to 1ms. https://github.com/WebKit/WebKit/commit/25e575313d12e97a9e6c2b1d9b6ddd1d510e01a9, 2018.

[44] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *ACM CCS*, 2015.

[45] Bodo K Parady. Threshold-based load address prediction and new thread identification in a multithreaded microprocessor, June 14 2005. US Patent 6,907,520.

[46] PassMark. Passmark - intel core i5-2500t @ 2.30ghz. https://www.cpubenchmark.net/cpu.php?cpu=Intel+Core+i5-2500T+%40+2.30GHz&id=806, 2024.

[47] Arthur Perais and André Seznec. Practical data value speculation for future high-end processors. In *IEEE HPCA*, 2014.

[48] Arthur Perais and André Seznec. Bebop: A cost effective predictor infrastructure for superscalar value prediction. In *IEEE HPCA*, 2015.

[49] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[50] Filip Pizlo. All about libpas, phil's super fast malloc. https://github.com/WebKit/WebKit/blob/main/Source/bmalloc/libpas/Documentation.md, 2024.

[51] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *ACM CCS*, 2021.

[52] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, 2021.

[53] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *IEEE SP*, 2021.

[54] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. PACMAN: Attacking Arm pointer authentication with speculative execution. In *ISCA*, 2022.

[55] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: process separation for web sites within the browser. In *USENIX Security*, 2019.

[56] Rami Sheikh and Derek Hower. Efficient load value prediction using multiple predictors and filters. In *IEEE HPCA*, 2019.

[57] Rami Sheikh, Harold W. Cain, and Raguram Damodaran. Load value prediction via path-based address prediction: avoiding mispredictions due to conflicting stores. In *IEEE/ACM MICRO*, 2017.

[58] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[59] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ACM ASPLOS*, 2019.

[60] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[61] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[62] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE SP*, 2020.

[63] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why stopping cache attacks in software is harder than you think. In *USENIX Security*, 2018.

[64] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Rogue in-flight data load. In *IEEE SP*, 2019.

[65] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxe.com/files/SGAxe.pdf, 2020.

[66] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *USENIX Security*, 2021.

[67] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. In *IEEE SP*, 2022.

[68] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *IEEE SP*, 2019.

[69] Haonan Wang, Mohamed Ibrahim, Sparsh Mittal, and Adwait Jog. Address-stride assisted approximate load value prediction in gpus. In *ACM ICS*, 2019.

[70] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. https://foreshadowattack.eu/foreshadow-NG.pdf, 2018.

[71] Johannes Wikner and Kaveh Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. In *USENIX Security*, 2022.

[72] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *IEEE/ACM MICRO*, 2018.

[73] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

Appendix A. LAP Behavior on Page Boundaries

Does the LAP Train Across Page Boundaries? We now seek to determine if the LAP maintains training state across page boundaries. To do so, we focus on two specific cases of the linked list workload from Section 4.4, where all use a stride of 128 bytes, in Figure 19.

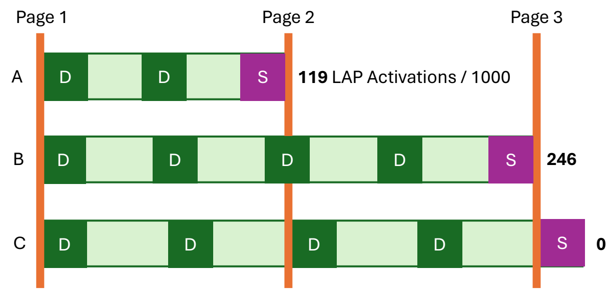


Figure 19: Diagram of how the buffer used for training the LAP is allocated across pages for each case, along with the number of activations to the right. The 'D' and 'S' squares represent dummy and secret values, respectively.

The page size on Apple CPUs is 16 KiB. Here, Case Ⓐ trains the LAP on 128 nodes, such that training occurs across one page and the predicted load address is near the end of the page (offset `0x3f80`). Case Ⓑ uses 256 nodes, training across two pages: likewise, the predicted load address is near the end of the second page. Hence, we compare the number of LAP activations across cases Ⓐ and Ⓑ. According to the heatmap in Figure 9, doubling the training length from 128 to 256 generally doubles the number of activations. As case Ⓑ has 128 training addresses on the first and second pages each, we expect to observe about double the activations if the LAP keeps state across pages, and the same otherwise.

We present the number of activations for each case as the bolded number to the right of each memory diagram in Figure 19. Comparing Ⓐ with Ⓑ, we answer the question in the affirmative from the approximately twofold increase coming from another page of training addresses.

Does the LAP Predict Across Page Boundaries? In addition, we determine if the LAP will generate a prediction on a new page. We start with Case Ⓑ from Figure 19, and add Case Ⓒ for comparison. Its 257 nodes (instead of 256) make training take place across two pages, but cause the predicted load address to be on a new page. Viewing the number of activations again, we answer the question in the negative this time, as we see no activations when the predicted address is on a new page.

Appendix B. Confirming Instruction Address Tagging

We use the training window experiment from Section 4.5 as the groundwork, and the same training parameters (1,000 loads striding 32 bytes apart) to optimally activate the LAP. We now describe our modifications to first test for the presence of an instruction address tag, and then test whether the tag is a partial or full match.

Effects of Loop Unrolling. Firstly, we observe 997 LAP activations on Listing 4 with no `mul` instructions inserted. Then, we direct the compiler to completely unroll the for-loop in Line 2, and manually inspect the resulting binary. Here, we observe 0 LAP activations when the training loads are not from the same program counter. This observation indicates that the M2 LAP does indeed tag training states using some or all parts of the instruction address, and not the global history of load addresses.

The Control Experiment. Now, we aim to determine if the LAP uses a partial or full address match. To do so, we first design a control experiment where we divide training into two phases with a short interruption in between (which we know the LAP can tolerate from Section 4.5) and ensure that we can still activate the LAP. During the interruption, we write a different secret value to the LAP's predicted address. Our goal is to place the interruption where the LAP's training state persists. That is, the second phase alone should not be able to activate the LAP. However, when we run both phases, we should receive the new secret value.

We start by putting Listing 4 in a function. We call the function to traverse the first $1000 - x$ linked list nodes for the first training phase. Then, we write the new secret value. Finally, we call the function again to traverse the last x nodes using the same instruction addresses. We increment x from 1 to 30 and measure the number of LAP activations (where we receive the new secret value) in Figure 20 (Left). From the plot, we conclude that $x = 30$ causes the LAP to reliably maintain training state.

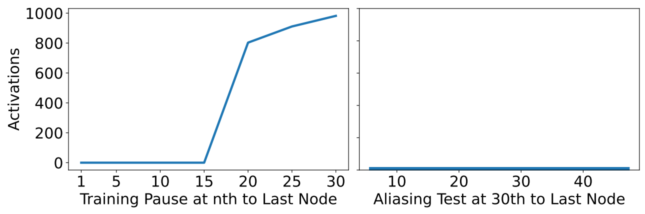


Figure 20: (Left) Number of LAP activations when training is interrupted across a function call, starting at the last linked list node and ending at the 30th-to-last. (Right) Number of LAP activations when all but the last 30 nodes are traversed by the original function, and the rest are traversed by the aliased clone function.

Aliasing the Second Training Phase. Next, we create a clone of this function. We use linker scripts for `ld` to place the two functions apart at a fixed offset to alias the instruction addresses, ranging from the lowest 6 to 47 address bits. We traverse the first 970 nodes using the original function and write the new secret value, identically to the control experiment. However, we then traverse the remaining 30 nodes using the cloned function. With this setup, we observe 0 LAP activations for all aliasing offsets, as we show in Figure 20 (Right). Therefore, we confirm that the LAP looks for a full match of all canonical address bits, since a partial match would lead to the LAP's training state persisting (and thus the new secret being transmitted) at some point during the experiment.

Appendix C. Timer-Resilient Covert Channel

In order to make cache hits distinguishable from misses in Safari, we reference the NOT gate-based cache amplification primitive from [29, Section 5], adjusting the speculation parameters for the M2 CPU. We run the amplifier 500 times when the target address is cached and 500 more times when it is evicted, in native and WebAssembly implementations. Table 3 summarizes the timing distributions, with units in ms. We observe that they are clearly separable even in a web environment, allowing us to distinguish cache hits from misses with WebKit's 1 ms timer.

Test	Cached Avg. Time	Stdev.	Evicted Avg. Time	Stdev.
Native	5.12	0.30	9.40	1.14
WASM	7.54	0.51	11.87	1.17

Table 3: Average runtime and standard deviation (both in milliseconds) for the NOT gate-based cache amplification primitive on native and WebAssembly runtimes on the Apple M2 CPU.

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

This paper shows that some Apple CPUs use load address prediction, a data speculation technique that can increase performance in the presence of data hazards, but that also opens up the possibility of new speculative execution attacks. The paper characterizes the conditions under which load address prediction activates, estimates the speculation window that it gives rise to, and develops exploitation techniques. The paper succeeds in building an end-to-end attack that targets a web browser and succeeds in reading cross-origin data.

D.2. Scientific Contributions

- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

D.3. Reasons for Acceptance

1) The paper identifies an impactful vulnerability in several widely used processors. It rigorously reverse engineers and documents the data speculation technique that these vulnerable processors use, and it develops practical exploitation techniques showing how to exploit the vulnerability.
2) The paper provides a valuable step forward in an established field. While the study of speculative execution vulnerabilities is well-established by now, this paper studies data speculation which is less well-understood than control flow speculation.