Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest

Jose Rodrigo Sanchez Vicarte[1*], Michael Flanders[1†], Riccardo Paccagnella*, Grant Garrett-Grossman*,
Adam Morrison‡, Christopher W. Fletcher*, David Kohlbrenner†
*University of Illinois Urbana-Champaign, ‡Tel Aviv University, †University of Washington
{josers2, rp8, grantlg2, cwfletch}@illinois.edu, mad@cs.tau.ac.il, {mkf727, dkohlbre}@cs.washington.edu

***Abstract*—Microarchitectural side-channel attacks are enjoying a time of explosive growth, mostly fueled by novel transient execution vulnerabilities. These attacks are capable of leaking arbitrary data, as long as it is possible for the adversary to read that data into the processor core using transient instructions.**

In this paper, we present the first microarchitectural attack that leaks *data at rest* in the memory system, i.e., never directly read into the core speculatively or non-speculatively. This technique is enabled by a previously unreported class of prefetcher: a data memory-dependent prefetcher (DMP). These prefetchers are designed to allow prefetching of irregular address patterns such as pointer chases. As such, DMPs examine and use the contents of memory directly to determine which addresses to prefetch.

Our experiments demonstrate the existence of a pointer-chasing DMP on recent Apple processors, including the A14 and M1. We then reverse engineer the details of this DMP to determine the opportunities for and restrictions it places on attackers using it. Finally, we demonstrate several basic attack primitives capable of leaking pointer values using the DMP.

I. Introduction

As the demand for performance gains in general purpose CPUs continues and the gains from Moore's scaling dwindle, microarchitects have consistently delivered surprising and powerful optimizations. However, these optimizations come with drawbacks, notably in how they inadvertently leak information via microarchitectural side channels.

Today's microarchitectural side channels are only capable of leaking *data in use*. That is, data is speculatively or non-speculatively *architecturally accessed* and then *transmitted* through an "unsafe instruction". For example, data might be read into an architectural register and then acted on by an instruction that changes hardware resource usage in an operand-dependent fashion [29, 30]. This restriction to *data in use* limits attacks in situations where victim programs lack access or transmit gadgets or when the programs are specifically written to not contain such gadgets as in constant-time programming [13, 14, 16, 19, 49].

Yet, there has recently been speculation that we are standing on a precipice, about to face an even more insidious threat: microarchitectural side channels that leak *data at rest* [44, 45]. These attacks are brought on by exotic microarchitectural optimizations such as silent stores [32], cache compression [37] and data memory-dependent prefetchers [50] all of which can leak data *even if it is never brought into the processor core*. Consider for example a processor that implements a *data memory-dependent prefetcher (DMP)*. Unlike well-known and widely implemented prefetchers whose behavior is "just" a function of a program's address pattern (limiting their leakage to program address pattern/control flow [17, 20, 42]), DMPs read and initiate cache fills *based on the contents of program data memory directly*.

This immediately puts all of program memory at risk. For example, Vicarte et al. [45] point out how a specific proposed DMP called the indirect-memory prefetcher [50, 51] can be coerced into leaking all of program memory, similar to Spectre/Meltdown but without relying on transient instruction execution. Making matters worse, as the DMP lives in the memory system, it accomplishes this without the secret data ever being read from the cache into the processor core, rendering current constant-time programming techniques ineffective.

Fortunately for defenders, data-at-rest attacks have been purely theoretical. While there is a rich literature on DMPs, there has been no evidence to suggest they have ever been implemented in commercial processors.

In this paper, we demonstrate for the first time the existence, and resulting security implications, of a DMP *in the wild. By extension, this shows that microarchitecture leaking data at rest is real.* We refer to our techniques as Augury due to their reliance on interpreting what the prefetcher believes about the future.

Specifically, we found that the Apple M1, M1 Max, M1 Pro, and A14 processors possess an *Array-of-Pointers* (AoP) prefetcher that recognizes streaming and striding reads and dereferences over an array of pointers, and then prefetches the result of dereferencing future pointers. To see the difference from conventional prefetchers, suppose a program is looping from `i=0...N` and has allocated an array `A` which is indexed by `i`. A conventional prefetcher would prefetch an access pattern such as `A[i]` or `A[stride*i]`. The M1's AoP DMP prefetches access patterns such as `*A[i]` or `*A[stride*i]`.

As the AoP DMP operates only on a stream of memory accesses, and does not have any concept of array bounds, this prefetcher can overshoot the legal set of pointers to access and attempt a prefetch of unrelated memory addresses up to its prefetch depth. *This act of dereferencing the out-of-bounds pointer (potentially even if it is not actually a pointer!) creates a memory side channel that an attacker can use to learn the pointer.* In fact, we show that this pattern recognition is relatively robust, can operate at large strides, and can trigger even if all memory accesses are speculative and eventually

[1]The two first authors contributed equally to the paper.

squashed. Together, these capabilities enable the attacker to target and leak pointer values across much of memory.

There have been many proposed DMP patterns, and since none have previously been found, there is little in the way of guideposts for understanding DMP behaviors. As there is no documentation for even the existence of the M1 AoP DMP, simply finding the activation pattern is a non-trivial matter.[1] Even then, knowing that an AoP DMP exists does not clearly lay out a plan for attack primitives or for software mitigations. To aid in this, we present a detailed analysis of the AoP DMP behavior and we also provide guidance for the reverse engineering and security analysis of *any* DMP system.

For attackers, this prefetcher opens up new, previously unconsidered exploitation scenarios. For defenders, the existence of this prefetcher, and the attacks it enables, is a call to action for developing new approaches for programming techniques that can protect data not even being operated on.

Contributions. Our major contributions are:

- We analyze the security relevant design factors for DMPs.
- We demonstrate the existence of the first known data-memory prefetcher in a commercial processor family.
- We reverse-engineer the activation criteria, depth of prefetch, and other features of the M1 DMP.
- We demonstrate that this prefetcher can be used to cause unexpected pointer de-references, putting data at risk.
- We demonstrate the first microarchitectural attacks on data at rest by using the M1 DMP to construct exploit primitives that leak pointer values.

Disclosure. We coordinated with Apple on disclosure and mitigation prior to publication.

Release of tools. Following disclosure, we have made our tools for investigating DMP behavior on ARM and x86 chips, as well as our proof-of-concept attack primitives available at: https://github.com/FPSG-UIUC/augury.

II. Background and Motivation

A. Classical prefetchers

We now review how hardware prefetchers work, and their status from a security perspective.

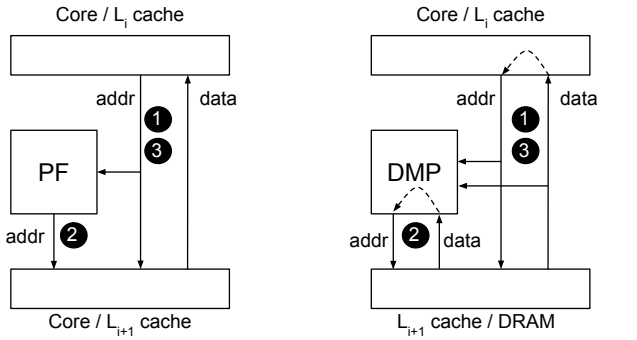

Fig. 1: A high-level architecture for classical prefetchers/PFs (left) and data memory-dependent prefetchers/DMPs (right).

[1]We thank Anandtech for their analysis of the A14 processor that speculated a "pointer-chase prefetch mechanism" [22] might exist in that processor.

To start, we review what we call *classical prefetchers* (prefetchers for short). These are widely deployed in commercial processors. For example, both Intel and AMD report multiple distinct prefetchers in their recent processors [2, 4].

Like prefetchers, caches are present in multiple forms and levels on nearly all modern processors. Generally, each processor core will have at least 1 level of private cache, split into an instruction and data cache. Below this, there is at least one level of shared cross-core cache, generally storing both data and instructions. By default, caches will exploit temporal locality to hide the latency of repeated accesses to memory by storing data recently used by the core.

Prefetchers are next-in-sequence address predictors that proactively fetch data into the cache to help hide memory latency. Figure 1 (left) gives a high-level overview. In Step ❶ (*training*), the prefetcher records whether the address sequence coming from the core matches a specific pattern. In Step ❷ (*prefetching*), if a pattern was recognized with sufficient confidence, the prefetcher autonomously makes accesses to memory to fill the cache with cache lines that it thinks will be requested in the near future. In Step ❸ (*validation*), the prefetcher checks whether its predictions were correct by checking the core's subsequent requests. Note, we separate the above steps to ease explanation; similar to a branch predictor, train/prefetch/validate all occur concurrently and continuously.

For example, a typical address pattern that can be captured by a prefetcher is a stride through an array, e.g.,

```
int A[M];
... // e.g., initialize A
for (i = 0 ... M) A[k*i];
```

for some constant *stride* `k`. Upon observing the core request addresses `&A[0]`, `&A[k]`, ..., `&A[k*N]`, the prefetcher predicts that the core will request `&A[k*(N+1)]`, ..., `&A[k*(N+delta)]` in the near future and proactively issues cache fills for those cache lines. Here, `N` is the *confidence threshold*: how many accesses must be seen before the prefetcher activates. `delta` is the *depth*: how far ahead the prefetcher prefetches once it activates. Depth is often directly correlated to the confidence: As more accesses are made, the prefetcher will fetch proportionally farther ahead.

Importantly, prefetchers live in the memory system and are software transparent. For example, they might live between the core and level 1 (L1) cache, or between two lower-levels (L2+) of cache. As such, they are unaware of program semantics: they only see the program address pattern and try to predict the next address. In the above code snippet, the prefetcher is unaware of the array `A`, its base or its bound. Thus, the prefetcher will prefetch data *out of bounds of* `A`, i.e., up to address `&A[M+k*delta]`, before it realizes through subsequent failed validations that the program doesn't intend to access beyond the array bounds. This will have important security implications later on.

B. Classical prefetcher security implications

Several recent papers have studied prefetchers in a security context [17, 20, 42]. We also note Gruss et al.'s [25] work on vulnerabilities in software prefetch instructions, which we

consider out of scope. At a high level, prefetcher attacks work in a similar fashion to branch predictor- and cache-based attacks [8, 35]. Specifically, when a victim program unknowingly interacts with a prefetcher, these interactions create microarchitectural persistent state changes such as in the cache or the prefetcher's internal state. An attacker (receiver) can use techniques like cache-based side-channels to measure these changes. Interestingly, prior work has shown how this can increase leakage beyond normal cache attacks. For example, many cache attacks leak the address pattern at a coarse, e.g., cache line- or page-granularity [35, 47]. The prefetcher, however, stores address pattern information at a finer, e.g., byte, granularity.

Despite the above, leakage through the prefetcher is limited to the victim's address pattern. This means that prefetcher attacks can be mitigated through constant-time programming practices that ensure that the memory address pattern is completely independent of secret data [13, 14, 16, 19, 49].

C. Data memory-dependent prefetchers (DMPs)

Beyond classical prefetchers, there is significant work in the computer architecture literature [9, 10, 18, 21, 40, 50] and several industry patents [41, 51] on what we refer to as *data memory-dependent prefetchers* (DMPs). These are designed to prefetch irregular address patterns such as pointer chases or indirections that cannot be predicted without understanding dependencies between the address pattern *and the contents of memory itself* [9, 50].

See Figure 1 (right) for an overview. Similar to a classical prefetcher, a DMP trains ❶, prefetches ❷, and validates its predictions ❸. During the train phase, the DMP monitors the data returned to the core as well as subsequent addresses and tries to determine whether the address stream is a specific function of the data returned. For example, in a pointer chase data will be directly used as an address. In the prefetch phase, the prefetcher will initiate reads to memory that follow the predicted pattern. *This, crucially, requires the prefetcher to examine and act on the contents of data memory directly.* For a pointer chase, the prefetcher will read a cache line that it believes contains a pointer *and then dereference the pointer*.

Depending on the address pattern, this can be a complex multi-interactive procedure. The common proposed patterns are shown in Figure 2, and discussed in more detail in Section IV-B. For example, the DMP in [50] patented by Intel [51] is capable of prefetching through address patterns such as shown in Figure 2d with $\mathsf{L} = 2$. This requires that the DMP not only perform multiple levels of indirection autonomously—each of which may require virtual-to-physical address translations/direct interactions with the TLB—but also infer each array's base address through relations between data and subsequent accesses. Recall, the DMP only sees data returned to the core and subsequent addresses sent by the core. Specifically, the DMP only sees physical (post-translation) addresses. In this example, the data returned to the core in the first stage of the indirection is `A[k*i]` – an offset into array `B` – and the subsequent address sent back to the memory system is `&B[A[k*i]]`. Thus, to predict the indirection into `B` for future `k*(i+delta)`, the DMP must use the data and addresses it has seen so far to infer `&B[0]`. Note, `A[k*i]` is an offset into an array in the program's *virtual* address space, whereas `&B[A[k*i]]` is likely a *physical* address. So, the DMP must autonomously perform virtual to physical address translations to identify data-address correspondences.

```
arr A;
# Fill A with pointers
for(i = 0; i < len(A); i++)
  *A[k*i];
```

(a) 1-level pointer-chasing.

```
arr A;
# Fill A with pointers
for(i = 0; i < len(A); i++)
  *(...) **A[k*i];
```

(b) L-level pointer-chasing.

```
arr A;
arr B;
# Fill A with offsets
for(i = 0; i < len(A); i++)
  B[A[k*i]];
```

(c) 1-level indirection-based.

```
arr A;
 ... # Many such arrays
arr Y;
arr Z;
# Fill arr A–Y with offsets
for(i = 0; i < len(A); i++)
  Z[Y [(...) A[k*i ]]];
```

(d) L-level indirection-based.

Fig. 2: Examples of types of DMP expected access patterns. k is the constant stride. L is the number of pointer dereferences that occur ignoring streaming over A. The left column is the explicit $\mathsf{L} = 1$ case. L is a DMP design decision. We discovered a 1-Level Pointer Chasing DMP (a) on the Apple M1.

D. DMP security implications

To our knowledge, prior to this paper, there has been no evidence to suggest that *any* DMP is implemented in *any* commercial processor. Nor (by extension) has there been analysis of the security implications of DMPs in the wild. Vicarte et al. [45] did recently perform an analysis of the security implications of proposed microarchitecture, including DMPs, but these were not known to be implemented. While their work was theoretical, it points out how DMPs have potentially disastrous security implications. For example, consider an indirection-based DMP that prefetches the pattern `C[B[A[k*i]]]` similar to the one in Figure 2d. Misused, this DMP can be coerced to leak all of program memory, similar to Spectre and Meltdown [30, 33]. To see this, suppose that the DMP was used in a sandbox setting. In that case, the attacker controls the program and can therefore easily force the DMP to activate. The attack proceeds as follows. 1) the attacker specifies a value (call it `j`) stored off the end of array `A`. `j` will correspond to the address of the value in memory the attacker wants to learn. 2) the DMP erroneously reads `j` and accesses memory to read `C[B[j]]`. Recall from Section II-A, prefetchers do not know array bounds. Thus, `B[j]` can refer to the data at any memory location. Finally, `C[B[j]]` serves as the transmitter in a memory side channel: `B[j]` is a secret and `C[B[j]]` turns that secret into an address to memory.

E. Apple Silicon

Modern Macs no longer use Intel processors, but instead use the new (ARM) M1 line. As the M1 is very similar to previous Apple processors, vulnerabilities in it may affect millions of consumers. We have confirmed that our findings apply to the A14 (iPhone 12) and the new M1 Max at a minimum.

The M1 has eight cores: four high-performance Firestorm and four energy-efficient Icestorm cores [12]. As we find the DMP to only be present on the Firestorm cores, we focus on the relevant Firestorm details. Each core has a private L1 cache, and there are two large L2 caches shared between cores of the same type. The four Firestorm cores share a 12 MiB L2, and the Icestorm cores share 4 MiB. Each Firestorm core has a private 192 KiB L1 instruction cache and 128 KiB L1 data cache [11]. While there is no official information on the associativity or cacheline size, we found that L2 lines are 128 bytes, and L1 lines are 64 bytes. When filling L1d lines, two adjacent L1d lines are brought in at a time, and both are independently evictable. We also believe that the L1 is 8-way associative and the L2 is 16-way associative from our experience building eviction sets (Section VII-C).

A major complication for reverse engineering is reports that the M1 DRAM controller performs frequency scaling [34]. This matches our observations that a cache miss to DRAM can return in a wide range of times. We find that increasing the pressure on DRAM can reduce the average access time more than amortizing measurement costs would anticipate. The net effect is that we observe otherwise inexplicable decreases in memory access times for longer experiments.

Other relevant aspects of the M1 include that it can have an unusually large number of instructions in flight to exploit instruction level parallelism [28], and does not support any form of Simultaneous Multi-Threading (SMT).

III. Threat Model and Attacker Objectives

There are two main threat models we consider for the M1 DMP: adversarial unprivileged (or sandboxed) code, and latent gadgets in benign code. This is similar to prior microarchitectural vulnerability research that exploits unprivileged attacker code as well as cases of privileged programs containing speculative gadgets [30]. The M1 does not support any form of simultaneous multithreading and so it is not considered.

A. Sandboxed Adversarial Code

In this model, we assume a standard microarchitectural sandboxed attacker: the adversary is able to run arbitrary sandboxed code on a system that does not trust the sandboxed code. The adversary is attempting to perform memory reads outside the sandbox and will leverage microarchitectural details of the processor to achieve this. This is a scenario commonly seen with JavaScript sandboxes in browsers, the kernel sandbox for eBPF code, NaCl modules, and more.

As we assume the sandbox model, the adversary will control the *training pattern* that will eventually activate the DMP. The training pattern is the series of legal, in-sandbox memory accesses made by the adversary that will cause the DMP to activate and fetch data based on the predicted next accesses after the training pattern. The adversary leverages this behavior to cause the DMP to read outside of the sandbox, and then leak that information back to the adversary.

We additionally assume that the adversary can use standard cache side channels to retrieve information about the cache state. We demonstrate specifically using Prime+Probe on the M1 in Section VII-C, but these include Prime+Probe [35, 38], Flush+Reload [48], and other similar styles of attack. The attacker will use these techniques to receive the secrets transmitted via cache state by the DMP.

B. Latent DMP Gadgets

Like with other microarchitectural attacks, it may be the case that a victim program already contains the necessary code patterns an adversary can use to induce an adversarial training pattern. This is not unlikely, as during our reverse engineering we found it easy to *unintentionally* activate the DMP by accessing stack variables that are pointers and causing the DMP to prefetch other pointers on the stack.

In this model, we assume the adversary at most has control over a set of inputs to the program, and must leverage an existing set of memory operations. This model can facilitate an attack, e.g., if the memory operations' access pattern is a function of the attacker input. It is also possible for a program to, without any adversarial interaction, cause a DMP to activate and leak information.

One possible example of the former would be a syscall that dereferences userspace-defined pointers, such as `readv` or `writev`. In these situations, the adversary may be able to induce activation of the DMP during kernel execution and cause the DMP to leak data near the kernel buffer containing data copied from userspace.

For the latter, consider a program that accesses (unconditionally) addresses X, A, B, and C, where X is the address of an attacker controlled string buffer. If the buffer contains the values "A,B,C,Z" then it is possible the DMP will interpret X as an Array-of-Pointers currently being iterated over and dereferenced, and then (attempt) to prefetch Z.

IV. The Dangers of DMPs

As part of our study of the DMP present in Apple CPUs, we first had to consider the *possible* design dimensions of a DMP, and the relevant security impact of each. While *any* DMP will have security implications, understanding the implementation of a specific DMP is necessary for making definite claims about the vulnerability of real programs and to formulate platform-specific software defenses. For example, a DMP using a prefetch buffer (see Section IV-D) may not even provide an advantage over standard cache side channels!

A DMP performs several important actions during operation that allow for the use of side channels to determine secrets. We will use the terminology of access-transmit-receive for discussing the leakage of secrets [29]. After activating, any data read by the DMP to determine addresses for prefetching is considered *accessed*. The DMP is then considered to have *transmitted* that data when it performs a prefetch to an address which is a function of the data. Finally, the adversary uses some side-channel attack (cache occupancy, cache contention, etc.) to *receive* that data.

Below, we explore the possible design space for a DMP through a security lens. This analysis is driven by our survey of existing DMP and prefetcher literature, existing prefetcher reverse engineering, and questions that arose while working on

this paper. As real DMPs have not previously been evaluated for security impact, this is an unexplored area useful for framing both the M1's DMP as well as any future DMP analysis. Relevant axes of interest are:

- What are the preconditions for DMP activation?
- What memory is accessed to inform prefetching?
- What function of memory values is transmitted?
- How can the adversary receive the transmitted values?

A. Preconditions for a DMP to activate

Like classical prefetchers, a DMP must track memory accesses made by programs and decide when to activate. Based on previous prefetcher designs, we know that this may track only address suffixes, may organize tracking entries by PID, may organize memory accesses by the instruction address they originate from, and may rely on another non-DMP prefetcher to retrieve data. Each of these possibilities has significant impact on what an attacker can do with that DMP.

If, like the Intel L1i prefetcher [17], the DMP only tracks address suffixes, then it is vulnerable to *aliasing* attacks. This allows an adversary to train the prefetcher using non-contiguous memory accesses that only appear to be contiguous when the upper bits of the address are ignored. For example, an adversary could train a DMP over a sandboxed memory region but a safe access outside of the sandbox that aliases to the same pattern could activate prefetches outside the sandbox.

Instruction-pointer (IP) tagged pattern tracking on the other hand limits attacker capabilities by restricting the code performing dereferences to loops. Without IP tagging, the memory access pattern can originate from any series of instructions that perform memory accesses. These instructions may not even intentionally be referencing related memory, and may simply appear to the DMP to be a contiguous streaming access.

Like general DMP address tracking, IP tagging may only track address suffixes [31]. Once again, this will allow an adversary to perform aliasing attacks where two distinct instructions that share an address suffix will be conflated by the DMP as the same originating address.

Process ID (PID) tagging, like IP tagging, limits the adversary by forcing all accesses and prefetches to occur in the same process. Without PID tagging an adversary may be able to train the DMP on one process, and then allow the unrelated victim accesses to cause DMP prefetches.

Finally, it may be the case that the DMP *follows* some classical prefetcher on the system. This would mean that the DMP's top-level activation criteria and restrictions are the same as that classical prefetcher's.

B. Data access patterns for DMPs

The most important feature to the attacker is which values the DMP will access to inform prefetching.

The first concern is if the DMP is single or multi-layer (see Figure 2). A single layer DMP performs only one (effective) memory dereference per-prefetch. An N-layer DMP will perform N dereferences per-prefetch. As an example, a prefetcher that simply prefetches the memory backing all pointers in an array-of-pointers (`*arr[n+1]`) is a single-layer DMP. A prefetcher that fetches not only the data backing a pointer in memory, but also interprets that data as a pointer and dereferences again (`**arr[n+1]`) is a two-layer prefetcher.

Multi-layer DMPs are exceptionally powerful for an attacker and most other design decisions become irrelevant if the DMP is multi-layer. The reason is that the attacker can precondition the first value being accessed (e.g., `arr[n+1]`) to refer to an arbitrary memory location, meaning the DMP can subsequently access arbitrary program memory. This is well demonstrated in Vicarte et al. [45] which shows how to use the 2-level Indirect Memory Prefetcher (IMP) [50] to construct a Universal Read Gadget and transmit the contents of all of virtual memory. For the rest of this section we assume a single-layer DMP.

As with classical prefetchers, a DMP is likely capable of detecting a stride pattern where the access pattern touches non-adjacent items in memory. Stride detection will have some maximum distance within which sequential accesses are considered part of the pattern. This maximum stride will determine the maximum distance from the end of the training pattern that the DMP will prefetch from.

Any prefetcher will also have a maximum number of elements that it is willing to prefetch, generally increasing with higher confidence. This is the effective depth of the prefetcher. Fundamentally, the furthest value that can be targeted by a DMP is $(max\_stride \times depth) + end\_of\_training\_address$. We will refer to $max\_stride \times depth$ as the *maximum prefetch distance* (in bytes).

As we will see with the M1 AoP DMP, there can be other unusual restrictions on what memory the DMP can access. These don't follow any particular set of rules.

C. Function of data transmitted by a DMP

A DMP can be either a pointer-chasing prefetcher or it can be an indirection based prefetcher (see Figure 2).

A pointer-chasing DMP dereferences pointers in memory and prefetches the cache lines found there. Thus, an attacker that controls the train pattern can trick the DMP into dereferencing a secret value as if it were a pointer. These DMPs allow an adversary in control of the training pattern to cause a secret memory location to be treated as an address, and to attempt a load of that address. If the secret value is a valid pointer, this will transmit the pointer value through standard cache side-channels. If it is not an address, it may still be possible for the adversary to monitor the (failed) page-walk or use TLB side-channels [23, 43] to learn the upper bits of the secret.

Indirection DMPs dynamically determine the base address of some array(s) and prefetches portions of it based on a series of index values in memory. These are more powerful transmitters than pointer-chasing DMPs as they easily transmit values that are not valid virtual addresses. Specifically, an attacker that controls the train pattern can trick the DMP into treating a secret as an offset in a base-plus-offset calculation. Suppose the attacker additionally controls the base address (which *is* the case, if the attacker controls the training pattern). Then the indirection-based DMP avoids the previously-discussed issue in the pointer-chasing prefetcher: the attacker

can arrange for the base-plus-offset to fall within a mapped memory region that is accessible to the attacker. This allows both for simpler transmission and straight-forward cache-occupancy side-channels for reception.

D. Receiving data transmitted by a DMP

Once a DMP has accessed and transmitted a secret, the adversary must now receive that secret. In the simplest case of an indirection prefetcher this would involve checking the access time (cache occupancy status) of every entry in the base array. For pointer-chasing prefetchers this would mean running a cache contention side-channel to detect what address was brought into the cache.

A DMP may alternatively prefetch to a *prefetch buffer* rather than directly to the cache. A prefetch buffer is a small, typically fully associative, cache which only holds prefetched data. Only prefetched data can induce contention on this buffer, making it significantly harder for the adversary to observe the effect of the prefetch. Then, there must be an actual access to the address corresponding to the transmitted value to observe a timing difference or affect cache state.

Since the value being transmitted is the address in question for single-layer DMPs, it may not be possible for the adversary to directly access this address at all. This would occur because it is unlikely that a secret address is mapped into the adversary's virtual address space.

V. Existence of the M1 DMP

In this section, we provide a detailed walkthrough of our initial experiments confirming the existence of a specific DMP while ruling out the existence of several other DMPs. We also describe the steps we took to determine that the root cause of our observations was, in fact, a DMP as opposed to other microarchitectural features (like speculative execution).

We tested for the existence of four DMPs: both single- and two-level versions of pointer-chasing and indirection-based DMPs (Section IV). Our findings show the existence of a single-level pointer-chasing DMP, so our focus below is on how we setup the experiment for that variant. We discuss other variants (negative results) at the end of the section (V-F).

A. Experiment overview

To confirm the existence of the single-level pointer-chasing DMP (referred to as the DMP for short), we compare the execution time of two different methods for accessing the same randomly generated sparse series of memory addresses. The first method–the *AoP DMP* pattern–pre-computes all memory addresses and stores them sequentially in an array-of-pointers (AoP). The addresses are then accessed by streaming over the AoP and dereferencing each pointer. The second method–the *baseline*–accesses the same series of addresses by computing them on the fly. Computing the addresses ensures they cannot be prefetched by either a DMP or a classical prefetcher. Both experiments generate addresses using the same PRNG seed.

All experiments insert dependencies between operations such as loads and PRNG calls. This action prevents out-of-order execution from issuing multiple operations simultaneously and makes overall execution time strongly correlated to the average memory access time.

Finally, the cache is flushed between experiments to remove inter-experiment interference. We discuss this and other methodological details, e.g., ensuring that both experiments run the same instruction sequences, in Section V-C.

B. Setting up the sequence of memory accesses

We set up the experiment by allocating two large buffers as shown in Figure 3. One of the buffers, the *data buffer*, is filled with random data, and the other buffer, the *AoP*, is filled with unique pointers to disjoint and non-consecutive 128-byte chunks of the data buffer.

These constraints are not necessary to activate the DMP but will amplify the signal to noise ratio of DMP-caused speed ups and minimize any affects from noise or other microarchitectural optimizations. In particular, uniqueness and 128-byte aligned accesses ensure that data backed by pointers is not already cached from being accessed earlier in the experiment. Recall from Section II-E, the M1 has 128 byte (L2) cache lines. Using pointers to non-consecutive chunks improves the likelihood that a classical prefetcher (Section II-A) will not activate from data buffer accesses.

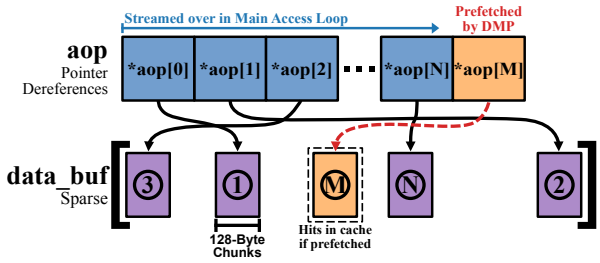

Fig. 3: Memory layout of the DMP AoP and data buffer. Black arrows illustrate memory accesses due to *aop* dereferences (Line 17) in Algorithm 1. Baseline accesses (Line 16) directly load the same entries in the *data_buf* without dereferencing from the *aop*. The AoP shown contains pointers which are consecutive in memory (unit stride). Each pointer points to disjoint and non-consecutive 128-byte chunks of the data buffer. If we access the AoP from index 1 through N and the DMP activates, the contents of the data buffer at $aop[M]$ for $M > N$ may be brought into the cache.

C. Access patterns and other considerations

After setting up the data buffer and the AoP, there are still some precautions that must be taken when accessing the pointers to ensure a speedup can only be caused by a DMP.

If we were to just measure the execution time of two different loops–one baseline access pattern loop and one AoP access pattern loop– then we would be comparing the execution time of two different instruction sequences. The AoP access pattern has an extra memory access per loop iteration since it must: 1) indirect by some offset into the AoP and 2) dereference the pointer stored in the AoP. Whereas the baseline must 1) make a single memory access and 2) use a PRNG to compute the next data buffer address on each iteration. Thus, seeing different runtimes for the two loops would not be surprising, and not necessarily be due to the presence of a DMP.

```
1  data_buf = ... /* some large buffer */
2  aop = ... /* some large buffer */

3  aop_mode = IS_AOP_RUN /* 0 - baseline, 1 - AoP */
4  aop_idx = 0
5  rand_idx = PRNG(RAND_SEED)

   /* Fill the AoP */
6  for i in 0...NUM_PTRS do
7      *aop[i] = data_buf + (rand_idx * aop_mode * CL_SIZE)
8      rand_idx = PRNG(rand_idx)
9  end

10 FLUSH_CACHE

11 MEM_BARRIER
12 dep_val = 0
13 start_time = READ_TIMER(dep_val)
14 dep_val = MSB(start_time)

   /* Training loop */
15 for _ in 0...NUM_PTRS do
       /* Baseline Access */
16     dep_val = MSB(data_buf[rand_idx * (1 - aop_mode) *
         CL_SIZE | dep_val])
       /* AoP Access */
17     dep_val = MSB(*aop[aop_idx | dep_val])

18     aop_idx = aop_idx + aop_mode
19     rand_idx = PRNG(rand_idx)
20 end

21 MEM_BARRIER
22 stop_time = READ_TIMER(dep_val)
```

Algorithm 1: Pseudocode for the baseline experiment (which computes pointers on the fly) and the experiment testing for the presence of a single-level pointer-chasing DMP and baseline. READ_TIMER calls mach_absolute_time which returns time in ticks. CL_SIZE stands for the (128 byte) cache line size. Dependencies are guaranteed to resolve to zero by using only their Most Significant Bit; denoted by MSB. MEM_BARRIER is an instruction/data serialization instruction. PRNG is a C macro that expands into a Lehmer random number generator, i.e., is not implemented as a syscall. The code is compiled with compiler optimizations turned off, and the assembly code was manually inspected to ensure intended behavior.

To address these discrepancies, we ensure that both the baseline and AoP experiments execute the same instructions, while taking care to ensure that the baseline does not activate a DMP. The code used for both experiments is shown in Algorithm 1. For the baseline, we add an access and dereference the pointer at index 0 of the AoP during each iteration (Line 16). For the AoP case, we compute the address on the fly as in the baseline (Line 19) but use the pointer read from the AoP to lookup the data buffer (Line 17). With both experiments executing the same instructions, we expect the baseline to run slightly faster than the AoP pattern due to occasional cache misses from AoP traversal.

It is also necessary during baseline runs to set all pointers in the AoP to point to the first element in the data buffer: otherwise the DMP activates during the *baseline* run due to the single AoP access combined with the computed pattern being similar to the AoP case pattern. This is an instance of the pattern described in III-B where a single read of a cacheline containing pointers is misinterpreted as the source of multiple pointer dereferences.

Finally, we add dependencies between operations to ensure that speedups are not due to out-of-order execution. Specifically, *dep_val* in Algorithm 1 ensures that all loads are executed serially and between the timer start and stop operations. Note that the first load in each iteration, which looks up *data_buf*, depends on *both* the previous load into the AoP and the PRNG computation—regardless of whether the baseline or AoP-based experiment is being run. This, coupled with the fact that both the AoP lookup and PRNG operation are expected to be relatively fast, implies that the loop's performance will be strongly correlated to the *data_buf* access latency.

D. Other Notes on Methodology

We find the DMP to be present solely on Firestorm cores. To improve consistency, we core pin our experiments by setting the thread quality of service, as described in [1]. We do not perform any kind of frequency pinning (for the Firestorm cores or for DRAM) throughout our experiments.

Our experiments make use of two different timers: the M1's performance monitoring counter (PMC) [27] and the `mach_absolute_time` macOS syscall [3]. The PMC can measure time at cycle-granularity with Apple reporting a maximum clock speed of 3.2 GHz for the Firestorm cores whereas `mach_absolute_time` can measure time at a granularity of (on average) 42 ns per 'tick'. Despite the coarser-grain measurement, we use the `mach_absolute_time` timer in many situations since we found it easier to work with (e.g., accessible from userspace, accessible across cores) and sufficient to distinguish between cache hit vs. DRAM access events. We use the PMC in select experiments to distinguish between finer-grain events (e.g., an L1 cache vs. L2 cache hit).

For all experiments with either timer, we first measure the timer's overhead by running a start timer-stop timer pair back to back in an empty loop. This overhead is subtracted from all points on graphs that use that timer. For `mach_absolute_time`, we found the overhead to be $\sim$ 42 ns and convert measurements using that timer to ns using that conversion rate.

E. Results

Figure 4 reports the time elapsed (stop_time - start_time) for the baseline and DMP variants tested in Algorithm 1, for different length sequences of pointers (AoPs). The main takeaway is that the "Array of Pointers" variant (testing the presence of a single-level pointer-chasing DMP) sees significant speedup ($3-8X$) on medium to large AoPs compared to the baseline ("Computed") access times.

We note that, while the AoP variant always sees speedup relative to the baseline, the speedup varies as a function of NUM_PTRS. To break this down, we divide Figure 4 into three regimes (1), (2) and (3) demarcated with vertical dashed lines. In regime (1)–small NUM_PTRS–the speedup is initially zero and increases quickly with NUM_PTRS. This is due to timer granularity: for small NUM_PTRS, timer overhead dominates. In regime (2)–medium NUM_PTRS–the speedup converges given sufficiently large NUM_PTRS. We attribute this to the DMP improving the average memory access time in the AoP-based experiment. Finally, in regime (3)–large NUM_PTRS–the speedup decreases with NUM_PTRS.

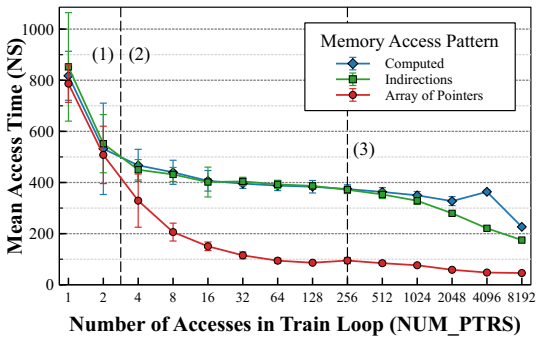

Fig. 4: Execution times for the single-level pointer-chasing DMP ("Array of Pointers"), single-level indirection-based DMP ("Indirections"; c.f. Section V-F3), and Baseline/pointers computed on the fly ("Computed") patterns, using the setup in Algorithm 1. We measure access time using on-chip timers according to Algorithm 1 (`stop_time - start_time`). Times are obtained using `mach_absolute_time` and converted to nanoseconds as described in Section V-D. For the indirection-based access pattern, the AoP dereference is replaced with an indirection. Each point represents average pointer dereference time (averaged across 2560 runs and the number of accesses in the train loop) with error bars representing standard deviation.

We attribute this to the DRAM frequency scaling discussed in Section II-E. From the core's perspective, an increase in DRAM frequency will appear as a decrease in access times.

These results provide evidence that the speedups are not the result of speculative execution. First, speculation would not cause such a large and consistent speedup for the AoP while leaving the baseline–which executes the same (serialized) instruction sequence in the same loop–unaffected.[2] Second, these speedups vanish when we run the same experiment on the M1's Icestorm cores, which still feature speculative execution but (presumably) lack other high-performance microarchitectural features such as the DMP.

The results are consistent with the behavior of a DMP (Section II-A). When iterating through smaller AoPs (regimes (1) and (2) in Figure 4), the DMP is less confident and dereferences fewer pointers. When iterating through larger AoPs (regime (2)), the DMP is more confident and aggressively dereferences more pointers resulting in larger speed ups [17, 50].

1) **Testing for prefetches+dereferences of unaccessed AoP entries:** We also tested the existence of the DMP with a second methodology. Shown in Figure 3, the idea is to have the test program stop iterating through the AoP without accessing all of the pointers and to then test whether the next *unaccessed* AoP entries have been prefetched and dereferenced. If the AoP stores M pointers, we have the test program access N pointers for $N < M$. We then perform what we call a *test access*, and measure the time to load the cache line at the address given by $aop[M]$.[3] Critically, we avoid interaction with the DMP by not accessing the *aop* during the test access. That is, by computing and accessing the address pointed to by $aop[M]$ in the same manner as in the baseline case.

[2]Subsequent results in Section VI are also inconsistent with speculative execution but consistent with prefetchers.

[3]This assumes $M \leq N + \triangle$ where $\triangle$ is the prefetcher depth (Section II-A). In this experiment, we assume that this holds and that $\triangle$ is known. Analyzing what is the depth $\triangle$ in different situations is a subject in Section II-A.

Figure 5 shows the time to perform one of these test accesses for $N = 256$ pointers and $M = 259$ as well as measured access latencies to various memory levels (L1, L2, DRAM). Lower latency test accesses indicate the DMP prefetched and dereferenced data. The figure shows that test accesses for the DMP configuration track closely with the L2 cache hit latency. From this, we conclude that the DMP prefetches into the L2 cache and is likely built alongside the L2.

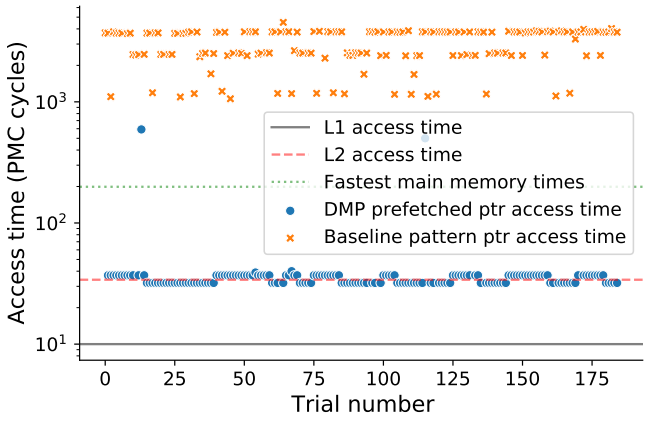

Fig. 5: Test access latency, relative to the baseline (computing pointers on the fly) and measured access latencies to different level memories. Time is measured using the fine-grain performance monitoring counter from Section V-D. The label 'Fastest main memory times' refers to the fastest main memory time we observed with DRAM frequency scaling (Section II-E).

F. Testing for the existence of other prefetchers

So far we have only discussed the existence of a single-level, pointer-chasing DMP. We also tested whether the M1 contained other classical prefetchers and other data memory-dependent prefetchers. We found the M1 *does* feature at least one other classical prefetcher but does not contain the other data memory-dependent prefetchers described in Section IV.

1) **Testing for classical (stride) prefetchers:** We confirmed through a separate analysis that the M1 contains a separate classical (stride) prefetcher that prefetches data into the L1 cache. Based on our analysis, this prefetcher seems to be completely separate from the DMP—i.e., has different depth, confidence, etc., parameters—and thus does not impact what data the DMP can access (Section IV-B). Thus, we do not study it further in this paper.

2) **Testing for multi-level pointer-chasing DMPs:** Next, we tested whether the pointer-chasing DMP we had been focusing on was multi-level. Confirming whether such a prefetcher is present is very important since having more than one level dramatically increases the scope of data that the DMP can access (Section IV-B).

For this experiment, we added another level to the AoP and reran the previous experiments. Specifically, we allocate an additional array which holds pointers to random 128-byte chunks of the original AoP. We call this additional array the *outer AoP* and the original AoP the *inner AoP*. The pointers in the outer AoP are again spaced out so that there is only one pointer at the start of every 128-byte chunk with the rest of the chunk zero-padded. We ensured that the pointers chosen

for the outer AoP would not cause the DMP to activate for the inner AoP and data buffer, which would create false positives. The access pattern and training loop is then the same as the AoP DMP (Algorithm 1), but this time we *double* dereference the pointer at the current index of the AoP.

3) **Testing for single-level indirection-based DMPs:** Next, we tested for the existence of a single-level indirection-based prefetcher such as the indirect memory prefetcher (i.e., B[A[i]] [50]). Such prefetchers also have interesting (and different) security implications due to their ease of leaking non-pointer data (Section IV-C). For this experiment, we changed the single-level AoP-style code in Algorithm 1 so that the AoP would store offsets into a second array, as opposed to direct pointers into memory.

4) **Results:** Both of the above experiments did not indicate the presence of other styles of DMP. Figure 4 "Indirections" shows the performance of the indirection-based experiment relative to the pointer-chasing variants. We did not try to equalize the instruction sequences between this variant and the pointer-chasing variants. Yet, the Indirections variant results in performance that is very similar to the pointer-chasing variant. This is expected assuming no such indirection-based DMP exists: both codes are memory bound (hence, performance is largely a function of the average memory access time) and exhibit the same memory system performance.

Since the single-level indirection-based prefetcher experiment returned a negative result, we did not directly test the existence of a multi-level indirection-based prefetcher.

5) **Other microarchitectures:** We also ran existence tests for indirection-based DMPs (Section V-F3) and the single-level, pointer-chasing DMP (Algorithm 1) on more than 5 Intel and 3 AMD processor families, more than 50 machines in total. In none of these systems did we observe test pointers being prefetched.

VI. Reverse Engineering the M1 DMP

After confirming that there is a DMP on the M1, we now turn to reverse engineering the parameters of the M1 DMP so that we can exploit it. Recall from Section IV that we need to answer the following questions to understand how to exploit or mitigate a DMP:

- What are the preconditions for DMP activation?
- What memory regions can a DMP access?
- What function of memory values is transmitted?
- How can the adversary receive the transmitted values?

These questions are discussed below and summarized in Table II.

A. What are the preconditions for activating the M1 DMP?

We identified four conditions necessary to activate the M1's DMP.

1) **Fire vs. Icestorm cores:** Any thread interacting with the DMP must be running on a Firestorm core (Section II-E). Developers and MacOS users can specify whether a program should run on an Icestorm or Firestorm core by modifying a process's quality of service (QoS) bits. For example, setting a process's QoS to 'user interactive' will cause the process to be scheduled only on Firestorm cores [12, 26].

2) **DMP "noise" tolerance:** There are restrictions on operations between sequential accesses of the AoP. We examined four types of noise: serialization, system calls, other spurious operations, and time. We found that system calls and serialization such as instruction and data synchronization barriers[4] placed between accesses to pointers in the AoP prevents the DMP from activating. On the other hand, we found that the DMP is generally tolerant of time-based delays between memory accesses and the insertion of unrelated arithmetic (e.g., incrementing a counter variable from 1 to 1000) or memory operations between accesses.

3) **DMP minimum confidence threshold:** For the DMP to activate, the program needs to dereference at least 3 pointers in the AoP. That is, the DMP has a minimum *confidence* threshold of 3 accesses to activate at all (Section II-A). We determined this threshold by running the Section V-E1 experiment with various lengths of training loop (N). We then find the smallest N at which we observe cache hits on dereferencing the (un-touched) test pointers. On the M1, this occurs after 3 AoP pattern accesses. This is consistent with Figure 4, which shows the AoP and baseline patterns diverging for train lengths between 2 and 4.

4) **AoP alignment requirements:** The M1 DMP will not build confidence if the addresses of the pointers in the AoP are not aligned to eight-byte boundaries. This is easily observed by offsetting the start of the AoP by any amount that is not a multiple of 8, and running any of the previous experiments. This is slightly disappointing for attackers, as it precludes a sliding-window style attack where the attacker learns a secret byte-by-byte through repeated experiments with differently-aligned AoPs.[5]

We also make several observations that weaken assumptions needed to activate the DMP.

5) **The DMP is not IP indexed:** We observed that the M1 DMP is not IP indexed. That is, the instructions that cause the memory accesses matching the AoP pattern need not be related in any way. We tested this by unrolling the training loop from Algorithm 1 into straight-line memory accesses without branches and observing that both of the experiments from Section V still show the DMP activating.

6) **The DMP can be activated using only speculative accesses:** We found that the DMP can be activated using only speculative memory accesses that are eventually squashed. We demonstrate this by adapting a Spectre attack example [5] to run on the M1, and using it to activate the DMP on branch mispredictions (Algorithm 6). Instead of tricking the branch predictor into reading out of bounds, the experiment will read the first *three* elements in the DMP AoP pattern when it mispredicts and speculatively executes. Since DMP activation is slightly noisy for an AoP with 3 pointers (Figure 2), the experiment performs this branch-predictor training and misprediction loop many times (lines 7-16) to raise confidence in whether *target_ptr* was dereferenced or not. If the DMP

[4] https://developer.arm.com/documentation/100941/0100/Barriers.

[5] Recall from Section IV-C, the attacker may be able to learn high-order bits of a secret (even if it is not a virtual address) by monitoring TLB and related MMU state. A sliding-window attack can amplify this leakage by tricking the DMP into interpreting different secret bytes as high-order address bits.

can be triggered via speculation only, then the target pointer should be dereferenced and it will be a cache hit which is tested for on line 17.

```
1  aop[0*128] = p1
2  aop[1*128] = p2
3  aop[2*128] = p3
4  aop[3*128] = target_ptr
5  aop[4*128] = target_ptr
6  FLUSH_CACHE
7  for train_iter in 1...30 do
8      idx1 = 0
       /* Branchless if-then-elses */
9      idx2 = if (train_iter%6) then 0 else 1
10     idx3 = if (train_iter%6) then 0 else 2
11     if idx3 == access_evicted_memory_containing(0) then
12         *aop[idx1*128]
13         *aop[idx2*128]
14         *aop[idx3*128]
15     end
16 end
17 result = was_l2_timing(target_ptr)
```

Algorithm 2: Pseudocode of experiment for determining whether the DMP will activate when all memory accesses are speculative and eventually squashed. This code will also be used in our ASLR break (Section VII-D). p1, p2, p3 are unique random pointers to a data buffer. The cache line storing the value 0 used in the conditional check is evicted on each iteration.

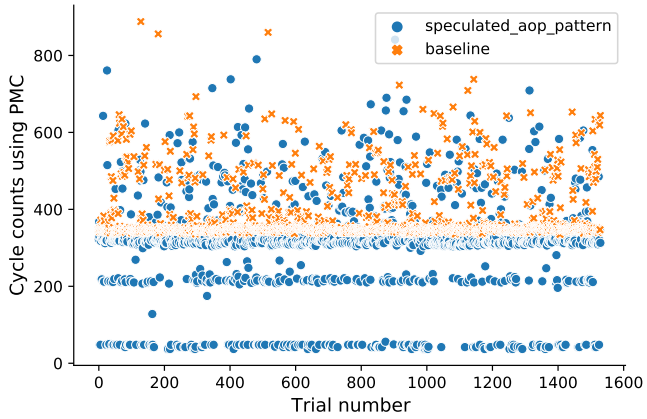

Fig. 6: Speculative accesses in an AoP pattern causes the DMP to activate.

Figure 6 shows the dereference times of the target_ptr (line 17) across 1530 experiment runs. The baseline shows dereference times for the target pointer without the three speculative accesses on each 6th train loop iteration–i.e., it never mispredicts.

B. What memory regions can the M1 DMP access?

There are two main considerations for whether or not the M1 DMP can leak a secret at a given address: 1) how far the secret is located from an adversary-interactable AoP and 2) whether the DMP is willing to prefetch memory located at any reachable address.

Recall from Section IV-B that the location of the furthest pointer past the end of an AoP that a single-level DMP can dereference is $(max\_stride \times depth) + end\_of\_training\_address$ where the maximum prefetch distance is $max\_stride \times depth$ in bytes. In the case of the M1, we found that the DMP will also activate when traversing an AoP backwards.

1) **Determining maximum prefetch distance (as a function of confidence, stride and depth):** We found there to be a non-trivial relationship between DMP confidence (the number of training accesses touching the AoP) and stride (distance between pointers) in determining the M1 DMP's maximum prefetch depth. This is shown in Figure 7 and Table I. The high-order bit is that a stride of 64 cache lines (8 KiB) enables the DMP to reach (access and dereference) a pointer 64 KiB (i.e., 8 pointers deep) away from either end of the AoP.

To start, Figure 7 shows that the number of entries the DMP is willing to prefetch and dereference (depth) is clearly proportional to the number of accesses the program makes that match the DMP's target pattern (confidence). This is consistent with expected DMP behavior (Section V-E). Note, consistent with Section VI-A3, the AoP DMP plot only shows low latency test accesses for train sizes 4 and larger. We note two other features in the data. First, when more than 2048 accesses are performed, either the first eight or sixteen accesses are not prefetched. This behavior is caused by a 16 KiB page boundary, and is further studied in Section VI-B2. Second, for both the AoP and baseline patterns, as the number of accesses increases, the access time for misses decreases. We propose an explanation for this in Section V-E.

Table I reports the maximum distance in bytes we can prefetch/dereference a pointer. We run this experiment with a training loop that is large enough to maximize confidence (and therefore depth, see previous paragraph). Our experiments show that the maximum distance is not monotonically increasing with stride, but larger strides do tend to enable larger maximum prefetch distances, as expected. To summarize, up to a stride of 8 KiB, the maximum distance increases (up to a maximum distance of 64 KiB). We did not see the DMP activate for strides larger than 8 KiB (1024 pointers).

Finally, we observed that the DMP does not activate when the stride, at cache line granularity, is not a power of two. This is shown in Figure 8. We confirmed that low measurements on the y-axis (faster times) occur iff test accesses are dereferenced, i.e., indicate that the DMP activated.

Stride (B)	Maximum Distance from AoP (B)	Stride (B)	Maximum Distance from AoP (B)
8	384	512	4096
16	384	1024	8192
32	256	2048	16384
64	1536	4096	32768
128	1024	8192	65536
256	2048		

TABLE I: The maximum distance ahead in memory prefetched is a function of stride. All experiments are performed using 4144 training accesses. This number of accesses achieves the maximum prefetch depth (Figure 7) while avoiding the page boundary interaction described in Section VI-B. All memory in the AoP in between touched pointers (for a given stride) is zeroed.

2) **Unprefetchable virtual address regions:** Unexpectedly, we found that the M1 DMP behavior is affected by the virtual address of the AoP itself and the virtual addresses of pointers contained in the AoP. We found that the M1

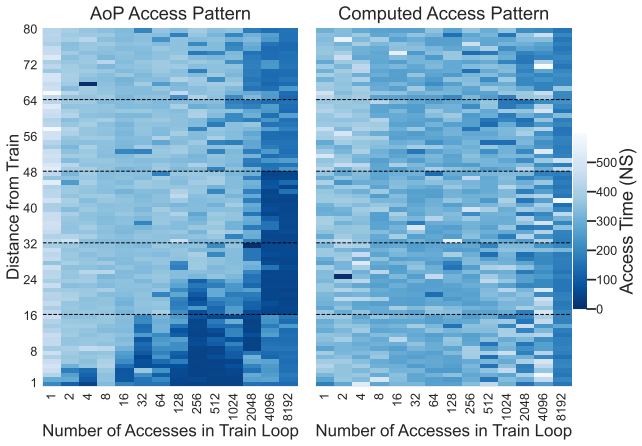

Fig. 7: DMP prefetch depth is a function of train loop size (NUM_PTRS in Algorithm 1). The left/right graphs show results for the single-level AoP/baseline experiments in Algorithm 1. Each column (along the x-axis) represents a single training loop size. Each row in a given column (along the y-axis) corresponds to a test access latency into *data_buf* (Section V-E1) that far away from the last pointer touched in the training loop. Times are measured using `mach_absolute_time` and converted to nanoseconds as described in Section V-D.

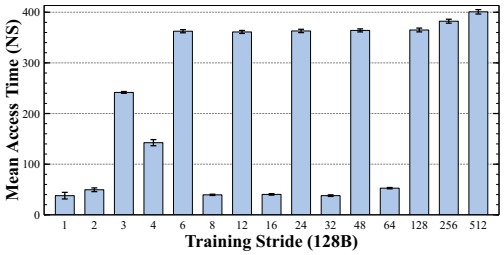

Fig. 8: DMP training loop performance as a function of stride. All experiments use a large training loop size, similar to Table I. All memory in the AoP in between touched pointers (for a given stride) is zeroed. Times are measured using `mach_absolute_time` and converted to nanoseconds as described in Section V-D.

DMP does not dereference pointers located in the cacheline immediately after 16 KiB or 2 MiB virtual address boundaries of the AoP and has additional odd behavior depending on which boundary (16 KiB or 2 MiB) we try to get it to cross.

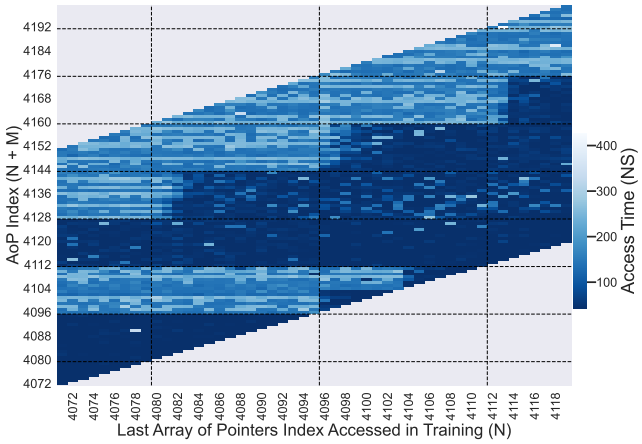

Fig. 9: For different training AoP lengths N (x-axis), what is the access latency for values in *data_buf* corresponding to pointers in the AoP at offsets $N+M$ (y-axis). The dashed lines indicate cacheline boundaries *in the AoP*. A 16 KiB Boundary occurs at *aop*[4096]. Times are measured using `mach_absolute_time` and converted to nanoseconds (Section V-D.)

We illustrate this behavior with a dense AoP in Figure 9. We see that the DMP does not dereference pointers located in the 128 bytes immediately after the 16 KiB page boundary (y-axis value 4096), but it does dereference the pointers before and after this 'dead zone'. However, if the last training access of the AoP occurs after the 16 KiB page boundary (x-axis values 4096-4103, then the DMP only refuses to dereference the pointers in the second 64 byte chunk after the 16 KiB boundary until train accesses touch this 64 byte chunk (x-axis value 4104). Notably, this behavior is independent of AoP sparsity. Though a sparse AoP will naturally have fewer pointers within these 128 byte regions.

We observed a more aggressive behavior at 2 MiB address boundaries of the AoP. This behavior is, once again, independent of AoP sparsity. The DMP will not dereference *any* pointers past a 2 MiB boundary of the AoP. When a training loop crosses this boundary, the DMP's confidence resets and must retrain based on pointers accessed after the boundary.

These interactions are difficult to explain because: 1) the L1 stride prefetcher (Section V-F1) prefetches AoP lines across the 16 KiB boundary, and 2) the DMP induces page walks that populate the TLB entries for both the AoP and the pointers in the AoP (Section VI-C). We observed similar results running this experiment on Asahi Linux.This means that the reluctance of the DMP to cross these boundaries is not due to any safeguards against address translation.

We additionally observed that on MacOS, the DMP will not dereference pointers contained in the AoP which have a virtual address between `0x280000000` and `0x7fe840004000`–the end of mappable user-space addresses. On Asahi Linux, which allows us to map virtual addresses above `0x7fe840004000`, we found the DMP to activate and dereference pointers above `0xffff3f2b0000`, but then addresses below `0x280000000` are not dereferenceable. At this time, we do not have an explanation for why there are restrictions on the virtual addresses the M1 DMP will dereference.

C. What function of memory values is transmitted?

The M1 AoP DMP makes prefetches based on memory content as if it were pointer values. This, naively, places a major restriction on the function of values transmitted. Only the top 57 bits of the address/value (i.e., L2 cacheline granularity) is transmitted, and only if they are a valid virtual address. As pointers must be placed at 8-byte alignments (Section VI-A), we cannot read partial values. (Section VI-A4).

We did, however, find that the M1 DMP fills entries in the TLB for the pointers in the AoP. To test this we set up an AoP in the usual way but made each 'test pointer' after the end of the AoP a pointer to a unique page. We can then use the `mprotect` system call to change the protection bits of pages associated with the test pointers to invalidate their TLB entries. We now set up two experiments: 1) where we `mprotect` the test pointers before streaming through the AoP and 2) where we `mprotect` the test pointers after streaming through the AoP.[6] If the DMP fills TLB entries, then the test pointer access

[6]We also pad the `mprotect` syscalls with 10,000 cycle delays on both sides of the call and again add data dependencies.

DMP Activation Requirements	The DMP can
Access at least 3 pointers	Activate solely through speculative `aop` accesses
Retrain at each `aop` 2 MiB page boundary	Recognize an unrolled `aop` (it is not IP-indexed)
`aop` must be pointer aligned	Recognize strides through the `aop` in powers of two
Run on firestorm	Prefetch in either direction

Leakage Target Must
Be a pointer
Be within current prefetch distance
Not be `0x280000000-0xffff3f2b0000`
Not be in first 128 bytes of any `aop` 16 KiB page boundary

TABLE II: Summary of the M1 DMP

times for experiment 1 should be faster than experiment 2, and if it does not fill TLB entries, then the access times should be the same. Across 250 runs of each experiment, we found that the average test pointer access time and standard deviation for experiment 1 was 27.95 cycles and 2.8 cycles respectively (using the PMC), and the times for experiment 2 were 110.83 cycles and 49.87 cycles respectively. From this, we conclude that the DMP does fill TLB entries which transmits the page bits of the value through another channel.

D. How can an adversary receive the transmitted values?

We found three attacker visible ways that the M1 DMP affects microarchitectural state: it prefetches to the L2 cache, it fills TLB entries, and it has internal state (e.g., confidence).

To receive changes to L2 cache state, the adversary may use directly timed accesses to the cache, use a cache side channel like Prime+Probe [35, 38], or alternatively use an interconnect side channel [36]. The TLB entries may also be attacked directly using a TLB side channel [23, 43].

So far, the retrieval methods have had secrets or pointers of interest *after* the end of the AoP, but attackers can also learn about pointers *contained* in the AoP by using the DMP's confidence metrics as an indicator. We showed earlier in Figure 7 that higher confidences result in deeper prefetching, and frome Section VI-A3 that the DMP needs 3 AoP patterned accesses to then prefetch the "4th" (next) pointer. We can use these effects to determine the validity of pointers under the right circumstances. We later show in Section VII-D how one can use the DMP's confidence to break ASLR.

VII. Examples of Augury Techniques

In this section, we cover four scenarios where the M1 DMP can be used in attacks: performing out-of-bounds reads, beating speculative load hardening, retrieving leaked addresses via Prime+Probe, and breaking ASLR.

A. Out-of-Bounds Reads

Algorithm 3 shows a proof-of-concept (PoC) which uses the DMP to read past the end of a buffer. We start by picking three random pointers (*test_p_1*, *test_p_2*, and *test_p_3*) that point to different cachelines of memory. Although in this example these pointers are accessible to the attacker, we know that the DMP alters the L2 cache and TLB, so an attacker could instead conduct an attack like Prime+Probe if they did not have access to these pointers [35, 38]. The user then picks one of the pointers on line 2, and we will use the DMP to determine which pointer the user picked without reading it.

```
   /* Stick the user chosen pointer after the filled AoP */
1  aop[0 : AOP_SIZE − 1] = ... /* Random, unique ptrs */
2  test_p = user_choice(test_p_1,test_p_2,test_p_3)
3  thrash_cache()/* Evict test pointers */

   /* Train the DMP by streaming through the AoP */
4  *aop[0]
5  ...
6  *aop[AOP_SIZE − 1]

   /* Find the fastest test pointer access time */
7  time(*test_p_1)
8  time(*test_p_2)
9  time(*test_p_3)
```

Algorithm 3: PoC using straight-line memory accesses to activate the DMP and distinguish between three pointers.

To activate the DMP, we first create an AoP filled with AOP_SIZE random pointers to disjoint 128-byte chunks of memory. This AoP is placed immediately before the memory location containing the user-chosen pointer. AOP_SIZE must be at least 3 to activate the DMP, with larger sizes increasing the DMP's confidence and the clarity of the signal. Next, we flush the entire cache state by reading in several MB of unrelated data on line 3. We do this to ensure that the only test pointer that has a cache hit will be the pointer off the end of the AoP, assuming it is prefetched. We now stream through the AoP, accessing and dereferencing each pointer in it, and not the test pointer outside the AoP (see lines 4-6.) We have unrolled the loop for two reasons: it makes it clear that this is not a speculative execution effect, and it also demonstrates that attackers only need to induce an access pattern that looks like the cache misses caused by streaming through an AoP.

After training the DMP, we measure the access time to each of the 3 test pointers. Since only the the user selected pointer should be prefetched into L2, it will be the fastest to access

We ran this PoC 500 times using an AoP of size 64, selecting a different test pointer number each time, and measuring its accuracy in distinguishing pointers. For the first 250 runs, we used the M1's PMC (see Section V-D) which can very accurately distinguish between L2 and main memory access times. For the latter 250 runs, we used `mach_absolute_time` which is noisier than the PMC. The with the PMC the PoC had a 92.0% average accuracy–i.e., number of times the PoC correctly picked the pointer. With `mach_absolute_time`, the PoC had a 70.2% average accuracy.

B. Beating Speculative Load Hardening

Speculative load hardening (SLH) is a defense against conditional branch-based speculative execution attacks, known by the name of Spectre Variant #1 [15, 30]. Some pseudocode with and without SLH applied is shown in Algorithm 4.

SLH prevents Spectre Variant #1 by adding a branchless recheck of each branch condition within each conditional's

```
1  N = NUM_TRAIN_PTRS
2  stride = LINE_SIZE

   /* AoP train loop without SLH */
3  for i in 0...N do
4  |  *aop[i * stride]
5  end

   /* AoP train loop with SLH */
6  mask = 0
7  for i in 0...N do
   |  /* branchless set */
8  |  mask = (i >= N) ? 0 : ALL_ONES_BITMASK
9  |  *aop[(i & mask) * stride]
10 end
```

Algorithm 4: Example of gcc and clang's SLH hardened AoP iteration loop [6, 7, 15]. We apply SLH by providing clang with the `-mspeculative-load-hardening` flag [6, 7]. This option for AArch64 masks only the loaded value, and this is the only SLH option for AArch64 when using LLVM 14 (the current latest version). Note that there are additional speculative-execution-specific instructions inserted in the final compiler output.

body to apply an all-ones or all-zeroes bitmask to a data load. This results in the load working as expected when the branch predictor is correct and only loading from offset 0 when the branch predictor guesses incorrectly. In the case of the SLH train loop in Algorithm 4, the hardening of loaded values also applies to the index into the array of pointers; this should prevent the accesses from speculatively reading past the bounds of the array of pointers.

Since the DMP only ever sees cache misses, the (non-speculative) access pattern caused by both loops in Algorithm 4 will be the same. SLH provides no protection against using the DMP to bypass the bounds check. We reran our existence experiments from Section V with the SLH compiler flag enabled and confirmed that they still work. We also reran the PoC from Algorithm 3 getting an accuracy of 88.0% with the AoP accesses turned back into a loop, SLH enabled, and using the PMC. Indeed, the exploits should still work since to the memory system, the memory accesses caused by both loops in Algorithm 4 will be the same, and the additional instructions inserted from SLH do not prevent DMP activation.

While it is unsurprising that SLH does not protect against DMP leakage, it is important to note that some code vulnerable to Spectre V1, but protected by SLH, will continue to be vulnerable to the same receive side-channel as before. As such, a developer applying SLH to attacker submitted code or code containing latent DMP gadgets (such Algorithm 4) gains almost no defensive advantage. However, unlike the Spectre attacks that SLH was designed to prevent, the M1 DMP has a maximum stride and depth which constrains the furthest value past the end of a buffer that can be prefetched.

C. Retrieving leaked pointers via Prime+Probe

The previous example primitives rely on the adversary being able to directly time accesses to the targeted pointers to determine their cache state. This is often not possible, and we can instead use cache side-channels like Prime+Probe to determine if a given pointer was prefetched.

We set up this experiment identically to the basic out-of-bounds read in Section VII-A. However, we use only two test pointers ($test\_p\_0$ and $test\_p\_1$) and build eviction sets of size 24 for each (ev_0 and ev_1) using the baseline algorithm from Vila et al [46]. Each run of the experiment randomly chooses either $test\_p\_0$ or $test\_p\_1$ as the test pointer.

After the training accesses are complete, we time an access to each eviction set (ev_0 and ev_1) independently. The eviction set with a longer access time corresponds to the pointer we guess as the test pointer. In general this manifested as one eviction set taking around 100 PMC cycles longer to access than the other. Across 4300 runs, this resulted in a correct guess in 60.0% of runs. However, if we remove runs where the Probe step failed, and did not result in ev_0 or ev_1 being significantly slower, the accuracy rises to 84.8%. The net effect is that Prime+Probe, while effective, adds another layer of noise to the recovery of pointer values.

D. Breaking ASLR by testing virtual addresses

Address space layout randomization (ASLR) is a widely deployed defense that prevents attackers from knowing a priori where important parts of a program live in memory. It does this by randomizing the memory locations of portions of a program such as the stack, heap, code, and libraries.

Breaking ASLR (that is, discovering the virtual addresses of code and data pages) is a core step in a larger exploit. We show how the DMP can be used to check whether arbitrary pointers are valid mapped virtual memory addresses and thus aid in breaking ASLR. Using the DMP rather than a cache side-channel removes the need for knowledge of the cache system, or creating eviction sets, and is significantly less noisy.

We set up an experiment similar to Algorithm 2, with the third pointer ($p3$) replaced with the address we wish to test validity of. Since the DMP requires 3 accesses (see Section VI-A) that match the AoP pattern to activate, we can use the DMP's confidence threshold as a metric for the validity of $p3$. Since the test address may not be readable and reading it would cause a segfault, ensure that all three training accesses are only speculative, and eventually squashed. Since Section VI-A showed that the DMP can be activated in these conditions, the fourth pointer ($p4$, the *target pointer*) will be prefetched if and only if the $p3$ (the test address) was valid.

Using our experiment code from Algorithm 2 written in C (and using `mach_absolute_time`), we can test a virtual address for validity on average every 24.91 ms with standard deviation 0.79 ms. This long duration is due to an unoptimized implementation that uses cache thrashing rather than targeted eviction sets. The attacker can repeat this per-pointer validity test to sweep across the address space, trying each virtual memory page and determining which are mapped. This will, at the least, reveal the location and size of memory regions that are mapped for use by the program.

VIII. Mitigating the threat of DMPs

Unfortunately, the AoP DMP is already widely deployed on at least the A14 and M1 family of processors. This DMP, to our knowledge, cannot be disabled via software updates. Given

that our experiments show the DMP is not present on Icestorm cores (See Section V-E), the only dependable mitigation is to execute sensitive software on the Icestorm cores at a significant performance cost. For sensitive software running on Firestorm cores, our remaining option is to modify the software to best-effort avoid DMP-caused data leakage.

A. Removing secrets

If we assume a sandboxed threat model, our most straight forward solution is similar to the one adopted by most Spectre defenses: do not keep secret data in the same virtual address space as the adversary sandbox or user-space program. This is only applicable to cases where secret data and attacker code are co-located, and is not relevant to other situations.

Since Spectre vulnerabilities have put all of a process' virtual memory space at risk of being leaked, we have seen widescale deployment of policies like Chrome's Site Isolation [39]. These policies segment untrusted code (like sandboxed JavaScript) from sensitive data (such as the rendering data from another web origin) by placing them into entirely separate virtual address spaces. Similarly, KPTI/KAISER [24] removed virtual address mappings for the kernel from user-space processes. The net effect of these changes was the removal of valuable targets from the virtual address space of highly attacker-influenceable code. Thankfully, these partitioning efforts have removed most of the obvious sandbox or userspace to kernel attack surfaces for the M1 DMP.

B. Preventing M1 DMP interaction

For both the sandbox and latent gadget cases, we can use any features or implementation quirks that cause the DMP to ignore values or never activate. We consider this as preventing the DMP from ever accessing and transmitting a secret bit.

In Section VI-B2 we found that the M1's DMP is unwilling to prefetch pointers to specific virtual address regions. As the DMP will skip pointers that are in this address range even after it begins fetching nearby pointers, we can put all data in this region and prevent pointers to it from leaking. We caution that there is no known explanation for why this region exists, and leveraging it should not be considered a complete mitigation.

We also found that the DMP requires pointers to be aligned on 8-byte boundaries. If all pointers in the program are non-8-byte aligned, the prefetcher cannot to prefetch them.

C. Protecting non-pointer values from the M1 DMP

Both of the above approaches assume that the DMP leaks only *pointer* values. We believe that this is not a fundamental limitation of the M1 DMP, and that by observing changes to the cache caused by page walks and the TLB an adversary may be able to receive information about a failed (invalid pointer) prefetch. If this is the case, we must consider any page walk that varies based on secret bits to be leaking information [23]. One possible defense would be to only store secret data in the bottom N-bits of every 64-bit aligned chunk, and ensure that the top N-bits are never a valid virtual address prefix. Any attempted prefetch of a 64-bit chunk containing secrets would then fail before the pagewalk encountered secret related bits.

D. General DMP mitigations

The only generalized, but incomplete, mitigation to *all* DMPs is to remove secrets from the virtual address spaces accessible to adversaries, similar to many Spectre mitigations. Unfortunately there is no guarantee that all DMP implementations will happen to reach a subset of the memory reachable by Spectre. As we outlined in Section IV there are many possible design possibilities like aliasing or cross-PID training that would reach beyond what a Spectre attack can.

Orthogonal to removal of secrets, we should also consider cases where a privileged non-malicious program contains latent DMP gadgets that must be detected and removed. In our experiments we repeatedly *unintentionally* activated the AoP DMP by storing pointers on the stack. With a DMP this aggressive, it is possible for a program to be accidentally leaking secret values without any intervention by an adversary.

IX. Conclusions

Exotic microarchitectural optimizations that leak data never accessed by the core have arrived in mainstream processors and are unlikely to disappear any time soon. The M1 has been rightfully lauded for performance and efficiency, and the recent M1 Pro and Max continue to drive excitement for novel microarchitectural approaches. While exceptional now, we expect that this AoP DMP is only the first of many DMPs to be deployed across all architectures and manufacturers.

Here, we've demonstrated that, while difficult to wield, the M1's DMP is capable of being abused by an adversary. It can read and transmit some types of memory values outside of sandboxes or test the validity of pointers controlled by an attacker. This is despite a single-level pointer-chasing DMP being nearly the worst-case DMP for an attacker, leaking only pointers and only under restricted situations. Thankfully, many particularly worrying scenarios like JavaScript sandboxes already assume that an adversary can leak any value in the virtual address space. These systems are unlikely to have significant security impacts from the M1 DMP. However, given the ease with which the DMP can be activated, it is likely that existing programs and kernels contain latent DMP gadgets that can be leveraged to leak data in their own address spaces.

As with timing attacks, Spectre attacks, and others, we emphasize the need for compiler and program transformation tools to adapt to mitigate data at rest leakage. The M1 DMP is an opportunity to prepare our defensive software techniques for the next generation of microarchitectural attacks.

X. Acknowledgments

We thank our anonymous reviewers for their valuable feedback to this paper. We thank Andrei Frumusanu for their exceptionally insightful remark "[...] we might believe they're using some sort of pointer-chase prefetching mechanism."[22]. We thank Dean Tullsen for seeding this idea. This work was funded partially by NSF grants 1954521 and 1942888, as well as by an Intel RARE grant.

REFERENCES

[1] Energy Efficiency Guide for Mac Apps: Prioritize Work at the Task Level. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx/PrioritizeWorkAtTheTaskLevel.html.

[2] Intel x86_64 and ia32 developers manual. https://software.intel.com/en-us/articles/intel-sdm.

[3] Mach absolute timer. https://developer.apple.com/documentation/kernel/1462446-mach_absolute_time.

[4] Software optimization guide for amd epyc 7003 processors. https://www.amd.com/system/files/TechDocs/56665.zip.

[5] Spectre attack example. https://github.com/Eugnis/spectre-attack.

[6] Spectre V1 defense in GCC. https://lwn.net/Articles/759423/.

[7] Speculative load hardening. https://llvm.org/docs/SpeculativeLoadHardening.html.

[8] Onur Aciicmez, Jean-Pierre Seifert, and Cetin Kaya Koc. Predicting secret keys via branch prediction. *IACR*, 2006.

[9] Sam Ainsworth and Timothy M. Jones. Graph prefetching using data structure knowledge. *ICS*, 2016.

[10] Sam Ainsworth and Timothy M. Jones. An event-triggered programmable prefetcher for irregular workloads. *ASPLOS*, 2018.

[11] Apple. Apple event - november 10, 2020. https://www.apple.com/apple-events/, 2020.

[12] Apple. Optimize for apple silicon with performance and efficiency cores. https://developer.apple.com/news/?id=vk3m204o, 2020.

[13] Daniel J. Bernstein. The Poly1305-AES Message-Authentication Code. *FSE*, 2005.

[14] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. *PKC*, 2006.

[15] Chandler Carruth. Speculative load hardening. https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0/edit#.

[16] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. Fact: A flexible, constant-time programming language. *SecDev*, 2017.

[17] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. Leaking control flow information via the hardware prefetcher. *arXiv*, 2021.

[18] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. *SIGOPS Oper. Syst. Rev.*, 2002.

[19] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. *S&P*, 2009.

[20] Patrick Cronin and Chengmo Yang. A fetching tale: Covert communication with the hardware prefetcher. *HOST*, 2019.

[21] Babak Falsafi and Thomas F. Wenisch. A primer on hardware prefetching. *Synth. Lect. Comput. Archit.*, 2014.

[22] Andrei Frumusanu. Apple announces the Apple Silicon M1: Ditching x86 - What to Expect, Based on A14, Nov 2020.

[23] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. *Sec*, 2018.

[24] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. *ESSOS*, 2017.

[25] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. *CCS*, 2016.

[26] hoakley. Cores shouldn't all be the same: M1 Macs do better, May 2021.

[27] Dougall Johnson. Apple CPU. https://github.com/dougallj/applecpu, 2021.

[28] Dougall Johnson. Apple M1 Microarchitecture Research. https://dougallj.github.io/applecpu/firestorm.html, 2022.

[29] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. *MICRO*, 2018.

[30] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *S&P*, 2019.

[31] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *Sec*, 2017.

[32] Kevin M. Lepak and Mikko H. Lipasti. On the value locality of store instructions. *ISCA*, 2000.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. *Sec*, 2018.

[34] Hector Martin. M1 dram scaling observed. https://twitter.com/marcan42/status/1450364369519276032, 2021.

[35] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *CT-RSA'06*, 2006.

[36] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. *Sec*, 2021.

[37] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. *PACT*, 2012.

[38] Colin Percival. Cache missing for fun and profit. *Proc. of BSDCan*, 2005.

[39] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: Process separation for web sites within the browser. *Sec*, 2019.

[40] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. *SIGOPS Oper. Syst. Rev.*, 1998.

[41] Sreenivas Subramoneyand Stanislav Shwartsmanand Anant Noriand Shankar Balachandranand Elad Shtiegmannand Vineeth Mekkatand Manjunath Shevgoor and Sourabh Alurkar. System, method, and apparatus for enhanced pointer identification and prefetching, August 2021.

[42] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. *CCS*, 2018.

[43] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering. *Sec*, 2022.

[44] Po-An Tsai, Andres Sanchez, Christopher W. Fletcher, and Daniel Sanchez. Safecracker: Leaking Secrets through Compressed Caches. *ASPLOS*, 2020.

[45] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data. *ISCA*, 2021.

[46] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. *S&P*, 2019.

[47] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. *S&P*, 2015.

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

[49] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. *NDSS*, 2019.

[50] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. Imp: Indirect memory prefetcher. *MICRO*, 2015.

[51] Xiangyao Yu, Christopher J. Hughes, and Nadathur Rajagopalan Satish. Hardware prefetcher for indirect access patterns, February 2017.