# FLOP: Breaking the Apple M3 CPU via False Load Output Predictions

Jason Kim*
*Georgia Tech*
nosajmik@gatech.edu

Jalen Chuang*
*Georgia Tech*
jchuang@gatech.edu

Daniel Genkin
*Georgia Tech*
genkin@gatech.edu

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

## Abstract

To bridge the ever-increasing gap between the fast execution speed of modern processors and the long latency of memory accesses, CPU vendors continue to introduce newer and more advanced optimizations. While these optimizations improve performance, research has repeatedly demonstrated that they may also have an adverse impact on security.

In this work, we identify that recent Apple M- and A-series processors implement a load value predictor (LVP), an optimization that predicts the contents of memory that the processor loads before the contents are actually available. This allows processors to alleviate slowdowns from Read-After-Write dependencies, as instructions can now be executed in parallel rather than sequentially.

To evaluate the security impact of Apple's LVP implementation, we first investigate the implementation, identifying the conditions for prediction. We then show that although the LVP cannot directly predict 64-bit values (e.g., pointers), prediction of smaller-size values can be leveraged to achieve arbitrary memory access. Finally, we demonstrate end-to-end attack exploit chains that build on the LVP to obtain a 64-bit read primitive within the Safari and Chrome browsers.

## 1 Introduction

The computer industry is witnessing an ongoing paradigm shift in the desktop and server markets, where x86-based Intel and AMD CPUs are now competing with a growing portfolio of Arm-based CPUs from Apple, Qualcomm, and Amazon. As new contenders in the market, many of these heavyweight Arm CPUs are notable for being clean-sheet designs, differing significantly not only in instruction set but also in microarchitecture from x86 CPUs.

However, clean-sheet designs are not free of clean-sheet problems. For instance, Apple's M-series CPUs have been reported to exhibit security issues pertaining to novel CPU features such as data-dependent prefetching [12, 57], pointer authentication [46], and instructions that allow for timerless cache attacks [63], in addition to more traditional issues that have also plagued x86-based CPUs such as speculative execution [18, 25] and data-dependent throttling [51].

In a never-ending quest to improve performance, architectures also have attempted to streamline the execution of dependent instructions. One such proposed optimization aims to alleviate slowdowns from Read-After-Write (RAW) dependencies. These occur when younger instructions read from the same location that an older instruction writes to, forcing the CPU to serialize these instructions. Accordingly, several works propose mechanisms that de-serialize RAW dependencies through prediction [2, 9, 21, 23, 30, 41, 42, 48, 49, 59], resulting in performance speedups.

With Spectre [27] and followups [1, 8, 10, 15, 20, 25, 28, 34, 46] demonstrating the security hazards of predictions, in this paper we ask the following question:

*How are RAW dependencies handled on emerging CPU designs? What optimizations do they entail and what are their security implications?*

## Our Contributions

In this paper, we discover a prediction mechanism for RAW dependencies that, to the best of our knowledge, was previously unseen in the wild. We show that Apple's M3, M4, and A17 Pro CPUs all optimize RAW dependencies via a load value predictor (LVP), which observes data values returned from load operations. If the values are constant, these CPUs can open a speculation window the next time this load executes, rather than waiting for the result to become available after a RAW dependency resolves. Within the speculation window, the predicted load values can be forwarded to arbitrary younger instructions that depend on them, thus causing the CPU to compute on the predicted value under speculation.

After characterizing Apple's LVP implementation, we proceed to demonstrate its security implications especially when the mispredictions cause transient computation using stale load values. Here, we show how the LVP causes type confu-

* Equal contribution joint first authors.

sion attacks in Safari and hijacks control flow in Chrome. In both cases, we package our primitives into end-to-end attacks that read arbitrary 64-bit addresses, allowing us to recover sensitive data across webpage origins.

**Discovery and Characterization of the Apple LVP.** We begin with an experiment that serially reads from a randomly shuffled collection of memory addresses and measures the running time. Remarkably, despite serializing the instruction via a RAW dependency, the M3 CPU runs drastically faster when the data stored in our memory addresses is a constant, compared to when it is randomly generated. Next, we proceed to reverse engineer the LVP's activation criteria, finding that the Apple LVP activates only for constant load values as opposed to striding ones, and keeps training state per instruction address for up to 72 different addresses. Moreover, it predicts arbitrary values only for 4-byte-wide loads and smaller, and not for 8-byte loads (which could be pointer values).

Subsequently, we demonstrate a primitive that causes the LVP to mispredict and speculatively compute on a stale value. Here, 250 training loads suffice to train the LVP with enough confidence for reliable mispredictions, with the resulting speculation window lasting up to 330 cycles. Next, we add a layer of indirection, causing the stale value to select and dereference an incorrect pointer, or even perform incorrect function calls. Thus, we demonstrate that LVP-induced speculation violates memory safety not only with 64-bit out-of-bounds reads, but also by diverting execution to rogue functions that are never architecturally invoked. Finally, we also show that the LVP can be mistrained in kernel-space assuming the existence of suitable gadgets, exacerbating the security risks.

**Exploiting LVP Mispredictions in Safari.** Going beyond LVP activations in the OS kernel, we show the practical security implications of the LVP by demonstrating 64-bit out-of-bounds reads in Safari, despite Apple's recent hardening attempts. To that aim, we use a gadget that accepts an object, which has a string member. The gadget dereferences the 64-bit pointer to the backing store of the string, and transmits the value it reads through a cache covert channel. In execution, we pass an array that contains raw binary data instead of the expected object type, but we exploit the LVP to confuse the processor about the input type.

Consequently, the CPU transiently executes the gadget on the wrong object type. Thus, instead of dereferencing a pointer to a string, the processor uses the attacker-controlled data in the array, confusing the CPU to read from an address of the attacker's choosing and transmitting the read value through a covert channel. With this speculative type confusion primitive, we run the FLOP-Data attack end-to-end, recovering the target's location history from Google Maps, inbox content from Proton Mail, and events stored in iCloud Calendar.

**Exploiting LVP Mispredictions in Google Chrome.** Going beyond disrupting data flows, we present the FLOP-Control attack, which causes the CPU to execute the wrong WebAssembly function under speculation in Google Chrome. In this attack, we call functions indirectly from a dynamic function dispatch table. Firstly, we mistrain the LVP on the table index such that the CPU retrieves the wrong dispatch object. Then, we mistrain the LVP again in a nested manner, such that the CPU will mistakenly validate the function arguments against the dispatch object when they are in fact invalid. This results in the CPU branching to a rogue function, which then computes on unchecked arguments.

When this nested LVP misprediction happens, architecturally, we provide a 64-bit integer argument to a function which takes a 64-bit integer. However, the LVP causes control flow under speculation to be misdirected to a function taking a `struct` reference, which is implemented as a 64-bit pointer in Chrome. Therefore, we cause the CPU to confuse data with an address, leaking the target's credit card information and billing address on Square storefronts.

**LVP and DIT Interaction.** Finally, we discover that the Arm ISA's Data Independent Timing (DIT) bit disables the LVP on the M3 CPU on a per-process basis, with no privileges needed to set the bit. See Section 7 for details.

**Summary of Contributions.** We contribute the following:

- We identify that an LVP exists on recent Apple CPUs, reverse engineering the preconditions for activation. We demonstrate a gadget to cause load value mispredictions, showing how the LVP can cause loss of control flow and data integrity under speculation (Section 4).
- We weaponize the LVP to perform a speculative type confusion attack on Safari, escaping its sandbox and recovering sensitive data from popular web services (Section 5).
- We weaponize the LVP again to transiently execute the wrong function in Chrome, demonstrating another sandbox escape and recovery of secrets (Section 6).

**Responsible Disclosure.** We disclosed our results to Apple's Product Security Team on September 3, 2024. Apple has acknowledged our disclosure and is continuing to investigate our report. For more details, see Section 9.

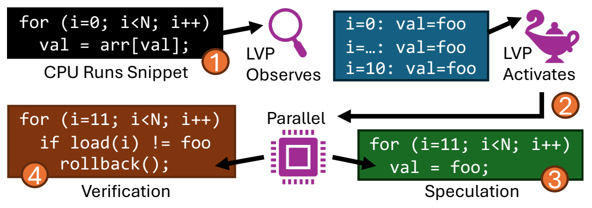
## 2 Background

**Cache Side-channel Attacks.** Virtually all modern CPUs use caches, which are small data buffers on or close to the CPU cores. For data that is used recently or frequently by the cores, the cache reduces memory access latency. However, this also means an adversary on the same system or core can time accesses to data and gain information about a target program's memory access patterns. Previous works demonstrate several techniques to do so, with some examples being FLUSH+RELOAD [17, 62] and PRIME+PROBE [13, 22, 32, 38, 43, 44, 58].

**Out-of-Order and Speculative Execution.** Another ubiquitous performance optimization beyond caches is speculative and out-of-order execution, which allows processors to deviate from program-induced instruction order, particularly when arguments are not readily available. When an execution reaches a branch whose outcome cannot be immediately resolved, the CPU attempts to generate predictions based on prior behavior, and continue executing instructions down that path until the correct control flow can be computed. However, this implies that CPUs can transiently execute the wrong instructions or operands. The Meltdown [31] and Spectre [27] attacks pioneer the insight that such transient execution carries grave security consequences, due to microarchitectural state not being completely reverted in case of mispredictions. This has since resulted in numerous followup, breaking nearly all hardware-backed security domains [1, 8, 10, 11, 15, 20, 25, 26, 28, 33, 34, 35, 45, 46, 52, 53, 54, 55, 56, 61].

**Read-After-Write Dependencies.** In a never-ending quest to improve performance, computer architectures also have attempted to streamline the execution of dependent instructions. While most dependencies can be solved via on-the-fly register renaming, Read-After-Write (RAW) is a fundamental data dependency for all pipelined CPUs when an older instruction writes to the same location that is read from by a younger instruction. Typically, the CPU must run the older instruction to completion before the younger one can execute, because its operand cannot be computed before that point. Accordingly, RAW dependencies result in a slowdown of the pipeline, prompting computer architects to propose value prediction [14, 23, 40, 41, 42]. After observing values from instructions repeatedly, a CPU with value prediction can speculate past RAW dependencies by executing younger instructions with the predicted value.

**Load Value Prediction.** Within value prediction, Load Value Predictors (LVP) are among the most proposed [7, 9, 30, 39, 48, 49, 59], because not only do loads (and stores) occur frequently in program code, but they vary in latency greatly from <10 cycles for L1 cache hits to hundreds of cycles from main memory. Mitigating this, Figure 1 presents an outline of a typical LVP mechanism.



```
for (i=0; i<N; i++)
  val = arr[val];
```

```
i=0: val=foo
i=…: val=foo
i=10: val=foo
```

```
for (i=11; i<N; i++)
  if load(i) != foo
    rollback();
```

```
for (i=11; i<N; i++)
  val = foo;
```

Figure 1: Overview of load value prediction.

We begin at ① in the figure, depicting a loop with a RAW dependency. That is, the next index into the array to load from cannot be determined until the previous load completes and its value is saved into the `val` variable[1]. The LVP observes values being returned from the loads, oftentimes from the same instruction address as is the case for loops. If the loads predictably return the same value (such as `foo`) more than a set number of times as indicated in ②, the LVP activates.

Once activated, the LVP causes the CPU to perform two tasks, ③ and ④, in parallel. The first task ③ is to speculatively run the rest of the code using the predicted load value `foo` for the `val` variable, bypassing the loads caused by indexing into the array. The second task ④ is to execute the loads one by one, where we denote `load(i)` as the value returned from the i-th load in the loop. If all of them equal `foo`, the results of the first task are committed to the CPU's architectural state. Conversely, if there is a mismatch (e.g., from another core modifying the array), the speculative results are reverted and execution is replayed at i=11. Therefore, ③ can proceed until ④ either completes or rolls back.

## 3 Threat Model and Setup

We assume a typical browser-based threat model for our attack in Sections 5 and 6, where the target user runs a web browser and visits an attacker-controlled webpage. Furthermore, we focus on recently released Apple CPUs. Here, we assume the target system is a Mac with an Apple silicon CPU running macOS 14.5, Safari 17.5, and Chrome 128.0 with out-of-the-box settings. All versions are the most recent at the time of writing. Finally, we do not modify settings of side-channel countermeasures, leaving them in their default state.

## 4 Analysis of the Apple M3 LVP

### 4.1 Ascertaining the LVP's Presence

We now describe our procedure to test for load value prediction. Initially, we allocate a large buffer called `mem` that spans multiple cache lines. For our experiment, we fill this buffer with a constant value such that the value of a load from anywhere in the buffer will be predictable. The pseudocode in Listing 1 operates on this `mem` buffer.

```
1 memset(mem, CONST, MEM_SIZE);
2 int offsets[ITERS] = getOffsets(mem, ITERS);
3 shuffle(offsets);
4 flushBuffer(mem);
5 uint64_t start = getCycles(), junk = 0;
6 for (int i = 0; i < ITERS; ++i)
7     junk = junk + *(mem + offsets[i]);
8 uint64_t end = getCycles();
9 return end - start;
```

Listing 1: Our routine to measure the memory access latency on randomly shuffled addresses, where the load values may be identical or random.

We measure the number of CPU cycles to access several random cache line-aligned offsets (multiples of 128 bytes) comprising the `mem` buffer, where `ITERS` is the number of offsets accessed. To this aim, Line 1 `memset`s the entire buffer

[1]We note that this pointer-chasing scheme is in fact a RAW dependency in the CPU's register file, even though main memory is not written to.

such that any byte-wide load from `mem` will return `CONST`. Line 2 computes the aligned offsets, storing them in the `offsets` array. Then, Line 3 randomly shuffles the offsets, making the access pattern unpredictable to avoid activating hardware prefetchers. Line 4 flushes all of `mem` from the cache. By doing so, we ensure each access is a cache miss, creating favorable conditions for an LVP to activate (if one exists) to alleviate the slowdown in performance. After Line 5 obtains a timestamp, Lines 6-7 perform byte-wide loads to the offsets into `mem` in a loop. Here, to cause the loads to run serially, we insert a RAW dependency as the `junk` variable. Finally, Line 8 takes the second timestamp, and Line 9 returns the elapsed cycles.

We compare this experiment against our control, where we replace the `memset` in Line 1 with filling randomly generated values into the `mem` buffer such that the load values from `mem` will be unpredictable instead of constant. Figure 2 shows a graphical representation of the load addresses and values handled by Lines 6-7 in our experiment and control.

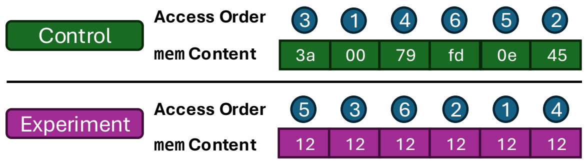


Figure 2: Comparison of our experiment against its control. While the access order for load addresses is always random, we vary the content of the `mem` buffer to be constant in our experiment and random in our control.

**Experimental Setup.** For precise measurements, we need kernel-level support for counting CPU cycles, pinning programs to CPU cores, and flushing cache lines. As these features are not available on macOS by default, we use Apple's Kernel Debug Kit (KDK) for macOS 14.5 build 23F79 to run macOS with the KDK's development kernel, and specify the details in Appendix A. With this setup, we run the control and experiment on the Apple M2 and M3 CPUs. All modern Apple CPUs have heterogeneous core designs, packaging a combination of high-performance P-cores and energy-efficient E-cores. Therefore, we consider each core type separately since they vary significantly in microarchitecture. We vary the `ITERS` parameter in increments of 10 from 10 to 500, and we use the median of 100 samples for each data point.

**Results.** Figure 3 shows the resulting plots. On the M2's P-cores (top left), our experiment does not result in particularly faster runtimes compared to our control. That is, the runtimes increase linearly with the number of loads performed, due to the RAW dependency between instructions, and does not depend on the values loaded from memory being randomized or constant. Thus, we conclude that an LVP is not present. In contrast, the runtimes from the M3's P-cores (top right) show a significant speedup from 40 iterations and above on our experiment when the load values are constant. At the maximum of 500 iterations, our workload only takes about half the cycles in that case compared to when the load values

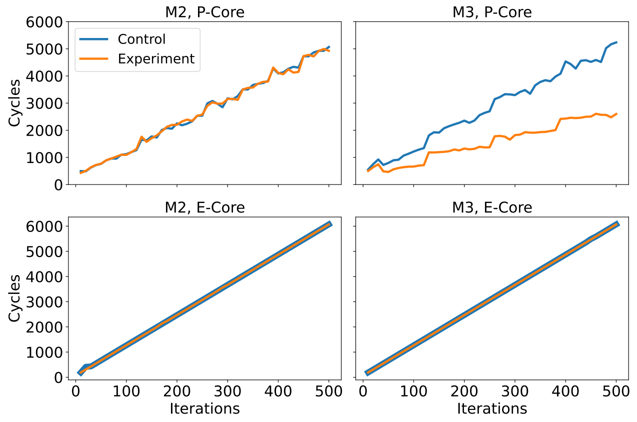


Figure 3: Runtimes of the gadget in Listing 1 on each CPU and core type. The control and experiment plots heavily overlap for the E-cores.

vary randomly. As such, we determine the P-cores are indeed equipped with an LVP, alleviating the performance penalty from RAW dependencies. Moving on to the E-cores of both CPUs (Figure 3 bottom), these generate plots that do not differ significantly by load value, indicating the lack of LVPs.

**LVP Presence on Other Apple CPUs.** After determining that the LVP is present on the M3 and not the M2, we seek to extend our experiments to more Apple CPUs. To that aim, we create a portable version of Listing 1 by compiling it to WebAssembly. Next, we fix the `ITERS` variable to 10 million to make the runtime difference measurable using Safari's coarse timer (1 ms). From this, we indicate whether each CPU has an LVP on Table 1. That is, the LVP is present on the M3, M4, and A17 and absent on the M2, A15, and A16.

| Apple CPU | Tested Device | Has LVP? |
|---|---|---|
| M2 | MacBook Air (A2681) | ✗ |
| M3 | MacBook Pro (A2918) | ✓ |
| M4 | iPad Pro 7th Gen. (A2926) | ✓ |
| A15 Bionic | iPhone 13 Mini (A2481) | ✗ |
| A16 Bionic | iPhone 14 Pro Max (A2651) | ✗ |
| A17 Pro | iPhone 15 Pro (A2848) | ✓ |

Table 1: Survey of LVP presence on recent Apple desktop and mobile CPUs.

## 4.2 Identifying LVP Activation Criteria

Having established the LVP's existence, we now investigate its behavior for load instructions of different width, whether it detects striding load values (as opposed to constants) and its response to loop unrolling. We compare the outcomes of these changes against our initial M3 P-cores observations (top right of Figure 3), which is copied over and shaded in gray. Finally, we also identify how many predictions the LVP mechanism can manage simultaneously.

**Width of Loads.** We repeat the measurements from Section 4.1, but change the pointer dereference in Line 6 of Listing 1 to have a load instruction of varying widths. We test for a speedup compared to our control when using 1, 2, 4, and 8-byte loads, and for each of the 100 samples we vary the constant value that every load returns. See Figure 4.

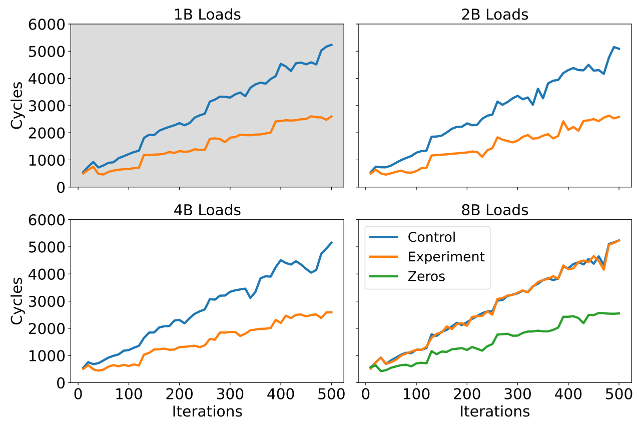


Figure 4: The effect of load width on LVP activation for the M3 CPU.

Remarkably, while all other load widths activate the LVP on any constant value fitting that width, we observe that activation on 8-byte wide loads occurs only when the load value is zero. We conjecture that this may be a countermeasure for memory safety such that the LVP will not learn values of pointers. That is, with the M3 being a 64-bit CPU, pointers are 8 bytes wide. Furthermore, on 64-bit macOS executables, any virtual address below `0x100,000,000` is invalid [3].[2]

**Striding Load Values.** To test if the LVP learns load values that change with a fixed stride, we insert a subroutine in Listing 1. Between Line 3 where we randomize the access order and Line 4's flushing of all cache lines in the `mem` buffer, we write values to `mem` incrementing from 0 in the order obtained from Line 3. As before with load widths, we repeat measuring but with striding load values on the right of Figure 5, and compare it with constant load values on the left.

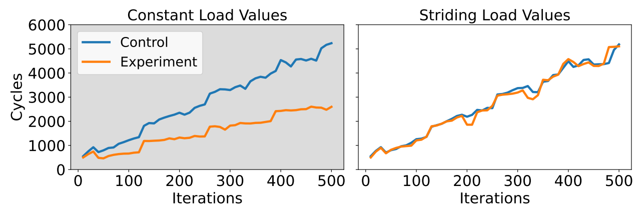


Figure 5: Runtime when the M3 CPU loads values that stride, instead of the same value on each load. Our control continues to load random values.

Here, we observe that the runtime as the number of loads increases resembles the trend when the load values are random. As such, we conclude that Apple's implementation of the LVP trains only on constants.

**Loop Unrolling.** Without modifying the source code from Listing 1, we direct the compiler to fully unroll Lines 6-7 to exactly `ITERS` different load instructions for each value of `ITERS` (10 to 500, in increments of 10). This causes a load placed at a particular instruction address to be executed only once per trial, as opposed to tens or hundreds of times. We plot the runtimes of our control and experiment on the unrolled binary to the right of Figure 6, and contrast them with the original loop on the left.

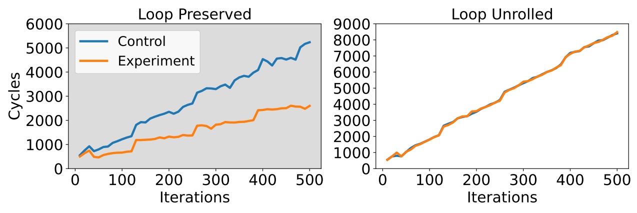


Figure 6: Effect of loop unrolling on LVP activation. We attribute the longer latency on the unrolled plot to the much larger binary size, as more instructions must be fetched by the CPU.

Notably, unrolling the loop causes the runtime on constant load values to scale nearly identically to our control with random load values, indicating that the LVP does not activate. Hence, we conclude that LVP trains with local scope (presumably with entries tagged with the instruction address) instead of training on a global window of recent load values.

**Simultaneous Prediction Capacity.** Lastly, after noticing the LVP's local scope, we seek to determine how many distinct load instructions the LVP can track and activate on. To that aim, we create several copies of the `mem` buffer from Listing 1, shuffling the access order individually. For the traversing loop in Lines 6-7, we copy the load instruction in Line 7 such that each line traverses a distinct clone of `mem`, and we also insert $n$ dummy instructions between these loads to test if the LVP's capacity is dependent on the instruction address bits (akin to set-associative caches). Then, we divide the timestamp in Line 8 by the number of load instructions to record the average traversal time. For each value of $n$, we increase the number of copies that simultaneously train the LVP until we observe the experiment's average traversal time spike towards that of the control (indicating that the LVP failed to predict at least one copy). We plot the results in Figure 7.

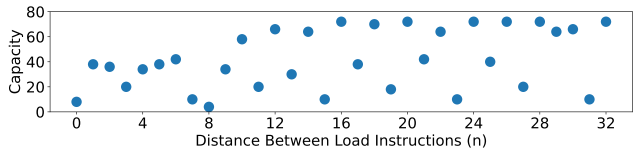


Figure 7: Maximum number of distinct load instructions the LVP can support before a timing spike is observed, depending on the instruction distance between them. In the 64-bit Arm ISA, every instruction is 4B wide.

Interestingly, we observe that the maximum number of copies the LVP can accommodate is 72, but this also depends on how far the load instructions are spaced apart. Rerunning the experiment with ASLR disabled, we observe the same

[2]We note however that this strategy is not entirely safe, as LVI-NULL [53] demonstrates transient execution attacks using address zero.

result. Thus, we conjecture the LVP's internal state cache may be 4-way set-associative (as the LVP could always predict at least four addresses regardless of the value of $n$) and uses a hash function on the page offset bits to determine the set.[3]

## 4.3 Measuring Mispredicted Load Values

Moving away from LVP-induced timing differences, we now investigate if the LVP uses its prediction to compute speculatively on arbitrary downstream instructions. Listing 2 outlines our experiment for measuring LVP speculation.

```
1  uint8_t gadget(int offset) {
2      return *(mem + offset);
3  }
4  // LVP training on load value foo
5  int offsets[ITERS] = getOffsets(mem, ITERS);
6  shuffle(offsets);
7  uint8_t foo = 0xca, bar = 0xfe, val = 0;
8  memset(mem, foo, MEM_SIZE);
9  for (int i = 0; i < ITERS; ++i)
10     gadget(offsets[i]);
11 // Make LVP mispredict bar as foo
12 memset(mem, bar, MEM_SIZE);
13 flushBuffer(mem);
14 val = gadget(offsets[0]);
15 frTransmit(val);
16 return frRecv();
```

Listing 2: Code snippet for measuring mispredictions. We train the LVP on the load value foo, but then change the architectural value to bar. We cause the LVP to still operate on foo (which is now stale).

**Gadget Overview.** First, we focus on the `gadget` function in Lines 1-3, where we perform loads from the `mem` via a function that is never inlined. This ensures that the LVP-training load in Line 2 is always at the same instruction address, following our results from Section 4.2 which suggest PC-tagging. Here, the `mem` buffer is identical to our experiments in Section 4.1.

**Training the LVP.** Subsequently, Lines 4-10 constitute the training routine. After we initialize and shuffle the order of accesses into `mem` in Lines 5 and 6, we declare values for the `foo`, `bar`, and `val` variables in Line 7. Then, we call the `memset` function on the buffer with `foo` in Line 8, such that all loads from `mem` will result in `foo` being returned. Indeed, this is how we train the LVP in Lines 9 and 10 by invoking the gadget to load `foo` several times, where we control the number of training loads with the `ITERS` parameter.

**Misprediction with Stale Load Values.** Next, we induce a misprediction in Lines 11-16. We call `memset` again but with the value `bar` in Line 12, changing the architectural load value from anywhere in the `mem` buffer to `bar`. Line 13, as before, flushes the whole buffer to induce the LVP to use its prediction instead of waiting for the load to resolve from memory[4]. We then call the gadget just once in Line 14.

This time when executing `gadget`, the CPU cannot retrieve the data for the load quickly, since it misses the cache. As the LVP has observed `foo` being returned from the load instruction in Line 2 before, the LVP uses `foo` as its prediction. Thus, `gadget` returns `foo` to be stored in the `val` variable in Line 14, and then in Line 15 we use FLUSH+RELOAD to encode `val` into the cache state such that we can recover it later. Next, at some later point, the CPU realizes the correct load value in Line 2 is `bar`, when it arrives from main memory. Hence, it rolls back the incorrect execution, returning `bar` into `val` and then encoding `bar` into the cache. Finally, in Line 16, we recover the encoded values, where we observe both `foo` (the stale load value) in addition to `bar`. See Figure 8.

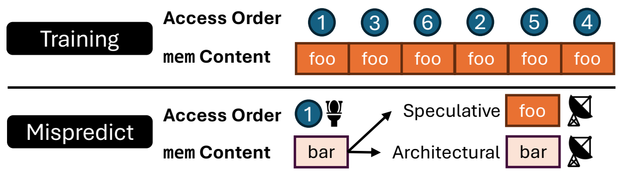


Figure 8: Graphical summary of the code in Listing 2. The LVP's misspeculation and subsequent rollback causes `foo` to be transmitted, followed by `bar` (indicated by the satellite icons).

**Measuring the Activation Threshold.** With the above gadget, our initial goal is to observe mispredictions reliably. Thus, we seek to identify how many training loads are necessary on the M3's P-cores. We vary the `ITERS` from 10 to 400 in increments of 2. For each value of `ITERS`, we run the code in Listing 2 1,000 times and plot the number of times we observe `foo` (the stale load value from the LVP) over the covert channel. Figure 9 plots the resulting misprediction counts.

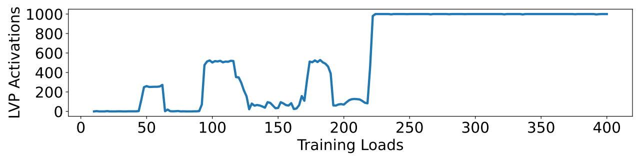


Figure 9: Effect of the number of training loads on the number of observed LVP activations with the stale load value (out of 1000).

We observe three spikes of activity before achieving reliable mispredictions. The first spike occurs around 60 loads, where the LVP activates about 25%. The second and third spikes occur around 120 and 180 loads respectively, with approximately 50% activation rate. Finally, past 240 loads, we observe activations with near-perfect reliability. Thus, in subsequent experiments, we train the LVP on 250 loads. Given that the previous spikes all occur around multiples of 60, we conjecture that the Apple LVP prefers training loads that are multiples of 60 until 240 loads, at which point it builds enough confidence to activate reliably regardless of training length.

[3] Our attacks use the same instruction address to train and exploit the LVP. We leave the reverse engineering of this hash function to future work.

[4] To rule out the effects of predictive store-to-load forwarding, this flush operation is serializing, writing the stores from `memset` back to main memory.

Furthermore, above 240 loads, we observe that LVP activations are reliable even when thrashing the cache on all cores, leading us to conjecture that value-prediction state is stored in a dedicated microarchitectural buffer instead of cache lines.

**Measuring State Persistence.** Suppose we insert extra instructions between Lines 11 and 12 of Listing 2. Then, these instructions execute after LVP training completes, but before we run `gadget` once more to make the LVP mispredict. Hence, they allow us to test for conditions that cause the LVP's internal state to either persist or attenuate. Thus, between Lines 11 and 12, we insert a busy-waiting loop of increasing duration to test for temporal conditions (such as a time-to-live field). In tandem, we test for the LVP's persistence with and without memory-intensive workloads, where for the latter we run stress-ng's `vm` workload concurrently. We measure the number of mispredictions out of 100 trials in Figure 10.

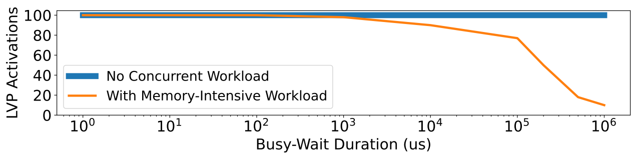


Figure 10: Number of observed LVP mispredictions when busy-waiting between training and misprediction, with and without a memory-intensive workload running on the same CPU core.

Observing Figure 10, we obtain reliable LVP activations regardless of the presence of stressors. However, the plots diverge at 10 ms, where the intensive load/store activity over time seemingly causes the activations to halve at 200 ms and decrease to one-tenth at 1 s. On the other hand, the LVP retains its internal state in the absence of stressors even after one-second busy waits (which, given the M3's peak frequency of 4.05 GHz [50], is more than four billion cycles). From this, we hypothesize that the LVP's state does not readily expire, but can be overwritten by memory activity on the CPU core over time. Furthermore, repeating this experiment with sending non-maskable interrupts instead of stress-ng, we identify that the LVP state is resilient to them after observing identical results to Figure 10 (Blue). On the other hand, if we replace the busy-wait with `sleep` to induce macOS to 'park' the CPU core in a low-power state, we observe that this resets the LVP state, resulting in no activations.

**Measuring the Speculation Depth.** Similarly to before, suppose we insert instructions between Lines 14 and 15 of Listing 2. During misprediction, the LVP transiently puts `foo` into the `val` variable in Line 14, and then transmits it over FLUSH+RELOAD in Line 15. Therefore, the inserted instructions become executed in the speculation window, allowing us to measure how long the LVP will compute on the predicted load value when the CPU's load target misses the cache. This time, we insert dummy `mul` instructions to keep multiplying the load value by 1 (in a data-dependent manner). We vary the number of `mul` instructions from 0 to 150 in increments of 5, and record the number of mispredictions out of 100 trials. In addition, we test for the LVP's speculation depth when the CPU's load target is cached by omitting Line 13 of Listing 2. We plot both speculation depths in Figure 11.

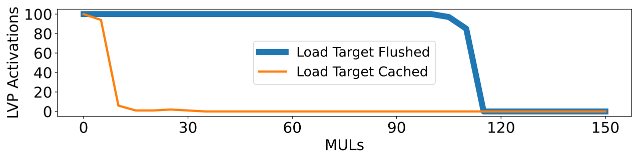


Figure 11: Number of observed LVP mispredictions when extra `mul` instructions are inserted between the speculative load and covert channel transmission. The speculation window is much larger when the load target is flushed.

Indeed, we observe the speculation window is much longer when the architectural load misses the cache, in which case we continue to observe LVP mispredictions up to 110 `mul` instructions. On the other hand, while the LVP still activates when the architectural load value is cached, we stop observing the stale load value being transmitted speculatively past 10 `mul`s. Given Apple documentation stating that each `mul` takes 3 cycles [4], our results translate to speculation windows of 330 (for cache misses) and 30 (for hits) cycles.

## 4.4 Inducing Memory Safety Violations

In this subsection, we investigate the security implications of LVP's computation on stale load values, which we observed in Section 4.3. Although the LVP does not predict arbitrary 64-bit values which could be pointers (cf. Section 4.2), we show that the LVP is dangerous when coupled with layers of indirection. That is, incorrect load values can be used to perform out-of-bounds reads throughout the address space, and also to call functions that are never invoked in the program.

**Reading Out of Bounds.** As the LVP does not learn pointer values, we aim to use the load value as an index into an array of pointers. We introduce minor changes to the `gadget` function of Listing 2 from Section 4.3, resulting in Listing 3.

```
1 uint8_t gadget(int offset, uint8_t *ptr) {
2     aop[foo] = ptr;
3     uint8_t val = *(mem + offset);
4     uint8_t *ptr = aop[val];
5     return *ptr;
6 }
```

Listing 3: Modified `gadget` function from Listing 2 that achieves 64-bit out-of-bounds reads. The LVP's stale load value causes the CPU to select the wrong pointer and dereference it.

In Line 1, we make `gadget` accept a pointer argument in addition to the offset into `mem`. Then, in Line 2, we insert this pointer into index `foo` of the array of pointers `aop`. In Line 3, we dereference `mem + offset` as before. Instead of returning the load value, we use it as the index into `aop` in Line 4, retrieving a pointer. Finally, in Line 5, we dereference the pointer and return the data at its address. Next, we show how the modified `gadget` function interacts with `aop` in Figure 12.

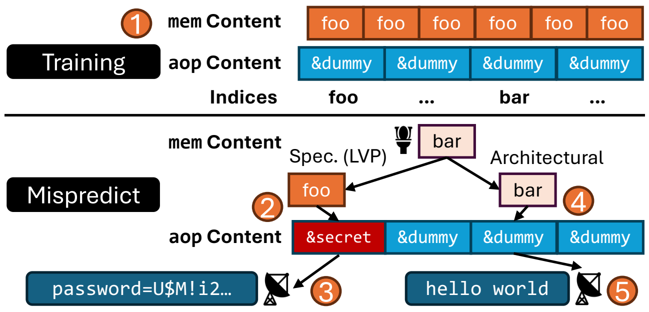


Figure 12: Graphical summary of our modified setup with indirection via an array of pointers to test for out-of-bounds reads during LVP speculation.

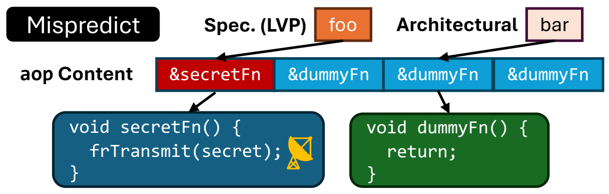
**Training Phase.** During LVP training (① in Figure 12), we fill `mem` with `foo`, and `aop` with the address of a dummy string. We run `gadget` with `ptr` set to `&dummy`, such that Line 2 does not change the contents of `aop[foo]`. Hence, when the LVP retrieves `foo` as the load value from `mem` in Line 3, the dummy string pointer is retrieved and dereferenced in Lines 4-5.

**Misprediction Phase.** Next, when we induce the misprediction in ②, we fill `mem` with `bar` and supply the address of `secret` to `gadget`, via `ptr`. Hence, `aop[foo]` now contains the secret pointer after Line 2. When the LVP activates in Line 3 because `mem` is flushed and assigns the stale load value `foo` to `val` instead of `bar`, this causes the secret pointer to be chosen and dereferenced in Lines 4-5. That is, `gadget` transiently returns data from the secret string, which then gets transmitted over FLUSH+RELOAD in ③. However, the CPU will eventually realize the correct load value is `bar` and replay the load into `mem`, as shown in ④. This results in the retrieval and transmission of the dummy string as well in ⑤, causing us to receive two values over FLUSH+RELOAD. Finally, since we know the dummy string's content, we can simply filter out the value corresponding to it to recover `secret`.

**Results.** Using 250 training loads as in Section 4.3, we perform the above setup to read the secret string for 100 trials. This results in mean and median accuracies of 0.97 and 1.00 (respectively) and throughput of 210,526 bits per second, demonstrating that the LVP can speculate into incorrect (and unsafe) data flows quite efficiently. Next, going beyond secrets in the current address space, we test for collisions in LVP state and LVP-induced memory safety violations across security boundaries in Appendix B. Our results indicate the LVP employs tagging using all instruction address bits and process ID, precluding its state from carrying over across processes. Moreover, after training the LVP in userspace, we observe that attempts to read kernel addresses are unsuccessful.

**LVP in the macOS Kernel.** One address space where out-of-bounds reads are particularly dangerous is that of the kernel. Thus, we ported our gadget from Listing 3 and Figure 12 into macOS's kernel space. We implement this as a kernel extension, wherein the training and misprediction phases can be invoked from a userspace driver program via `ioctl` syscalls. Remarkably, the LVP functions similarly in kernel-space: repeating the experiment to read a secret string (this time in kernel memory), we observe mean and median accuracies of 0.94 and 1.00 and throughput of 161,999 bits per second. While our scenario assumes the existence of suitable gadgets for LVP training in kernel space, we show that the a priori implications of in-kernel LVP activations are formidable.

**Branching to Rogue Functions.** We modify the out-of-bounds read experiment to determine if the LVP will also speculate into incorrect control flows. Firstly, we declare `aop` to hold function pointers instead of data pointers. Secondly, we replace Line 5 of Listing 3 with a call to the selected function pointer from `aop`, branching execution into the function's entry point. Thirdly, we replace the secret and dummy strings with secret and dummy functions with the same function signature (arguments and return type), as we show in Figure 13. This diagram takes the place of the bottom half of Figure 12.



```
void secretFn() {
  frTransmit(secret);
}
```

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

Figure 13: Modified lower half of Figure 12 to cause the stale load value `foo` to retrieve a function pointer and branch to it.

The secret function now contains the FLUSH+RELOAD transmission with a secret value. In contrast, the dummy function just returns. For our evaluation, we use 250 training loads and 100 trials as before, but measure how many times we correctly received the secret value and how many times the routine can be executed in one second. We receive the secret all 100 times and after a median of 0.000392 seconds: hence, our rogue function gadget can be run about 2,551 times per second. Therefore, we conclude that the LVP is also effective at diverting control flow under speculation.

## 5 Attacking Safari with the LVP

In this section, we study the security implications of the LVP on a major component of the Apple ecosystem: the Safari web browser and its underlying browser engine, named WebKit. Since JavaScript is a weakly typed language, WebKit is responsible for checking types of variables under the hood in order to determine the operations it can perform on the variable. By causing the LVP to mispredict the type, we orchestrate FLOP-Data, an end-to-end attack capable of reading sensitive data from a cross-origin target webpage.

## 5.1 Value Prediction on Variable Types

We start our discussion of the LVP's interaction with type checking in WebKit by introducing the typing mechanism. In WebKit, every JavaScript data structure starts with a header data structure named `JSCell`. Figure 14 shows its layout.

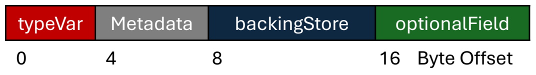


Figure 14: Memory layout of the `JSCell` header.

We note that the type information of the data structure is stored in the first 4 bytes of the `JSCell`. Therefore, for every object operation, WebKit starts executing the pseudocode in Listing 4, where it first performs a 4-byte load to the start of the `JSCell` to retrieve `typeVar` on Line 1.

```
uint32_t inpType = input->typeVar;
if (inpType != EXPECTED_TYPE)
    raiseException();
doSomething(input->backingStore);
doSomething(input->optionalField);
```

Listing 4: Pseudocode for the type checking procedure for JavaScript data structures in WebKit. The highlighted load trains the LVP.

Then, in Line 2, it compares the load value `typeVar` to the type that the code is expecting: if there is a mismatch, then the code aborts, throwing an exception in Line 3. That is, only after the type check passes will the code retrieve information from the rest of the `JSCell` such as `backingStore` and `optionalField`, as shown in Lines 4-5.

**Considering LVP Activation Criteria.** Now, we reexamine Line 1 of Listing 4 in the context of the LVP. As this is a 4-byte load, repeatedly running this code using an input of type `EXPECTED_TYPE` has the potential to train the LVP such that it predicts the load value will be `EXPECTED_TYPE`. That is, the edge case from Section 4.2 where the LVP does not predict arbitrary values for 8-byte loads does not apply.

Moreover, we reflect on our findings from Section 4.3: if we run Listing 4 again but with an input of a different type, the LVP may still use the (now stale) predicted load value for Line 1 and proceed to Line 2. The key insight is that the LVP's prediction will now cause the type check in Line 2 to incorrectly pass, resulting in the CPU proceeding to operate on the data structure (Lines 4-5). This paves the way for a speculative type confusion attack, which we now describe.

However, we also know from Section 4.3 that the load in Line 1 must miss the cache during the misprediction run for a prolonged speculation window that can reach Lines 4-5. In contrast, when execution reaches those lines, `backingStore` and/or `optionalField` from Figure 14 must be cached for speculation to continue into the `doSomething` subroutines. This necessitates the `JSCell` header of the attacking data structure to be split across two cache lines.

**Finding a JSCell Across Cache Lines.** The iLeakage attack [25] demonstrated that `Intl.Locale` objects can be allocated such that the `typeVar` and `backingStore` variables from Figure 14 are on different cache lines. However, Apple has since introduced a patch to WebKit [29] that ensures the two variables are always cache line-aligned, necessitating a novel attack vector. Accordingly, we turn our attention to the `optionalField` of the `JSCell`, which is only used by a small subset of JavaScript data structures compared to `backingStore`. It is this subset wherein we focus on typed arrays, a special class of arrays in JavaScript designed to handle raw binary data. We show their memory layout in Figure 15.

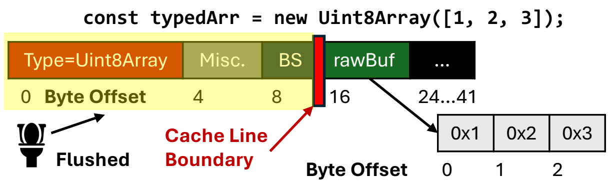


Figure 15: Memory layout of JavaScript typed arrays in WebKit. The Uint8Array (a class of typed array) declaration at the top of the figure produces this layout with a 3-byte buffer holding the data.

We observe that typed arrays leave the `backingStore` (denoted by `BS`) unused. Instead, the next variable (`rawBuf`) stores the address to their raw buffers. Furthermore, we observe that WebKit seldom allocates a typed array such that `Type`, `Misc.`, and `BS` are on one cache line, but `rawBuf` and some other metadata are on another cache line as shown in Figure 15. Thus, we can use typed arrays as the attacking data structure for speculative type confusion, since passing one to WebKit may activate the LVP when `Type` misses the cache.

## 5.2 Speculative Type Confusion

As an adversary, our end goal is to achieve a 64-bit read under speculation, which would allow us to recover secrets from anywhere in WebKit's address space. In this subsection, we describe the exploit chain that begins from confusing a malicious typed array as another data structure and ends at retrieving data from a pointer which we control.

**Target for Type Confusion.** First, we would like the CPU to parse a typed array as something else due to the LVP mispredicting. However, this other data structure cannot use `BS` because `BS` for our malicious typed array would be flushed from the cache, preventing speculation from continuing. We find our candidate from an optimization used by WebKit: JavaScript objects that contain only a small number of member variables, or properties. While most objects use `BS`, WebKit stores small objects inline in memory, using fields after `BS` to hold the properties. We juxtapose the memory layouts of typed arrays and small objects in Figure 16.

We observe that the attacker can modify the small object's `prop` variable by changing the properties contained by that object. Furthermore, `prop` overlaps with `rawBuf` of the typed

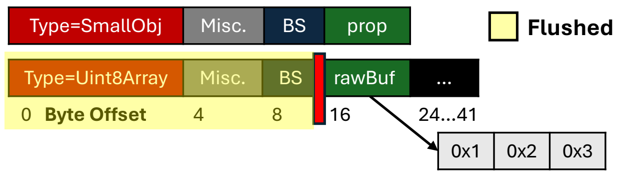
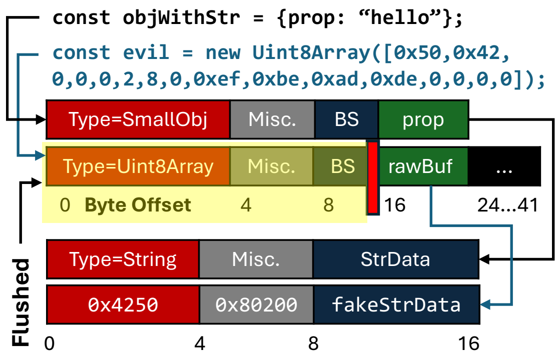


Figure 16: Memory layout of a small JavaScript object with an inline property (Top) and typed array split across cache lines (Bottom).

array in the memory layout. However, even though `prop` is 64 bits wide, we observe that the attacker cannot control all 64 bits of it due to WebKit's sandboxing measures. That is, to prevent attacker-controlled JavaScript values from resembling pointers, WebKit poisons the values when storing them in memory such that they would never represent valid addresses. Therefore, a level of indirection is necessary.

**Indirection with Strings.** Consider the two JavaScript data structures presented in Figure 17. The top object `objWithStr` is a small object from Figure 16 (top), modified to contain a string. The bottom object `evil` is a typed array. Eventually, we would like the CPU to misinterpret `evil` as `objWithStr` under speculation, due to type confusion.



```
const objWithStr = {prop: "hello"};
const evil = new Uint8Array([0x50,0x42,
0,0,0,2,8,0,0xef,0xbe,0xad,0xde,0,0,0,0]);
```

Figure 17: Memory layout of a small JavaScript object with a string (`objWithStr`) and typed array containing a malicious payload designed to imitate the string (`evil`). In this figure, the string's backing store (pointed to by `StrData`) is not shown for simplicity. We note that the Apple M3 is a little-endian CPU, hence we write the raw bytes in reverse order.

We augment `objWithStr` with a string after observing that strings do not contain poisoned values. Furthermore, we observe that `rawBuf` of `evil` aligns in memory with the string's metadata, which `prop` now points to. Akin to any other data structure, operations on strings are preceded by a type check. Therefore, for the CPU to operate on `rawBuf` under speculation, we must fill `rawBuf` with a payload to imitate the string's metadata. Forging a data structure is usually a difficult task, as WebKit attempts to randomizes the numerical values corresponding to each data structure's type. However, as builtin types (e.g., strings) are allocated in a fixed order, we observe that the numeric type assigned to strings is always `0x4250`, and the miscellaneous data value in byte offsets 4-7 constantly equals (`0x80200`). We write these values into `evil`, causing `rawBuf` to resemble the first 8 bytes of a string.

Next, we focus on the `StrData` variable of the string object. This is a 64-bit pointer to the string's underlying data structure, holding the string's length and character buffer (not shown for simplicity). Hence, by writing `fakeStrData` into `rawBuf`, it appears that we can create a fake 64-bit pointer to a string data structure. Despite this, we observe that WebKit does not expose the values from dereferencing `fakeStrData` to JavaScript code, as it instead parses the string's data structure for fields such as length, character buffer address, etc. This precludes our code gadget from directly obtaining 64-bit reads via `fakeStrData`, requiring us to examine the string data structure that `StrData` typically points to.

**JavaScript String Layout.** We describe the full memory layout of a JavaScript `string` object in Figure 18.

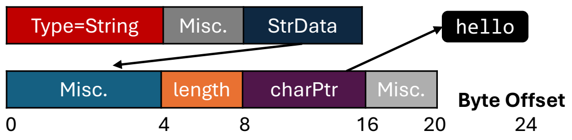


Figure 18: The full memory layout of the JavaScript string "hello", which is the `.prop` property of the `objWithStr` object in Figure 17 above.

The underlying data includes the `length` and `charPtr`, which points to the first character. When indexing into a string from JavaScript, the index is checked against `length`. If it is in-bounds, the index is added to `charPtr`, and the CPU loads from the resulting address to retrieve a character to JavaScript code. Hence, if WebKit dereferences an attacker-controlled value instead of `charPtr`, this immediately leads to our end goal of 64-bit reads. To that aim, we assume there is a secret at memory address `addr` and there exists an attacker-controlled buffer. As the miscellaneous values in Figure 18 surrounding `length` and `charPtr` are constants, we can make this buffer resemble the string's underlying data simply by writing these constants. Furthermore, we can write a large number for `length` such that any index would pass the check and write `addr` in place of `charPtr`. However, just as `StrData` points to this underlying data in Figure 18, we must write the address of this buffer in place of `fakeStrData` in Figure 17. As JavaScript does not have pointers, we now face a challenge to sidestep these sandboxing measures.

**Spraying Backing Stores on the Heap.** We address this problem using, rather ironically, another JavaScript string. First, we make the insight that the string's characters serve as an attacker-controlled contiguous buffer, where we can encode our payload from above. Next, strings can scale to massive lengths in WebKit, such as 1 GiB. Therefore, we repeat our payload to reach that length, effectively spraying multiple copies of the forged underlying data on the heap. Finally, we observe that declaring such a large string forces WebKit to allocate a new heap region. Remarkably, the new allocation consistently lands a couple bytes after `0x400000000`, making our spray technique extremely reliable. That is, we can replace `fakeStrData` in Figure 17's `evil` with `0x400000100` and expect this address to point to one of our sprayed instances. This completes our data structures for speculative type confusion, which we present in Figure 19.

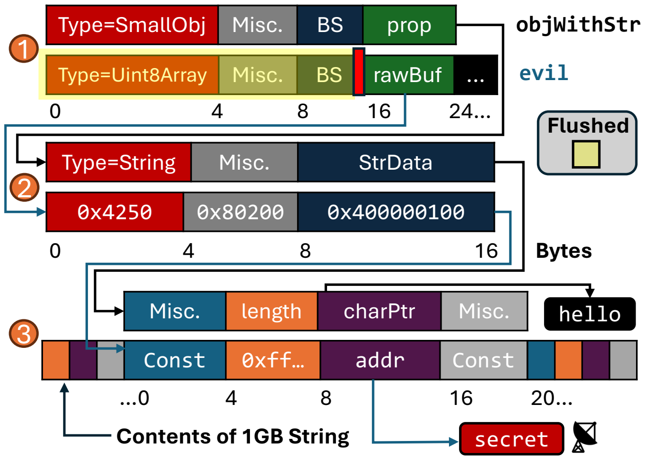


Figure 19: Our completed speculative type confusion primitive.

## 5.3 End-to-end Attack Overview

Using our data structures from Figure 19, we describe the remaining steps to orchestrate our attack end-to-end. First, we introduce our gadget in Listing 5.

```
function gadget(input, index) {
    const secret = input.prop.charCodeAt(index);
    channel.transmit(secret);
}
```

Listing 5: Our gadget that operates on the data structures in Figure 19 to achieve transient 64-bit reads via speculative type confusion in Line 2.

**Gadget Overview.** We train the LVP on Listing 5 with `objWithStr`, whose layout is represented at the top of each pair of memory diagrams in Figure 19, as the `input` argument. In Line 2 (highlighted), we retrieve characters using the `charCodeAt` function of `objWithStr`'s string (i.e., its `.prop` property) several times with an in-bounds `index`, resulting in WebKit reading from `charPtr`. We then provide `evil` as `input` and an arbitrary `index` to Listing 5, located at the bottom of each pair in Figure 19. Then, we reason about how the CPU (and thus LVP) reacts to Line 2.

**Type-confusing `evil` as `objWithStr`.** ① of Figure 19 shows the speculative type confusion, where the LVP causes the CPU to operate on `evil` while thinking the `input` is `objWithStr` because `evil`'s type is flushed. As `evil`'s `rawBuf` is cached, the CPU continues to speculate and interprets `rawBuf` as `objWithStr`'s `prop` variable. Thus, execution proceeds to ②, where the CPU first checks if it is computing on a string such that it can eventually retrieve a character. The type check on `rawBuf` passes as string, since we have written `0x4250` and `0x80200` to `type` and `misc`. Subsequently, the CPU interprets `0x400000100` as a pointer (in place of `strData`) and proceeds by dereferencing it.

**Achieving Transient 64-bit Reads.** We now revisit the heap spray at ③, where one instance of our forged string data, encoded inside a 1 GiB-long JavaScript string, is located at `0x400000100`. The CPU, thinking this data structure is a string, attempts to retrieve a character at `index`. As we have written the maximum possible value for `length`, we sidestep this check. Finally, the CPU interprets `addr` as `charPtr`, speculatively dereferencing the sum of `addr+index`, and returns the data to Line 2 under speculation. Finally, Line 3 leaks the data by encoding it in a microarchitectural covert channel. We describe the details of this covert channel and the full attack code implementation in Appendix C.

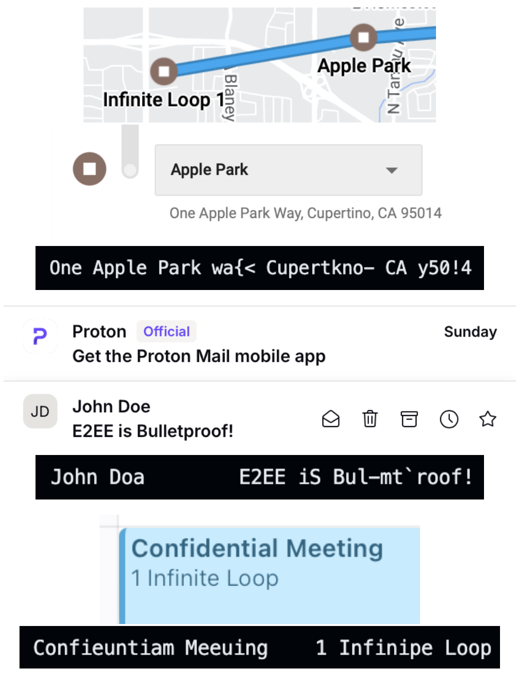
## 5.4 Evaluation

We first benchmark our 64-bit out-of-bounds read primitive, which we name FLOP-Data (a combination of the LVP and its ability to leak data). We host our attack code on a web server that is not publicly accessible. Then, we use FLOP-Data to read a buffer that we initialize with known data over 10 trials on a MacBook Pro with M3 CPU and 8 GB of RAM. Here, we observe a median accuracy of 89.58% and throughput of 0.492 bits per second. Subsequently, we evaluate the dangers of FLOP-Data on popular real-world websites. In these case studies, we assume the target user is authenticated into the target webpage via cookies stored by Safari.

**Bringing Secrets Into the Address Space.** Now that we can read from anywhere in the address space, we would like `addr` to point to meaningful targets, such as sensitive data from another webpage. As `addr` is a virtual address, we must cause the Safari process that renders the attacker's webpage to also render the target webpage. Prior work [25] reported that the `window.open` API in JavaScript achieves this, and also combined `window.open` with the `onmouseover` event listener to open the target page without triggering pop-up blockers when the target user puts their mouse cursor anywhere on the attacker's webpage. We confirm that both observations still hold true on the latest version of Safari. Hence, we cause mouse movement on our attacker webpage to call `window.open` with the target page's URL as an argument, such that secrets in the target page become allocated in our address space.

Furthermore, we must ensure that `addr` probabilistically points to those secrets. For large allocations such as webpage DOMs, we observe that WebKit expands its heap at deterministic addresses (such as `0x400000000` from Section 5.2) and that it enforces memory alignment of ≥16 bytes. This heavily reduces the entropy for a valid `addr`, oftentimes to brute-forceable 16-bit search spaces.

**Recovering Sensitive Data.** We wrap up our evaluation by showing FLOP-Data's ability to leak secrets from several web services. First, we notice that Google Maps' Timeline page tracks the locations the account owner has visited by default, recovering an address in Figure 20 (Top). Next, we recover the sender and subject of an email from the inbox of Proton Mail, an end-to-end encrypted email service, in Figure 20 (Middle). Finally, we head to Apple's own iCloud Calendar, where we recover the name and location of a private event in Figure 20 (Bottom).



One Apple Park Way, Cupertino, CA 95014

```
One Apple Park wa{< Cupertkno- CA y50!4
```

Proton Official Sunday
Get the Proton Mail mobile app

JD John Doe
E2EE is Bulletproof!

```
John Doa          E2EE iS Bul-mt`roof!
```

Confidential Meeting
1 Infinite Loop

```
Confieuntiam Meeuing     1 Infinipe Loop
```

Figure 20: Sensitive data recovered with FLOP-Data from Google Maps Timeline (Top), Proton Mail's inbox (Middle), and iCloud Calendar (Bottom).

## 6 Attacking Google Chrome with the LVP

In this section, we show that the LVP not only disrupts data flow in practical end-to-end attacks, but also control flow beyond our experiment in Section 4.4. We cause the CPU to execute the wrong function under speculation when running our WebAssembly gadget with Chrome, such that it treats a 64-bit integer as an address. Hence, we again achieve transient 64-bit reads to sensitive data throughout the address space.

### 6.1 WebAssembly Function Dispatch Table

First, we notice that Chrome prevents out-of-bounds reads for JavaScript objects by caging them in a 4 GiB memory region and limiting all pointers within this region to 32 bits. However, we observe that WebAssembly code is not subject to this caging restriction. Furthermore, we find that WebAssembly supports an interface where code can insert functions into a function dispatch table and subsequently invoke them by indexing into the table. We examine Chrome's implementation of the function dispatch table in Figure 21.

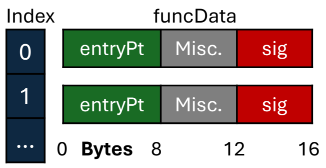


Figure 21: Memory layout of Chrome's function dispatch table.

**Dispatch Table Overview.** In Figure 21, each table index has a data structure `funcData`. `funcData` is comprised of `entryPt`, a code pointer to the function's entry point, miscellaneous metadata, and `sig`, a 32-bit value representing the function's signature (i.e., arguments and return type). Next, we show how functions are indirectly called in Listing 6.

```
1 function call(args, int index) {
2     funcData f = dispatchTable[index];
3     uint32_t funcSig = f->sig;
4     if (funcSig != EXPECTED_SIG)
5         raiseException();
6     setupArguments(args);
7     f->entryPt();
8 }
```

Listing 6: Pseudocode for calling a function using the WebAssembly function dispatch table in Google Chrome. The highlighted loads train the LVP.

WebAssembly code can call functions from the dispatch table by specifying a 32-bit `index` in Line 1, and `args`, the arguments for the function. Line 2 indexes into the table, retrieving the `funcData` at the index. In Line 3, the CPU performs a 32-bit load to obtain `sig`. Line 4 compares the result to `EXPECTED_SIG`, a unique value encoding the function's expected argument and return types. If there is a mismatch, Line 5 throws an exception to abort the code. Hence, Lines 4-5 safeguard the function at `entryPt` from being invoked with the wrong arguments. If this check passes, Line 6 sets up `args` and Line 7 branches to `entryPt` to execute the function.

### 6.2 Speculative Function Confusion

We recall our control-flow hijacking gadget from Section 4.4. There, we used the LVP to mispredict a stale index into an array of function pointers, thus transiently invoking the wrong function. We note that the function dispatch table in Figure 21 is very similar, with it being an array of `funcData` rather than pointers. Also, in Listing 6, we observe `index` is 32 bits and therefore learnable by the LVP (cf. Section 4.2).

Therefore, if `index` is not cached in Line 2, the LVP may activate on a stale prediction for `index`, leading the CPU to retrieve the wrong `funcData` under speculation. While this may seem to achieve our goal of invoking the wrong `entryPt`, it also implies that the value of `sig` is incorrect compared to what the code expects (`EXPECTED_SIG`) in Line 4. Unfortunately, this causes control flow to head to the exception in Line 5 instead of the wrong `entryPt` in Line 7.

**Bypassing Signature Checks.** However, we observe that `sig`, like `index`, is 32 bits. Thus, we aim to redirect the control flow one more time by training the LVP to mispredict on both `index` and `sig`. The loads for `index` and `sig` are at different instruction addresses, hence the LVP does not mix training state between them (cf. Section 4.2).

This requires `sig` not to be cached, such that the LVP will predict its value is `EXPECTED_SIG` and sidestep the argument check. However, if `entryPt` is also not cached, speculation will terminate in Line 7. Hence, we need to find an instance of `funcData` whose `entryPt` and `sig` are on different cache lines, and evict just the line containing `sig`. Fortunately, we discover such an instance by brute-forcing 32 table indices. This allows us to execute the wrong function on the wrong arguments, as we overview in Figure 22.

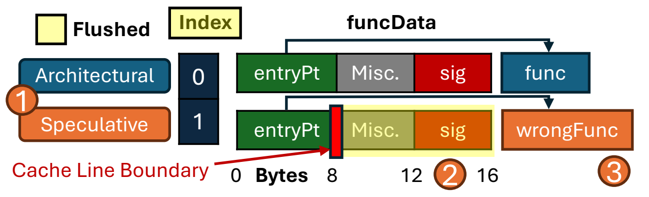


Figure 22: Our attack routine that causes two LVP mispredictions, disrupting control flow towards an incorrect function and also with the wrong arguments.

**Hijacking Control Flow.** As shown in Figure 22, we assume that the `funcData` split across cache lines is located at table index 1. We further assume that this `funcData`'s `sig` is not cached, as well as the `index` argument to Listing 6. Now, we invoke the `call` function with 0 for the `index` argument. In ① of Figure 22, the LVP first activates on Line 2 due to the cache miss on `index`. The LVP mispredicts `index` to be 1, retrieving the `funcData` of `wrongFunc` instead of `func`.

In ②, the CPU runs Line 3 to retrieve the signature. However, since this is also a cache miss, the LVP activates again, mispredicting that the signature will be that of `func`. This circumvents speculation from heading to the exception in Line 5. Therefore, in Line 6, we cause the CPU to set up `func`'s arguments for `wrongFunc` instead. Then, since the wrong `entryPt` remains cached in Line 7, the CPU incorrectly diverts control flow to `wrongFunc` in ③.

**Confusing Data as Addresses.** With the ability to transiently call rogue functions with unchecked argument types, we investigate the arguments that WebAssembly functions can take. Due to sandboxing in the browser, WebAssembly does not support pointers. However, it does support references to valid `structs` in memory. Although the reference is not attacker-controllable, we observe that Chrome remarkably implements it as a 64-bit pointer. Hence, we use `struct` references in lieu of pointers, aiming to provide an attacker-controlled 64-bit integer to a WebAssembly function that expects a `struct` reference. To that aim, we introduce Listing 7.

We first consider the `struct` in Line 1 that Chrome passes as a 64-bit pointer. Accessing the `data` variable dereferences

```
1  struct readType { uint8_t data; }
2  function func(uint64_t arg) { return; }
3  function wrongFunc(struct readType* arg) {
4      uint8_t readVal = arg->data;
5      channel.transmit(readVal);
6  }
```

Listing 7: Pseudocode for `func` and `wrongFunc` from Figure 22. We aim to run `wrongFunc` on a 64-bit integer, causing the CPU to treat it as a pointer.

that pointer, returning data at the `struct`'s address. Now, we consider the `func` and `wrongFunc` functions' implementation from Figure 22. We would like to run `wrongFunc` instead of `func` using `func`'s 64-bit integer argument, causing the CPU to confuse it with the 64-bit pointer to `struct`.

**Function Implementations.** `func` takes a 64-bit integer, but just returns, as shown in Line 2. In contrast, `wrongFunc` retrieves the data at the `struct` into `readVal` in Line 4, causing Chrome to perform a 64-bit pointer dereference. Then, in Line 5, `wrongFunc` uses a timer-resilient covert channel to transmit `readVal`, whose details we describe in Appendix D. Therefore, if we call `func` with a 64-bit integer but the CPU instead executes `wrongFunc` due to the LVP, the CPU will treat the integer as an address, and leak the data at that address through the cache. We name this primitive FLOP-Control, and describe the full implementation in Appendix E.

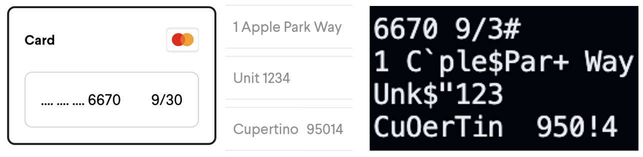
## 6.3 Evaluation

We first benchmark FLOP-Control by causing `addr` to point to a buffer we have initialized with known content. Next, we put Listing 7 on a non-publicly accessible web server and attempt to read the buffer for 10 trials using an M3 MacBook Pro with 8 GB of RAM. We report a median accuracy of 80.90% and throughput of 0.30 bits per second.

**Locating Target Websites.** Next, we would like `addr` to point to secrets from a target webpage. To do so, we must induce Chrome to render the target webpage in the attacker's address space. However, Chrome enforces restrictions on which webpages can be co-rendered due to site isolation [47]. Here, the attacker and target webpages are allowed to share an address space only if their extended top-level domains (eTLD) and the prefix before the eTLD are identical, a rule known as eTLD+1. An eTLD can be a traditional top-level domain such as `.com` or `.org` in addition to approximately 15,000 domains on the Public Suffix List [36] (PSL), which specifies domains under which users can create their own sites, such as `github.io`. For this example, the eTLD+1 rule prevents one Chrome process from rendering both `attacker.github.io` and `target.github.io`, as well as `attacker.org` and `target.org`.

Therefore, we seek for webpages that satisfy three conditions. Firstly, the webpage must not be on the PSL, such that `attacker.site.tld` can share an address space with `target.site.tld` (here, `site` is the common prefix). Secondly, the webpage must allow users to host their own JavaScript and WebAssembly on `attacker.site.tld`. Thirdly, `target.site.tld` must render secrets. Once we find such a target, we identify `addr` by running the attack in reverse: that is, we confuse addresses as data. First, as before, we locate a dispatch table entry that is split across cache lines. Then, by providing a 64-bit heap pointer to a function that transmits a 64-bit integer over our covert channel, we defeat ASLR from reading the pointer's value. This process takes approximately 30 seconds, constrained by our leakage throughput. Finally, breaking ASLR reduces the entropy for the secret's `addr` to 12 bits, allowing us to brute-force it.

**Attacking Square.** We find our target for FLOP-Control in the form of Square, a popular storefront service. The attacker can insert arbitrary JavaScript and WebAssembly into their storefront, which is hosted on `attacker.square.site`. Remarkably, Square is not on the PSL. This allows the attacker storefront to be co-rendered in Chrome with other storefront domains[5] by calling `window.open` with their URLs, as demonstrated by prior work [1]. One such domain is the customer accounts page[6], which shows the target user's saved credit card information and address if they are authenticated into the target storefront (see Figure 23 (Left)). As such, we recover the page's data in Figure 23 (Right).



Card

.... .... .... 6670 9/30

1 Apple Park Way

Unit 1234

Cupertino 95014

```
6670 9/3#
1 C`ple$Par+ Way
Unk$"123
CuOerTin  950!4
```

Figure 23: (Left) UI elements from Square's customer account page for a storefront. (Right) Recovered last 4 credit card number digits, expiration date, and billing address via FLOP-Control.

## 7 Mitigations

Now, we investigate how to mitigate the LVP-induced memory safety violations first introduced in Section 4.4 and weaponized throughout Sections 5 and 6. To that aim, we turn our attention to the Data Independent Timing (DIT) bit present in Armv8.4-A ISA and newer [6], and when set instructs the CPU that the latency to execute certain instructions should not correlate to the data in the operands.

Initially designed to protect constant-time code against dangerous microarchitectural optimizations [5], DIT is both readable and writable from unprivileged processes and is tracked across context switches [19]. Thus, changing the DIT bit for one process does not affect its value for other processes.

**DIT Interactions with the LVP.** With DIT already disabling data-dependent prefetching on the M3 [12], we now investigate if DIT also disables the LVP. We repeat the experiments of Section 4.1, setting the DIT bit before running any instructions. On the M3 CPU's P-cores, we no longer observe a speedup on constant load values compared to random load values. Subsequently, we also repeat the misprediction experiment of Section 4.3. Here, we do not receive the stale load value on the covert channel, regardless of how long we train the LVP. Therefore, we conclude that setting the DIT bit indeed disables the LVP. Hence, we recommend that developers patch their software to enable DIT on supported platforms, especially for code regions handling secrets or are untrusted.

For web browsers, this would correspond to setting the DIT bit when executing user-supplied JavaScript or WebAssembly, and on sensitive DOM operations such as password fields. As such, we patched Safari to set the DIT bit in the rendering process and observed that such a mitigation results in an overhead of 4.5% on the Speedometer 3.0 benchmark. In addition, we observe the overhead in native environments is even smaller, with 0.6% on average for our patched version of the BYTE Unix benchmark.

**Mitigations for Browsers.** In Safari, our cross-origin leaks of sensitive DOM data were possible due to WebKit rendering the attacker and target webpages in the same address space. Safari lacks Site Isolation [47], which is implemented in Chrome and mandates process isolation for webpages from different origins. However, Site Isolation does not solve the co-rendering problem completely due to corner cases, as we observed with Square in Section 6.3.

Next, we consider countermeasures for each browser engine. First, we suggest increasing the randomization entropy for types and memory allocations in Safari, as the lack thereof allowed us to create a fake string header and heap-spray the underlying string data[7]. Then for Chrome, we suggest caging the WebAssembly `structs` in a memory region, similarly to what it already does with JavaScript objects. This would allow references to them to be represented in base-relative offsets to that region (e.g., 32 bits) instead of 64-bit pointers, precluding an attacker from reading the whole address space[8].

## 8 Conclusion and Future Work

In this paper, we demonstrate that recent Apple CPUs perform load value prediction. After observing constant load values from a specific instruction address, the LVP bypasses subsequent loads, forwarding the same value to younger dependent instructions. This results in a speculation window that may compute arbitrarily on incorrect data. Using indirection, we show out-of-bounds addresses being read, the wrong function being executed, and finally sensitive login-protected data being exfiltrated across websites in Safari and Chrome. Finally, we identify that the LVP can be disabled via DIT.

[5]Square storefronts are generally hosted at `storename.square.site`.

[6]For each storefront, the customer account page is located at `storename.square.site/s/customer-account`.

[7]All types are stored in a sparse lookup table, and WebKit's memory allocation uses demand paging. Hence, we project the overhead to be negligible.

[8]The overhead is one addition, which is often just one CPU cycle.

On the other hand, we note that the DIT bit is not set (and thus the LVP is enabled) in the macOS kernel, where in Section 4.4 we demonstrated the possibility of in-kernel LVP training and exploitation. However, as we were not able to train the LVP across userspace / kernel boundary, to successfully exploit the LVP in a kernel environment, an attacker must find three types of gadgets: training gadgets containing loops of $\leq$32-bit loads, converter gadgets causing the wrong load value to access a secret, and finally leak gadgets to exfiltrate the secret over covert channels. We leave the task of creating tools for finding such gadgets, as well as creating realistic LVP kernel exploits, to 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 Qualcomm and Zama.

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.

## 9 Ethics Considerations

We describe our adherence to ethics guidelines while evaluating our attacks in Sections 5 and 6 and responsible disclosure.

**Ethical Testing of Attacks.** We own all devices used in our experiments, and these devices are free of any sensitive user data or personal information. These devices are only accessible to lab members, and are not exposed to unauthorized users. For online accounts, all of the accounts involved in our attacks are dummy accounts that we own, and are ensured not to contain any sensitive information. To reduce the risk to unrelated parties, we host the code for the attack on Safari (Section 5) on a local web server that is not publicly accessible.

To overcome Site Isolation [47], the code of the attack on Chrome (Section 6) has to be hosted in the same eTLD+1 domain as the target domain (cf. Section 6.3), which in our case is a dummy Square storefront that does not sell anything. To protect unrelated users that inadvertently visit our storefront, we modify the attack code to check for a specific cookie value and disable the attack if the value is not found. Consequently, running the attack requires opening Chrome's developer tools and manually adding a cookie, and thus we ensure it will not run on innocuous machines. Finally, we took down this storefront immediately after finishing our evaluation.

**Responsible Disclosure.** We disclosed our results to Apple's Product Security Team on September 3, 2024 upon completing the initial version of the writeup. Apple has acknowledged our writeup, and after an internal investigation, communicated that they plan to address this in an upcoming security update without sharing further details.

## 10 Open Science

We list all artifacts supporting this paper below. In adherence to the open science policy, we publish all of them at https://zenodo.org/records/14680908.

1. Source code, instructions, and compilation scripts for LVP reverse-engineering experiments (cf. Section 4)
2. Source code, binaries, instructions, and testing scripts for a browser-based read primitive on Safari (cf. Section 5)
3. Source code, binaries, instructions, and testing scripts for a browser-based read primitive on Chrome (cf. Section 6)

## 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] AMD. Security analysis of AMD predictive store forwarding. https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf, 2021.

[3] Gregory Anders. Exploring Mach-O, part 3. https://gpanders.com/blog/exploring-mach-o-part-3/, 2022.

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

[5] Apple. Writing ARM64 code for Apple platforms. https://developer.apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms, 2024.

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

[7] Sumeet Bandishte, Jayesh Gaur, Zeev Sperber, Lihu Rappoport, Adi Yoaz, and Sreenivas Subramoney. Focused value prediction. In *ACM/IEEE ISCA*, 2020.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Freddy Gabbay and Avi Mendelson. Using value prediction to increase the power of speculative execution hardware. *ACM Transactions on Computer Systems (TOCS)*, 1998.

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

[16] Google. Spectre. https://leaky.page, 2021.

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

[18] Lorenz Hetterich and Michael Schwarz. Branch different-Spectre attacks on Apple silicon. In *DIMVA*, 2022.

[19] Arm Holdings. Processor state in exception handling. https://developer.arm.com/documentation/100933/0100/Processor-state-in-exception-handling, 2024.

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

[21] Intel. Fast store forwarding predictor. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/fast-store-forwarding-predictor.html, 2022.

[22] 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.

[23] Kleovoulos Kalaitzidis and André Seznec. Leveraging value equality prediction for value speculation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2020.

[24] 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.

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

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

[27] 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.

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

[29] Mark Lam. Adjust preciseallocation alignment offset to also factor in cache line alignment requirements. https://github.com/WebKit/WebKit/commit/842bf586330dbf74f9e2d09d50c818ca3f792988, 2023.

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

[31] 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.

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

[33] 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.

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

[35] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *USENIX Security*, 2023.

[36] Mozilla Foundation. Public suffix list. https://publicsuffix.org/, 2024.

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

[38] 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.

[39] Lois Orosa, Rodolfo Azevedo, and Onur Mutlu. Avpp: Address-first value-next predictor with value prefetching for improving the efficiency of load value prediction. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2018.

[40] Arthur Perais and André Seznec. EOLE: Combining static and dynamic scheduling through value prediction to reduce complexity and increase performance. *ACM Transactions on Computer Systems (TOCS)*, 2016.

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

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

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

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

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

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

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

[48] 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.

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

[50] Anton Shilov. Apple M3 CPUs hit 4.05 GHz, challenge Raptor Lake in Geekbench. https://www.tomshardware.com/pc-components/cpus/apple-m3-cpus-hit-405-ghz-challenge-raptor-lake-in-geekbench, 2023.

[51] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. Hot Pixels: Frequency, power, and temperature attacks on GPUs and ARM SoCs. In *USENIX Security*, 2023.

[52] 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.

[53] 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.

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

[55] 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.

[56] 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.

[57] 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.

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

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

[60] Yoav Weiss and Eiji Kitamura. Aligning timers with cross origin isolation restrictions. https://developer.chrome.com/blog/cross-origin-isolated-hr-timers/, 2021.

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

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

[63] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. Synchronization storage channels (S2C): Timer-less cache side-channel attacks on the Apple M1 via hardware synchronization instructions. In *USENIX Security*, 2023.

## A macOS Development Kernel Setup

After installing the development kernel, we add `enable_skstb=1` to the boot arguments for the kernel to allow access to the `kern.sched_thread_bind_cpu` API in `sysctl` for setting core affinity. Then, we use a combination of kernel patches and extensions to allow user-space access to the `S3_2_c15_c0_0` system register (core cycles) and the `dc civac` (cache flush) instruction.

## B Extended Tests for Isolation

Having observed the LVP's potential to cause memory safety violations when mispredicting in one address space, we now extend those experiments to determine which security boundaries the LVP can cross, and if we can cause collisions in the LVP's internal state that can lead to training and exploitation happening in separate instructions or address spaces.

**Testing for Instruction Address Aliasing.** Earlier, in Section 4.2, we have observed that the LVP trains with local scope, failing to activate (and cause speedups) when the training loop is unrolled. We now test this more thoroughly using the `gadget` function from Sections 4.3 and 4.4. More specifically, we first clone the `gadget` function. Our goal is to train the LVP on the original `gadget`, then attempt to induce mispredictions when the cloned `gadget` executes.

To test for instruction address bits that are part of the page offset, we use the C `align` attribute to cause the least significant bits to alias. For more significant bits, we `mmap` code pages with the `MAP_FIXED` flag to map them at a specific page-aligned virtual address. Then, at the same page offset as the original `gadget`, we `memcpy` its instructions. These approaches allow us to alias from 8 to 46 least significant bits.

As a control, we train and evaluate the LVP only on the original `gadget` after implementing these changes, where we observe reliable mispredictions with 250 training loads. However, when evaluating on the cloned `gadget`, we no longer observe them regardless of the number of aliased bits. Hence, we conclude that the LVP likely tags its internal state with the full instruction address, precluding aliasing attacks.

**Testing for Process Isolation and ASID Collisions.** Now, we aim to run the training and misprediction routines in different processes. To that aim, we map the `mem` buffer with the `MAP_SHARED` flag such that writes to `mem` from one process become visible to the other. We also call `fork` just before training, using the child process to train the LVP. Subsequently, the parent runs the misprediction routine.

In a variant of this experiment, we keep forking until the parent and child processes share the same Address Space ID (ASID). An AArch64 feature, ASIDs are assigned by the kernel and are used to tag page table base registers among other microarchitectural features. ASIDs are either 8 or 16 bits wide, with the M3 implementing the former. Given that the process limit for unprivileged users in macOS is 2,666, it is straightforward to spawn at most 256 child processes and achieve two processes with different PIDs but the same ASID.

Again, as controls, we both train and evaluate the LVP in only the parent or child process and achieve reliable mispredictions. Nonetheless, we do not observe any signal from mispredictions when the two steps are performed in different processes even when the ASIDs collide, indicating that the LVP may employ PID-tagging in addition to PC-tagging.

**Reading Kernel Addresses from Userspace.** Finally, we augment Section 4.4's experiment with a kernel extension that initializes a secret string in kernel memory and discloses only its address. We then write that address into `aop[foo]` before causing the LVP to mispredict to determine if LVP activations from userspace instructions can also read kernel memory. However, we do not receive any data over the covert channel this time, possibly indicating that speculation may terminate when loading from a kernel address.

## C FLOP-Data Implementation Details

We present the full attack pseudocode that operates on Figure 19's `objWithStr` and `evil` in Listing 8.

```
1  function gadget(input, index) {
2      const secret = input.prop.charCodeAt(index);
3      channel.transmit(secret);
4  }
5  // Data structure setup
6  const objWithStr = {prop: "hello"};
7  const fakeStr = 0x4250 || 0x80200 || 0x400000100;
8  let evil = undefined;
9  for (let i = 0; i < 13; i++)
10     evil = new Uint8Array(fakeStr);
11 const spray = CONST || 2^32-1 || addr || CONST;
12 const str1GB = spray.repeat(SPRAY_SIZE);
13 // Training phase
14 for (let i = 0; i < TRAIN_REPS; i++)
15     gadget(objWithStr, 0);
16 // Attack phase
17 partialEvict(evil);
18 if (false) gadget(evil, leakIndex);
19 return channel.receive();
```

Listing 8: Pseudocode in JavaScript for our speculative type confusion attack. The `||` operator in Lines 9 and 13 represents concatenation.

**Covert Channel Details.** Once `addr` is transiently dereferenced after Line 2 and the data at `addr` is returned to the `secret` variable, we must consider that WebKit restricts the timer precision in JavaScript to 1 ms to deter fingerprinting and side-channel attacks [37]. This necessitates a cache amplification primitive for us to recover the secret using a cache covert channel. To that aim, we adapt a technique pioneered by Google's leaky.page attack [16] that amplifies one L1 data cache hit or miss into tens of thousands by leveraging the replacement policy. This amplified covert channel is represented as the `channel` variable in Line 3.

**Data Structure Setup.** Before invoking `gadget`, we initialize data structures for the attack. We declare `objWithStr` on Line 6 of Listing 8, and craft our payload for the typed array's data on Line 7. Then, in Lines 8-10, we declare 13 typed arrays with this payload, but discard all but the last declaration. Empirically, we observe that the 13th allocation usually creates a typed array instance split across cache lines for a newly spawned WebKit rendering process, and thus we assign this one to the `evil` variable. Next, Lines 11-12 spray the heap with the 1 GiB string that contains repeats of our second payload, which resembles the underlying data of a string. Line 11 forms the payload by sandwiching the largest possible value for `length`, followed by `addr`, between the miscellaneous constants. Line 12 constructs a new string that repeats Line 11 until the result is 1 GiB long.

**Training Phase.** Recalling from Section 4.3 that 250 training loads suffice to train the LVP reliably, we run the for-loop in Lines 14-15 with the `TRAIN_REPS` variable set to 250. Within the loop in Line 15, we repeatedly invoke `gadget` with 0 as the index such that it is in bounds. Also, we give `objWithStr` as the input, aiming to mistrain the LVP that the load value for `Type` should be `SmallObj`.

**Attack Phase.** Subsequently, in Line 17, we evict the first 16 bytes of `evil` from the cache to force the LVP into speculation, while keeping the rest cached for speculation to continue to `addr`. But now, we reason about WebKit's behavior after transmitting `secret`. Eventually, the CPU will realize the LVP's prediction was incorrect and replay execution from ① of Figure 19. Because we provided `evil`, a typed array, to code that was expecting `objWithStr`, WebKit will raise an exception that prevents us from using our attack more than once. To avoid this, we enclose the call to `gadget` on a mispredicted if-statement in Line 18 for speculative hiding [31]. This causes the exception to be raised under speculation, making the incorrect execution invisible to Safari. Finally, Line 19 recovers the secret via the amplified covert channel.

## D Cache Covert Channel Details

As a side-channel countermeasure, the timer resolution in Chrome is restricted to 100 $\mu$s [60]. Hence, to recover secrets encoded in the cache state, we require the ability to distinguish cache hits from misses even when the timer is orders of magnitude coarser than the memory access latency. To that aim, we adapt the timing amplification primitive from [24] which uses speculation windows to cascade one cache hit or miss into thousands of hits or misses, allowing us to distinguish the two outcomes even with 100 $\mu$s resolution.

## E FLOP-Control Implementation Details

We present the full attack pseudocode in Listing 9, which is the extended version of Listing 7.

```
1  struct readType { uint8_t data; }
2  function func(uint64_t arg) { return; }
3  function wrongFunc(struct readType* arg) {
4      uint8_t readVal = arg->data;
5      channel.transmit(readVal);
6  }
7  // Training phase
8  uint64_t addr = &secret;
9  dispatchTable[1].set(func);
10 for (let i = 0; i < TRAIN_REPS; i++)
11     dispatchTable.call(addr, 1);
12 // Attack phase
13 dispatchTable[0].set(func);
14 dispatchTable[1].set(wrongFunc);
15 int idx = 0;
16 evict(idx, dispatchTable[1].sig);
17 dispatchTable.call(addr, idx);
18 return channel.receive();
```

Listing 9: Pseudocode for our control flow attack on Chrome, where a mispredicted table index and signature cause a 64-bit number to be treated as a pointer. `dispatchTable.call` in the highlighted lines runs Listing 6.

**Training Phase.** We start by training the LVP on the table index and function signature. In Line 8, we assume there is a secret in our address space, but store its address as a

64-bit integer `addr`. In Line 9, we put `func` in table index 1 for LVP training, and we assume the `set` function updates `entryPt` and `sig`. Then, in Lines 10-11, we call `func` through the dispatch table, which causes Listing 6 to run. Hence, we train the LVP to predict 1 for the table index, and `func`'s `sig` value for the signature. We set the `TRAIN_REPS` parameter in Line 10 to 250, since we observed in Section 4.3 that 250 training loads suffice for reliable mispredictions.

**Attack Phase.** Next, we reassign `func` to table index 0 in Line 13. In Line 14, we cause index 1 to refer to `wrongFunc` now. Then, in Line 15, we set a zeroed variable `idx` to be passed as the dispatch table index. As shown in Figure 22, we now assume that `funcData` at index 1 can be partially evicted. Thus, in Line 16, we evict both `idx` and `sig` while keeping the entry point to `wrongFunc` cached. We call the dispatch table again in Line 17, where `func` should be called because `idx` is 0. However, `idx` is not cached, causing the LVP to mispredict the table index as 1 (① of Figure 22).

Under speculation, we sidestep the signature check from the second LVP misprediction, due to `wrongFunc`'s `sig` also not being cached. Therefore, the CPU regards `addr` as a valid argument, diverting control flow into `wrongFunc` along with `addr`. Then, `wrongFunc` treats `addr` as a `struct` reference, loading the secret at `addr` and leaking it through the cache. Subsequently, the CPU will realize it mispredicted the table index and execute Listing 6 architecturally, calling `func`. However, `func` just returns since it was provided a valid argument. Finally, the CPU runs Line 18, allowing us to retrieve the secret over the covert channel.