GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers

Boru Chen, *UIUC*
Yingchen Wang, *UT Austin*
Pradyumna Shome, *Georgia Tech*
Christopher W. Fletcher, *UC Berkeley*
David Kohlbrenner, *University of Washington*
Riccardo Paccagnella, *Carnegie Mellon University*
Daniel Genkin, *Georgia Tech*

Abstract

Microarchitectural side-channel attacks have shaken the foundations of modern processor design. The cornerstone defense against these attacks has been to ensure that security-critical programs do not use secret-dependent data as addresses. Put simply: do not pass secrets as addresses to, e.g., data memory instructions. Yet, the discovery of data memory-dependent prefetchers (DMPs)—which turn program data into addresses directly from within the memory system—calls into question whether this approach will continue to remain secure.

This paper shows that the security threat from DMPs is significantly worse than previously thought and demonstrates the first end-to-end attacks on security-critical software using the Apple m-series DMP. Undergirding our attacks is a new understanding of how DMPs behave which shows, among other things, that the Apple DMP will activate on behalf of any victim program and attempt to "leak" *any* cached data that resembles a pointer. From this understanding, we design a new type of chosen-input attack that uses the DMP to perform end-to-end key extraction on popular constant-time implementations of classical (OpenSSL Diffie-Hellman Key Exchange, Go RSA decryption) and post-quantum cryptography (CRYSTALS-Kyber and CRYSTALS-Dilithium).

1 Introduction

For over a decade, modern processors have faced a myriad of microarchitectural side-channel attacks, e.g., through the caches [64, 92], TLBs [43, 79, 83], branch predictors [6, 36], on-chip interconnects [32, 65, 86], memory management units [44, 51, 82], speculative execution [52, 55], voltage-frequency scaling [78, 88, 89] and more.

The most prominent class of these attacks occurs when the program's memory access pattern becomes dependent on secret data. For example, cache and TLB side-channel attacks arise when the program's data memory access pattern becomes secret dependent. Other attacks, e.g., those monitoring on-chip interconnects, can be viewed similarly with respect to the program's instruction memory access pattern. This has led to the development of a wide range of defenses—including the ubiquitous constant-time programming model [53, 62], information flow-based tracking [42, 80, 95], and more—all of which seek to prevent secret data from being used as an address to memory/control-flow instructions.

Recently, however, Augury [84] demonstrated that Apple M-series CPUs undermine this programming model by introducing a Data Memory-dependent Prefetcher (DMP) that will attempt to prefetch addresses found in the contents of program memory. Thus, in theory, Apple's DMP leaks memory contents via cache side channels, even if that memory is never passed as an address to a memory/control-flow instruction.

Despite the Apple DMP's novel leakage capabilities, its restrictive behavior has prevented it from being used in attacks. In particular, Augury reported that the DMP only activates in the presence of a rather idiosyncratic program memory access pattern (where the program streams through an array of pointers and architecturally dereferences those pointers). This access pattern is not typically found in security critical software such as side-channel hardened constant-time code—hence making that code impervious to leakage through the DMP. With the DMP's full security implications unclear, in this paper we address the following questions:

Do DMPs create a critical security threat to high-value software? Can attacks use DMPs to bypass side-channel countermeasures such as constant-time programming?

1.1 Our Contribution

This paper answers the above questions in the affirmative, showing how Apple's DMP implementation poses severe risks to the constant-time coding paradigm. In particular, we demonstrate end-to-end key extraction attacks against four state-of-the-art cryptographic implementations, all deploying constant-time programming.

Analyzing DMP Activation Patterns. We start by re-examining the findings in Augury [84], here we find that Augury's analysis of the DMP activation model was overly restrictive and missed several DMP activation scenarios. Through new reverse engineering, we find that the DMP activates on behalf of potentially *any* program, and attempts to dereference *any* data brought into cache that resembles a pointer. This behavior places a significant amount of program data at risk, and eliminates the restrictions reported by prior work. Finally, going beyond Apple we confirm the existence of a similar DMP on Intel's latest 13th generation (Raptor Lake) architecture with more restrictive activation criteria.

Breaking Constant Time Cryptography. Next, we show how to exploit the DMP to break security-critical software. We demonstrate the widespread presence of code vulnerable to DMP-aided attacks in state-of-the-art constant-time cryptographic software, spanning classical to post-quantum key exchange and signing algorithms. Our key insight is that while the DMP only dereferences pointers, an attacker can craft program inputs so that when those inputs *mix* with cryptographic secrets, the resulting intermediate state can be engineered to look like a pointer if and only if the secret satisfies an attacker-chosen predicate. For example, imagine that a program has secret s, takes x as input and computes and then stores $y = s \oplus x$ to its program memory. The attacker can craft different x and infer partial (or even complete) information about s by observing whether the DMP is able to dereference y. We first use this observation to break the guarantees of a standard constant-time swap primitive [54] recommended for use in cryptographic implementations. We then show how to break complete cryptographic implementations designed to be secure against chosen-input attacks.

Summary of Contribution. We contribute the following.

1. **Reverse Engineering Apple and Intel DMPs.** We reverse engineer the DMP found on Apple CPUs and discover new activation criteria (Section 4).
2. **Developing DMP Exploitation Techniques.** Using our new understanding of the DMP, we develop a new type of victim-agnostic chosen-input attack and associated attack primitives (e.g., eviction set construction) that does not require the attacker and victim to share memory. We use these primitives to mount a proof-of-concept attack on constant-time swap operations (Section 5).
3. **Breaking Constant-Time Cryptography.** Undergirded by our chosen-input attack framework, in Sections 6 and 7 we develop end-to-end key-extraction attacks on constant-time implementations of classical cryptography (OpenSSL Diffie-Hellman Key Exchange and Go RSA decryption) and post-quantum cryptography (CRYSTALS-Kyber and CRYSTALS-Dilithium).

1.2 Disclosure

We disclosed to Apple, OpenSSL, Go Crypto, and the CRYSTALS team. Apple is investigating our PoC. OpenSSL reported that local side-channel attacks (i.e., ones where an attacker process runs on the same machine) fall outside of their threat model. The Go Crypto team considers this attack to be low severity. The CRYSTALS team agreed that pinning to the Icestorm cores without DMP could be the short-term solution and hardware fixes are needed in the long term.

2 Background

Cache Architecture. Modern processors use a hierarchy of caches to reduce memory access latency. Typically, higher-level caches are smaller and faster to access, while lower-level caches are larger but slower to access. For example, the Apple processors we study in this paper have two cache levels, a core-private L1 and a shared L2. These caches are set-associative, meaning that they contain a fixed number of cache sets, each of which can fit a fixed number of cache lines. Cache lines are the basic unit for cache transactions. Multi-level caches have an inclusion policy that determines how the presence of a cache line in one level affects its presence in other levels. Most of our experiments were conducted on the Apple M1's 4 Firestorm (performance) cores, which are the only ones to have a DMP. Each Firestorm core has a 128 KByte, 8 way set-associative L1 data cache with 64 Byte cache lines and these 4 Firestorm cores share a 12 MByte, 12 way set-associative L2 data cache with 128 Byte cache lines. The shared L2 cache is inclusive of the L1 caches, i.e. every cache line present in the L1 is also present in the L2 [94].

Cache Side-Channel Attacks. In a cache side-channel attack, an attacker infers a victim program's secret by observing the side effects of the victim program's secret-dependent accesses to the processor cache. These attacks typically consist of three steps, during which the attacker (i) brings the cache into a known state, (ii) lets the victim execute, and (iii) checks the state of the cache to learn information about the victim's execution during step (ii). Two techniques commonly used to mount cache side-channel attacks are Flush+Reload [92] and Prime+Probe [64]. In Flush+Reload, an attacker that shares memory with a victim flushes individual shared cache lines and later reloads them to figure out if the victim accessed them. In Prime+Probe, the attacker builds an *eviction set* of addresses that map to the same cache set as the victim's target cache line, primes the cache set with the eviction set, and later probes it to figure out whether the victim accessed the target line / displaced a line in the eviction set.

Classical Prefetchers. Prefetchers are a hardware optimization used to hide memory access latency. Prefetchers live in the memory system, typically between the L1 and L2 or between the L2 and DRAM, and work by pre-loading data into the cache before it is requested by the core. In particular, given a program memory access pattern, *classical prefetchers* try to predict the next addresses the program will access based on its access pattern (an address trace) thus far.

Classical Prefetcher Security Implications. Several prior works have analyzed the security implications of classical prefetchers [17, 26, 27, 31, 76, 91, 98]. These works demonstrate that, through unintended interactions with prefetchers, victim programs can create cache state changes that can be measured by the attacker to leak information. Fortunately, leakage through these attacks is limited to the victim's access pattern and can be mitigated through constant-time programming practices that ensure the program memory access pattern does not depend on secrets.

Data Memory-Dependent Prefetchers (DMPs). DMPs are a class of prefetchers designed to prefetch irregular memory access patterns. In contrast to classical prefetchers, which only take the memory access pattern as an input, DMPs also take into account the *contents* of data memory directly to determine what to prefetch. The computer architecture literature and industry patents proposed several types of DMPs [7, 8, 16, 24, 29, 50, 84, 96, 97], which differ in the irregular access patterns that they are designed to speed up (e.g., linked-list traversals, sparse matrix traversals).

DMP Security Implications. Vicarte et al. were the first to perform an analysis of the security implications of DMPs [72]. In the worst case, they found that proposed (but not known to be implemented) indirect memory prefetchers could be used to build universal read gadgets that leak a program's entire memory, similar to Spectre [52, 60]. More recently, Augury demonstrated that modern Apple processors employ a type of DMP referred to as an Array-of-Pointers (AoP) DMP [84]. We describe this DMP's behavior in more detail in Section 4.1.

3 Threat Model and Setup

In this paper we assume a typical microarchitectural attack scenario, where the victim and attacker have two different processes co-located on the same machine.

Software. For our cryptographic attacks, we assume the attacker runs unprivileged code and is able to interact with the victim via nominal software interfaces, triggering it to perform private key operations. Next, we assume that the victim is constant-time software that does not exhibit any (known) microarchitectural side-channel leakage. Finally, we assume that the attacker and the victim do not share memory, but that the attacker can monitor any microarchitectural side channels available to it, e.g., cache latency. As we test unprivileged code, we only consider memory addresses commonly allocated to userspace (EL0) programs by macOS.

Hardware. Unless otherwise specified, we focus on Apple hardware. The M1-based experiments of Section 4 are run on a Mac Mini with an Apple M1 running macOS 13.5. For our investigation into the M2/M3 microarchitecture, we used a Mac Mini with an Apple M2 (running macOS 14.2.1) and a MacBook Pro with an Apple M3 (running macOS 14.2). Finally, when investigating Intel's DMP implementation, we used an Intel Core i9-13900K (Raptor Lake) CPU, running Ubuntu 23.04 with kernel version 6.2.0.

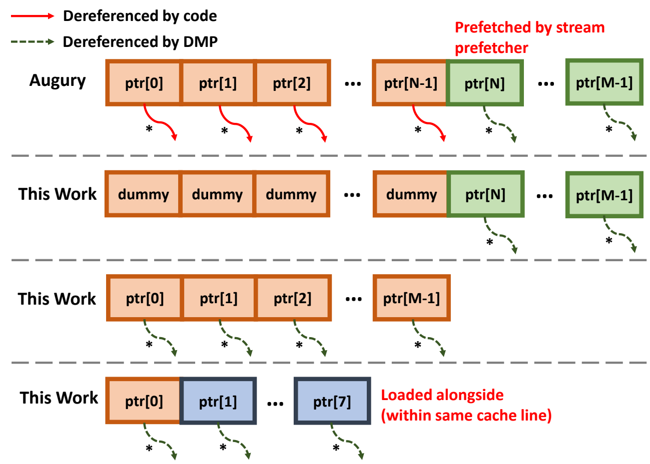


Figure 1: We compare memory access patterns and subsequent prefetches. The first row represents the activation pattern reported by Augury [84]: a streaming dereference access pattern causes the DMP to dereference out-of-bounds pointers. In the second row, we show that architectural/program-level dereferences are unnecessary; we see DMP activations even when the training array contains non-pointer values. In the third row, we show that the DMP even dereferences the in-bounds pointers that are architecturally accessed (but, again, not dereferenced). Finally, the last row shows that a single access to a memory location results in all pointers stored in the incident cache line being dereferenced.

4 Microarchitectural Characterization

4.1 Revisiting DMP Data Access Patterns

In this section, we investigate the access patterns required to activate the M1 DMP. We show that the M1 DMP dereferences more pointers and with fewer program assumptions than was claimed by Augury [84]. Figure 1 summarizes the subsection's findings.

Augury. We begin by reviewing the M1 DMP activation pattern and methodology described in Augury. Augury's code, summarized in Listing 1 (left), first allocates an array (`aop`) of length M and fills `aop` with pointers to memory addresses that correspond to unique L2 cache lines. Next, it evicts these cache lines from the L2 via cache thrashing (by loading an array eight times the size of the cache). The code then accesses (loads) and dereferences the first N elements of the `aop`, where $N \leq M$. We call `aop[0]`, ..., `aop[N-1]` the in-bounds pointers and `aop[N]`, ..., `aop[M-1]` the out-of-bounds pointers.

Augury inferred the DMP's activity by adding code after the loop to time how long it would take to dereference pointers in the `aop`. We call these test accesses. The main finding was that the latency of test accesses for out-of-bounds pointers in some index range $[N, N+\delta)$ corresponded to L2 cache hits. This is noteworthy because the code itself never dereferenced pointers located after `aop[N]`. Augury attributed this behavior to a new form of prefetcher, with prefetch distance δ.

```
uint64_t* aop[M];
// Fill aop with pointers
// to unique addresses
// or random values

for (i=0; i<N; i++) {
    *aop[i%N];
}

// Measure latency to
// set of test addresses
```

```
uint64_t* aop[M];
// Fill aop with pointers
// to unique addresses
// or random values

for (i=0; i<N; i++) {
    aop[i%N];
}

// Measure latency to
// set of test addresses
```

Listing 1: Left: The DMP activation code pattern studied by Augury [84]. Right: The DMP activation pattern studied in this work. For both, assume $N \leq M$. Both code patterns fill the aop before the loop begins and use a mod operation to inhibit speculative execution.

Observing DMP Activations. We reproduce Augury's experiments by setting $N = 256$ and $M = 264$, choosing a set of test pointers, and then either filling the out-of-bounds region with those pointers or random values. When the pointers are present, a test access (dereference) to one takes ~ 250 cycles,[1] as shown in Figure 2a. When the pointers are not present, the same test accesses take significantly longer. A cutoff of 300 cycles (red dash line) cleanly differentiates between the two cases and thus DMP activations. This corresponds to the L2 hit time and matches Augury's findings, consistent with $\delta \geq 8$.

Avoiding Architectural Pointer Dereferencing. To determine if the architectural pointer dereferences are required to trigger DMP activations we use the code in Listing 1 (right), where the in-bounds region does not contain pointers nor does the aop traversal loop perform any pointer dereferences. Again, we either fill the out-of-bounds region with test pointers or random values. See Figure 1 (second row).

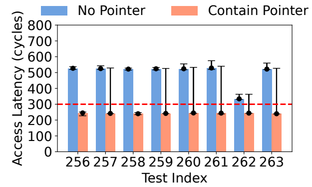
As seen in Figure 2b, when the out-of-bounds region contains pointers, test accesses are < 300 cycles despite no architectural dereferences occurring to the in-bounds pointers. From this, we deduce that architectural dereferences are not required for the DMP to activate, i.e., that the DMP will prefetch out-of-bounds pointers without them.

In-bounds DMP Dereferencing. We then further check if the in-bounds pointers are *also* dereferenced by the DMP as they are no longer architecturally dereferenced in Listing 1 (right). This is the memory access pattern outlined in Figure 1 (third row), where we iterate over an array containing valid pointers without performing any dereferences.

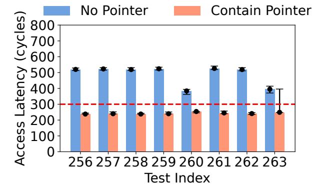
Figure 2c shows that for $N = 8$, we can still consistently differentiate between the two cases. This indicates that if the aop contains data which can be interpreted as valid pointers, merely iterating over it is sufficient to activate the DMP.

One Load, Single Pointer. Finally, we consider how general the memory access pattern can be by performing a *single* data

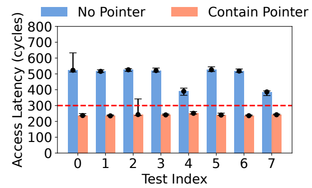
[1] We collect timing measurements by configuring and reading performance counters (PMC2-PMC7) for cycle counting via kperf.



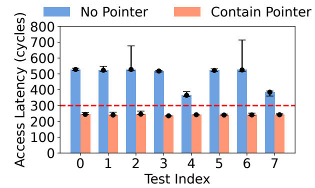
(a) Row 1: Traversing the AoP with dereferences; out-of-bounds pointers are prefetched



(b) Row 2: Traversing the AoP without dereferences; out-of-bounds pointers are prefetched



(c) Row 3: Traversing the AoP without dereferences; in-bounds pointers are prefetched



(d) Row 4: One load to AoP; pointers within the incident cache line are prefetched

Figure 2: Median, minimum, and maximum test access latencies (over 32 samples for each bar) using the access patterns of Figure 1. The x axis corresponds to the test access latency for the pointer at the corresponding index in the aop in case it contains the pointer. Blue bars (No Pointer) are for when the test pointer is not in the aop array, while red bars (Contain Pointer) are for when the pointer is in the array.

load and no architectural dereference, as shown in Figure 1 (fourth row). Even though the program only loads one aop index, other pointers in the same cache line are also brought into the cache. Figure 2d shows that with a single load,[2] we observe similar results to traversing the entire ($N = 8$) aop in Figure 2c. We further repeat the experiment but vary the number of pointers in the cache line from 1 to 8. In all cases, we observe DMP activations/dereferences for all pointers in aop, indicating that even a single pointer can trigger the DMP.

4.2 DMP Activation Criteria

Having established what memory access patterns activate the DMP, this section investigates *where data must reside in the memory hierarchy* to be DMP-searched for pointers. We show that the DMP dereferences pointers specifically on L1 cache fills and features two mechanisms to prevent redundant prefetches: a history filter and a do-not-scan hint. In this section, we make use of standard eviction sets, i.e., eviction sets for individual cache sets. We generate these eviction sets using standard techniques from prior work [85].[3]

History Filter. We start by rerunning the experiments from

[2] Replacing the load with the store instruction, we find that none of pointers in the accessed cache line are dereferenced.

[3] This is in contrast with Augury, which, as we mentioned in Section 4.1, relied on cache thrashing to precondition the cache.

Section 4.1 using standard L2 eviction sets to evict both the `aop` array and the L2 cache lines that are pointed to by pointers in the `aop` array. We call these L2 lines the *target lines*.

We observe that the DMP only reliably dereferences each pointer once, on the first access to its `aop` entry. That is, even if the previously prefetched target line is evicted from the cache, along with its `aop` entry, the DMP no longer activates when seeing that pointer in the future. This observation suggests that the decision to dereference a pointer is made based on not only the program's access pattern but also some additional mechanism. An Apple patent suggests that this mechanism might be a history filter that "attempts to identify whether a given memory pointer candidate likely corresponds to a candidate that has been recently prefetched, in which case the given candidate may be discarded as a likely duplicate" [47]. The same patent suggests that this filter may be organized as a direct-mapped 128-entry or 256-entry structure.

History Filter Reverse Engineering. To corroborate the history filter hypothesis, we design a new experiment where `aop` only contains a single pointer `ptr`. First, we access `aop`, causing the DMP to dereference `ptr`. We then evict `aop` and the target line for `ptr` from the cache using standard eviction sets. Next, we read S unique pointers stored in a different array, causing the DMP to inspect and dereference S additional pointers. Finally, we re-access `aop` and check if this second access causes the DMP to dereference `ptr`. We run the experiment 100 times for each value of S and report the success rate (i.e., the percentage of times that the DMP activated on the second `aop` access) in Figure 3.

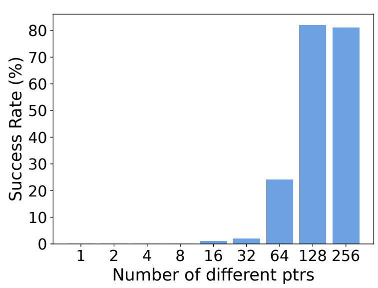


Figure 3: The percentage of experiments where the DMP re-activates when `ptr` is re-accessed (Success Rate, y-axis), as a function of the number of unique pointers accessed in between the first and second access to `ptr` (x-axis). Observe that Success Rate increases with the number of unique intermediate pointer accesses.

We observe that the DMP only reliably re-activates on `ptr` when $S \geq 128$. This behavior is likely due to the limited capacity of the history filter. That is, accessing S unique pointers results in the record of `ptr`'s target getting evicted from the filter when $S \geq 128$. We hypothesize that Augury's methodology was not affected by the history filter because its aggressive cache thrashing technique (i.e., accessing an array eight times the size of the cache) had a side effect of also flushing the history filter.

We further find that the history filter is a per-core structure and is reset if a core remains idle for an extended period of time. Specifically, the DMP reliably re-activates even when $S = 0$ if we (i) reschedule our experiment to a different core between the first and the second `aop` access or (ii) run the experiment on one core but leave the core idle for 100μs or more between the first and the second `aop` access.

L1 and L2 Cache Fills. The above observations indicate that the DMP activates when an `aop` entry is accessed from DRAM and the record of its target is not present in the history filter. Next, we investigate at which stage of a DRAM fetch the DMP scans the data for pointers. Recall that the M1 has an L2 line size of 128 Bytes and an L1 line size of 64 Bytes. With each pointer containing 8 Bytes, L2 lines can thus be split into "lower" and "upper" halves, each of which is an independent L1 line that can store $64/8 = 8$ pointers. When a program accesses either the lower or upper half, the accessed L1 line will be filled into both the L1 and L2 caches, while the other half will only be filled into the L2 cache.[4] In order to differentiate between L1 and L2 fills, we populate a L2 line size-aligned `aop` with 16 unique pointers and run the experiment from Listing 1 (right) in Section 4.1 with $N = 1$ and $M = 16$. Before each repetition, we use cache thrashing (as in Section 4.1) to evict the `aop` and its target lines from both the cache and the history filter.

Figure 4 (top) summarizes our findings, repeating each experiment 100 times and using the 300 cycle threshold from Section 4.1 for L2 cache hits. Here, we observe that when the program accesses `aop[0]`, the DMP only dereferences `aop[0]`, $\cdots$, `aop[7]`. We run 7 more variants of this experiment, varying the single `aop[i]` access from $i = 1, \cdots, 7$ and observe the same behavior for each choice of i. Next, we run 8 more variants of the same experiment, this time making a single access to `aop[i]` for $i = 8, \cdots, 15$. In this case, we observe that `aop[8]`, $\cdots$, `aop[15]` are all dereferenced for each choice of i, as shown in Figure 4 (bottom). We conclude that when filling an L2 cache line from DRAM, the DMP dereferences all pointers in the specific L1 line that is accessed, and not those in the other half of the L2 line.

We run 8 more variants of the above experiment. For these, before making an access to `aop[i]` for $i = 0, \cdots, 7$, we first make an access to `aop[8]`. We then repeat this setup while exploring the opposite case: before making an access to `aop[i]` for $i = 8, \cdots, 15$, we first make an access to `aop[0]`. As discussed above, the first access brings `aop[i]` from DRAM to the L2 cache and `aop[i]` further moves to the L1 cache with the second access. We observe that the DMP reliably dereferences the contents of the L1 line containing `aop[i]`. This means that L2 to L1 fills can also activate the DMP.

Do-not-scan Hint. The above experiments suggest that

[4] We empirically verify this by subsequently timing an access to the other half and observing that its access latency corresponds to the that of an L2 hit.

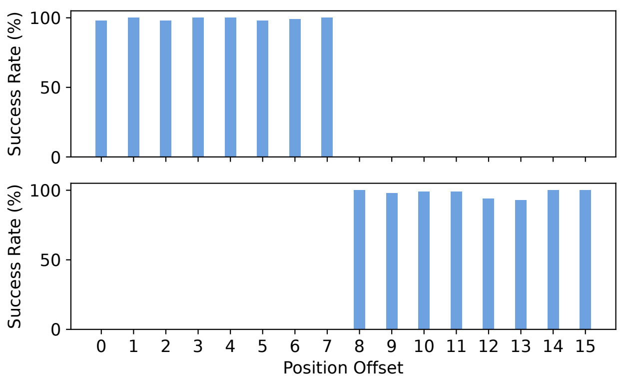


Figure 4: Which pointers in an L2 line are dereferenced when an access is made to data in that line? Top: the code accesses `aop[0]`. Bottom: the code accesses `aop[8]`. We conclude that the DMP dereferences pointers in the specific L1 line (either the upper or lower half of the L2 line) the code accessed.

the DMP searches for pointers in L1 cache lines during L1 fills, regardless of whether the L1 line is fetched from DRAM or the L2. To corroborate this hypothesis, we design another variant of the single-pointer experiment from Section 4.1. The experiment starts by loading the `aop` into the L1 and subsequently using eviction sets to either (i) evict the `aop` from the L1 or (ii) evict the `aop` from both the L1 and L2. In both cases, the experiment also evicts the target line from the cache and accesses a separate set of 256 pointers to evict the record of the target line from the history filter. Finally, the experiment re-accesses `aop` and tests if this second `aop` access causes the DMP to re-dereference `ptr`.

Interestingly, we observe that the DMP does not re-dereference `ptr` when the experiment re-accesses `aop` and `aop` was only evicted from the L1. However, when the `aop` is also evicted from the L2, the DMP re-dereferences it. This means that even if the previously prefetched target line is evicted from both the cache *and* the history filter, the DMP does not dereference that pointer again unless its `aop` entry is also evicted from both the L1 and L2. This behavior matches a mechanism also described in the previously referenced Apple patent [47], where the L2 sets a "do-not-scan" hint on L1 cache fills to prevent a previously scanned L1 cache line from being redundantly re-scanned. Fortunately, in our experiments, evicting the `aop` from both the L1 and the L2 is sufficient to clear the "do-not-scan" hint on the `aop`.

4.3 Restrictions on Dereferenced Pointers

In the previous section, we learned that the DMP activates on L1 fills and dereferences the pointers inside it if and only if those pointers' targets are not in the history filter and the filled line is not marked with the "do-not-scan" hint. We now investigate *what* pointers can be dereferenced by the DMP. For this, we again use Listing 1 (right) with $N = 1$ and $M = 1$ and rely on cache thrashing to ensure that the `aop` is uncached. We then try testing different pointer values in the `aop`, and checking for DMP activations.

4GByte Prefetch Region. We begin by investigating if the DMP requires there to be a relationship between the address of the `aop` entry and the value of the `aop` entry (i.e., the pointer). We call the address of the `aop` entry the entry's/pointer's position. To understand what the requirements are for one pointer to be dereferenced, we carry out a series of experiments that vary a pointer's position and value. See one such experiment in Figure 5 which shows that the pointer's position and value must be related for DMP activation to occur. Overall, we discover that the DMP only dereferences a pointer if the `aop` entry and target line are in the same 4 GByte-aligned region (Figure 6). In other words, that the upper 32 bits of their addresses match. Apple's patent [47] mentions similar pointer detection heuristic.

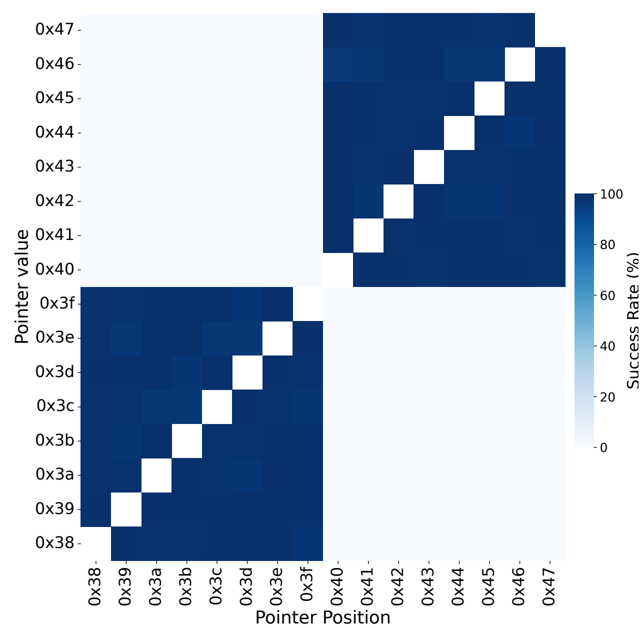


Figure 5: For various combinations of pointer position and value, when does the DMP dereference the pointer? Here, we sweep within the region between `0x380000000` and `0x480000000`. The white diagonal shows the degenerate case when the pointer's value equals its position, which is invalid. The lower 28 bits of the addresses are omitted for brevity.

Top Byte Ignore. The address space standards in ARMv8 direct the processor to ignore the top byte of the virtual address [2]. To learn whether the DMP follows this specification, we perform a series of experiments flipping different upper bits in a valid pointer. We then perform a test access to check whether, after these bit flips, the DMP still dereferences the original pointer. Figure 7 shows the results. We perform 16 experiments, where each flips a bit in the address starting at bit 48 and ending at bit 63. We observe that the DMP does not

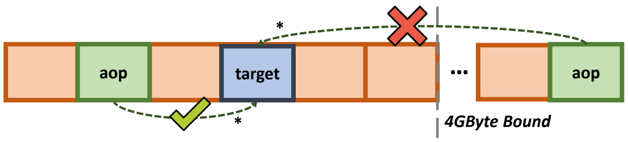


Figure 6: Outline of the placement of the target line and the `aop` entry. Following the observation from Figure 5, if the `aop` entry and target line straddle a 4GByte boundary, the DMP won't dereference the pointer.

dereference the original pointer if a bit in the range $[48, 55]$ is flipped. However, if a bit in the range $[56, 63]$ is flipped, the original pointer gets dereferenced. We conclude that the DMP ignores the upper 8 bits of a pointer when dereferencing it, which matches the "Top-Byte-Ignore" in ARMv8.

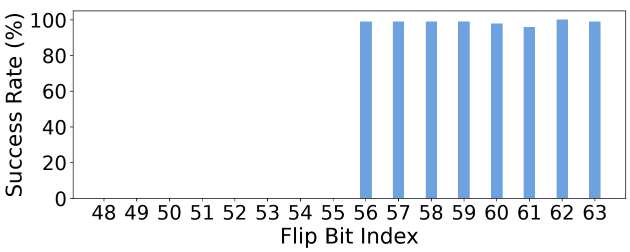


Figure 7: Activation success rate for a pointer when it is accessed by the program, after having one bit flipped between bit 48 to 63.

Auxiliary next-line prefetch. Finally, we investigate the amount of data prefetched when the DMP dereferences a pointer. We test this by performing a test access to not only a pointer's target line, but also to nearby lines. Apart from the target line, we also observe L2 hits to cache lines immediately next to the target line. We hypothesize that this is due to a next-line prefetcher being triggered alongside the DMP, which matches the adjacent-line prefetch behavior described in Apple's patent [47].

4.4 A Model for the DMP's Behavior

We now summarize the previous two subsections and make several new observations.

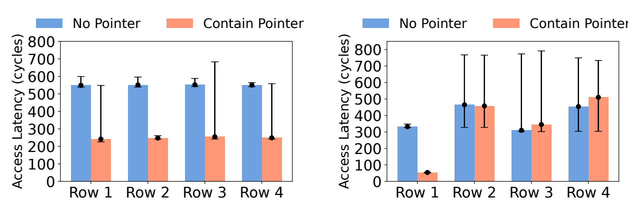
Step 1: Observing Cache Line Data. The DMP scans the data in an L1 line when that line is filled to the L1, if the line is not marked with the "do-not-scan" hint (i.e., the line has not been scanned since it was brought into the cache; Section 4.2). The DMP performs the scan by checking each pointer size-aligned chunk (the first 64 bits, second 64 bits, etc.) in the cache line.[5]

Step 2: Address Check. Next, the DMP applies additional checks and filters to each chunk (candidate pointer) to see if it should be dereferenced. Bits [63:56] are ignored (Section 4.3). Further, per Section 4.3, the cache line that stores the pointer (its position) must be in the same 4 GByte ($\log_2$ 4 GByte $=$ 32 bits)-aligned region as the cache line that the pointer points to (its target). In other words, the DMP checks whether bits [55:32] of the candidate pointer match the corresponding bits of the address of the target cache line. Finally, the DMP checks if the candidate pointer is present in the history filter (Section 4.2). If bits [55:32] match and the pointer is not in the history filter, the DMP attempts to prefetch two L2 lines. Specifically, it first prefetches the cache line targeted by the 64-bit chunk, ignoring the top byte value. Next, it triggers the CPU's next line prefetcher and fetches the neighboring cache line also into the CPU's L2 cache (Section 4.3). Both prefetched addresses are then inserted to the history filter.

As part of the prefetching process, the DMP looks up the translation lookaside buffer (TLB) and triggers page table walks to obtain the physical address corresponding to each candidate pointer (which is a virtual address [33]). On a TLB miss, the DMP inserts the missing translations into the TLB.[6]

4.5 Other Microarchitectures

We investigated the DMP behavior on other microarchitectures including the Apple M2/M3 and Intel's 13th Generation (Raptor Lake) CPUs, and display results in Figures 8a and 8b. As the Apple M3 behaves similarly to the M2, we omit its figure. In these two figures, the x-axis refers to the four access patterns shown as the rows in Figure 1, while the y-axis is the access latency for test accesses. For simplicity, we only show latencies for test accesses to the first pointer in each pattern. The Intel i9-13900K (Raptor Lake) shows a distinguishable timing difference only for the first access pattern from Figure 1, whereas the M2/M3 activates on all the patterns discussed previously. We conclude that while DMPs are present on Raptor Lake machines, they require different activation patterns. Finally, we leave the systematic investigation and exploration of Intel's DMPs to future work.



(a) Apple M2 (b) Intel Raptor Lake

Figure 8: We test four access patterns shown in Figure 1 on Apple M2 (left) and Intel 13th generation Raptor Lake (right).

[5]Pointers in `aop` should be 64-bit aligned, which is also discussed in [84].

[6]Prior work [84] also observes that the M1 DMP fills TLB entries for pointers in the `aop`.

5 Attacking Constant-Time Conditional Swap

To mitigate microarchitectural side channels, cryptographic code follows the constant-time programming principle: A secret should not determine which instructions to execute, which memory to access, or be used as input for variable-time instructions [11, 13, 14, 21–23, 62].

We now show how the DMP can break cryptographic security even when code is written to follow the constant-time principle. To introduce ideas and attacker tools, this section showcases a Proof-of-Concept (PoC) attack on a core constant-time cryptographic primitive [54] called `ct-swap` which conditionally swaps the contents of two arrays `a` and `b` based on a secret bit `secret`. We start with `ct-swap` to simplify the presentation. Later sections will reuse the ideas and processes described here to break real cryptographic code.

Constant Time Swap Overview. Listing 2 swaps the contents of array `a` and `b` based on the value of `secret` in a constant-time manner. The underlying swap operation for each 64-bit entry is borrowed from OpenSSL.[7] To achieve constant-time behavior, Line 4 in Listing 2 first extends `secret` to be a machine-sized word; i.e., `0x0000000000000000` or `0xFFFFFFFFFFFFFFFF` based on the value of `secret`. Next, for each loop iteration, Line 6 of Listing 2 computes a masked `delta` between the contents of the current elements of `a` and `b`. Finally, Lines 7 and 8 actually conditionally swap the contents of the two elements, based on the value of `secret`.

```
1  void ct-swap(uint64_t secret, uint64_t *a, uint64_t *b,
2          size_t len) {
3      uint64_t delta;
4      uint64_t mask = ~(secret-1);
5      for (size_t i = 0; i < len; i++) {
6          delta = (a[i] ^ b[i]) & mask;
7          a[i] = a[i] ^ delta;
8          b[i] = b[i] ^ delta;
9      }
10 }
```

Listing 2: Code snippet of constant-time swap. The contents of `a` and `b` is conditionally swapped based on `secret`.

5.1 Attack Overview and Challenges

Emulating realistic attack scenarios, we assume that `ct-swap` runs in a victim process, separate from the attacker's address space. We assume a simple but common protocol between victim and attacker, where the victim takes input from the attacker to populate the `ct-swap`'s `a` and `b` arrays and then executes `ct-swap`. The outcome of the swap is never directly revealed, nor is the value of `secret`. The attacker can learn page offsets (not randomized by ASLR) of array `a` and `b` by investigating the victim's program in advance. The attacker process' goal is to extract the value of `secret` from the victim, using microarchitectural side channels and the DMP.

Chosen-Input Attack. We now overview how the attacker uses the DMP to extract `secret`. At a high level, the attacker populates one of `ct-swap`'s arrays (`a` or `b`—let us assume it chooses `b`) with a pointer `ptr` of its choosing, and then arranges for the DMP to dereference the contents of the other array (`a`) during the conditional swap computation. Then, the attacker uses conventional cache side-channel analysis to observe whether `ptr` was dereferenced by the DMP due to `ct-swap`'s computation over `a`, which in turn reveals whether the swap occurred and therefore the value of `secret`.

Overcoming DMP Activation Criteria. To correctly attribute the DMP's activation to `ptr` being moved from `b` to `a`, the attacker must ensure that the DMP's activation criteria are only satisfied when accessing `a` (and not `b`). Based on Section 4.2, one necessary prerequisite to activate the DMP on an `aop` load is to evict the `aop` from the L2 cache. Thus, we need a means to evict `a`[8] (but not `b`).

Overcoming Address Space Separation. Yet, since the attacker runs in a separate process from the victim and without any shared memory, we must replace the Flush+Reload in Section 4 with Prime+Probe. In particular, we must build an eviction set to detect whether `ptr` was dereferenced by the DMP inside the victim process. However, it is not clear how to build eviction sets for `ptr`'s target line (or `a` mentioned above),[9] as we cannot time accesses to these since they are located inside the victim's address space.

5.2 Compound Eviction Set Construction

We now present a novel technique—compound eviction set generation—which solves the above problem by using `ct-swap`'s access to `a` as well as DMP dereferences to `ptr` to simultaneously build eviction sets for both elements.

Establishing a Timing Source. To start, we need to distinguish between L2 hits and misses. However, as the attacker is running without elevated privileges, it is unable to access nanosecond-accurate timers on Apple CPUs, instead being limited to the system's 42 ns timer. Unfortunately, we empirically find that this timer is not sufficient to reliably mount Prime+Probe attacks. We sidestep this issue by using the multi-thread timer approach of [46, 69, 73]. Here, the main idea is to use a dedicated counting thread, which constantly increments a shared variable with the attacker process in a tight loop. By loading the value of the shared variable, the attacker

[7] `constant_time_cond_swap_64`: https://github.com/openssl/openssl/blob/1751185154ab1f1a796e0f39567fe51c8e24b78d/include/internal/constant_time.h.

[8] Triggering the DMP also requires that `a` is refilled after it is evicted. We rely on the victim to perform this refill. For example, `ct-swap` reads `a` in a loop, which will cause each cache line making up `a` to be accessed (refilled) multiple times (`len` > 1).

[9] We assume that the base addresses of `a` and `b` have different page-offset bits, so that the eviction set for `a` would not evict `b`, which also holds for later attacks.

process is thus able to obtain high resolution timestamps, allowing us to distinguish L2 hits from misses.

Generating Standard Eviction Sets. Next, we need to generate a large number of standard L2 eviction sets, i.e., eviction sets targeted to individual L2 sets. The M1 has 8192 (2^{13}) L2 cache sets, indexed with 6 (upper) bits from the physical page frame and 7 (lower) bits from the page offset. We generate standard eviction sets for all these L2 sets by extending the technique used in Section 4.2 (detailed in Appendix A).

Generating Compound Eviction Sets. With all standard L2 eviction sets in hand, we now need to test which of these are capable of evicting the target of `ptr`. As described in Section 5.1, this is non-trivial because observing the dereference of `ptr` via DMP activations requires an eviction set for `a` which we cannot create with standard techniques.

To solve this problem, we will build and test what we call *compound eviction sets*, which simultaneously evict both the target of `ptr` and `a`. We build candidate compound eviction sets as pairs ($\text{EV}_{\texttt{a}}$, $\text{EV}_{\texttt{ptr}}$) of standard L2 eviction sets, where $\text{EV}_{\texttt{a}}$ (respectively $\text{EV}_{\texttt{ptr}}$) is an eviction set whose page-offset bits are compatible with `a` (respectively `ptr`).

We proceed as follows. First, the attacker will place `ptr` in both `a` and `b`. This is so that dereferences to `a` can occur regardless of the secret value. Next, the attacker tests whether each candidate compound set, denoted ($\text{EV}_{\texttt{a}}$, $\text{EV}_{\texttt{ptr}}$), can evict both `a` and `ptr`'s target by priming all lines in $\text{EV}_{\texttt{ptr}}$ and continuously traversing $\text{EV}_{\texttt{a}}$, and then probing/timing $\text{EV}_{\texttt{ptr}}$. If the probe results in an L2 miss, the target of `ptr` filled the cache and displaced a line in $\text{EV}_{\texttt{ptr}}$. This simultaneously implies that $\text{EV}_{\texttt{a}}$ evicted `a` because evicting `a` is the only way that the DMP would have dereferenced `ptr`. If the probe results in all L2 hits, either $\text{EV}_{\texttt{a}}$ or $\text{EV}_{\texttt{ptr}}$ were not eviction sets for `a` or `ptr`, respectively.

The complexity of compound eviction set finding is proportional to the number of possible candidates. With the knowledge of page offsets of `a` and `ptr`, the attacker can reduce the number of potential L2 sets each of them maps to from 8192 to 64. Meanwhile, $\text{EV}_{\texttt{a}}$ only needs to be a superset of the standard eviction. We group 8[10] standard eviction sets as one $\text{EV}_{\texttt{a}}$ in our PoCs, which leads to 512 candidates overall. We run our compound eviction sets construction algorithm 10 times and the mean time for all L2 eviction set generation is 263.9 seconds, while 113.6 seconds for finding the compound eviction set. Overall, we find that we are able to reliably construct these compound eviction sets using the above-described technique, allowing us to proceed to using the DMP in order to recover `secret` from within `ct-swap`'s address space.

5.3 Proof-of-Concept Results

With the compound eviction set ($\text{EV}_{\texttt{a}}$, $\text{EV}_{\texttt{ptr}}$) for `a` and `ptr`'s target in hand, we now demonstrate a proof-of-concept attack on `ct-swap` to learn the secret `secret`. For all proof-of-concept attacks, we use three attacker processes. The first process establishes a TCP connection with the victim process and transmits the value of `ptr` to the victim. The victim process upon receiving `ptr` subsequently executes `ct-swap(a,b,secret)` where `a` is some dummy value, `b` is full of multiple copies of `ptr`, and `secret` is a hardcoded value. In parallel, we use the second attacker process to continuously traverse $\text{EV}_{\texttt{a}}$, evicting `a` from the CPU's L2 cache during the execution of Line 7 of Listing 2. Finally, the third attacker process provides a high-resolution timing source via a counting thread that constantly increments a shared variable.

After transmitting the value of `ptr` to the victim, our first attacker process uses the Prime+Probe channel built on $\text{EV}_{\texttt{ptr}}$ to monitor the DMP activation. We perform 3200 attack trials,[11] for both values of `secret`. Figure 9 summarizes our findings, with the timing distributions for `secret=1` and `secret=0` being clearly distinguishable.

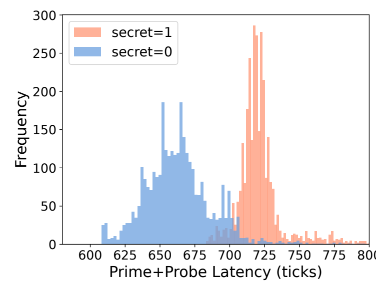


Figure 9: Prime+Probe latency of constant-time swap subroutine. If `ptr` shows up in `a` (`secret=1`), the attacker observes a high latency; otherwise (`secret=0`) it observes a low latency.

6 Attacking Classical Cryptography

We demonstrate that Go's RSA implementation and OpenSSL's Diffie-Hellman Key Exchange (DHKE) implementation, despite being constant-time, can leak secrets via the DMP side-channel. Both systems are otherwise secure against malicious inputs, but feature subroutines that activate the DMP based on the secret key. We draw inspiration from prior chosen-ciphertext side-channel attacks [4, 5, 9, 19, 20, 39, 40, 48, 53, 61, 90, 93], and adapt those techniques for the specific implementations considered in this section.

6.1 Go's RSA Encryption

Our targeted RSA implementation uses Montgomery multiplication, which implicitly blinds the RSA secret key except

[10] The group size is not "the bigger the better", since $\text{EV}_{\texttt{a}}$ needs to evict array `a` before the victim loads `a`.

[11] Based on Section 4.2, the attacker has to reset the history filter to achieve re-dereferences to the same `ptr`. In our PoC, while the methods discussed in Section 4.2 could help, we experimentally find that the TCP socket code used in the victim process to receive inputs from the attacker generates a sufficient amount of traffic to reset the history filter.

during a single, necessary modular operation.[12] We find that an attacker can craft ciphertexts to exploit this modular operation and extract a partial RSA secret key by observing DMP activations.[13] They can then use the standard Coppersmith method to recover the entire RSA secret key [30, 71].

Go's RSA (1.20+) encryption overview. RSA is a public-key cryptosystem. Go (1.20+) RSA implementation follows the specification in RFC 8017 [63]. RSA has a public exponent e (65537 in Go's RSA). An RSA secret key consists of two primes p and q, and an integer d such that $ed \equiv 1 \bmod (p-1)(q-1)$. An RSA public key is $(N = p*q, e)$. Without loss of generality we assume $p > q$. Go's RSA uses the Chinese remainder theorem (CRT) to accelerate decryption.

PoC overview. In our PoC, we target Go's RSA-2048 (1.20). Similarly to [93], our threat model assumes that the victim (server) generates a pair of static public and secret keys. The attacker sends a ciphertext to the victim, and the victim decrypts the ciphertext using its secret key.[14] The public N is 2048 bits long, and the secret p and q are about 1024 bits long. Factoring N into p and q breaks RSA-2048. In our PoC, the attacker extracts the 560 most significant bits of p by observing DMP activations, and then uses the Coppersmith method to break RSA-2048.

DMP-vulnerable subroutine in Go's RSA. Listing 3 shows the DMP-vulnerable subroutine in Go's RSA `Decrypt`. `Decrypt` takes in an RSA secret key and a ciphertext c, and outputs a plaintext $m = c^d \bmod N$. Due to CRT, `Decrypt` breaks this exponentiation into two: $m = c^{D_p} \bmod p$ and $m = c^{D_q} \bmod q$, where D_p and D_q are CRT-related parameters.

The first step of $m = c^{D_p} \bmod p$ is to compute $t_0 = c \bmod p$. A key observation is that if $c < p$, t_0 remains as c. On the other hand, if $c \geq p$, t_0 becomes $c - l * p$, which is unpredictable because l is an unknown integer. Suppose c contains a `ptr`:

- If $c < p$, $t_0 = c$ contains the `ptr` and activates the DMP.
- If $c \geq p$, $t_0 \neq c$ is random and does not activate the DMP.

In this case, we can extract p bit by bit by observing DMP activations resulting from loading t_0. This allows us to treat t_0 as the AoP `a` in Section 5.[15]

Challenge ciphertext construction. Next, we show how to construct c to extract the 560 most significant bits of p (one at a time). In Figure 10, assume the attacker has already recovered the $n-1$ most significant bits of p and targets the n-th bit. Since p is 1024-bit, the attacker sets the leading 1024 bits of the 2048-bit c to be 0. They set the next $n-1$ bits of c to be the recovered $n-1$ bits of p, and the n-th bit of c to be 1. Then, the attacker sets the remaining bits of c to be all 0, except the lower 448 bits that are filled with 7 64-bit `ptrs`. The 16 bits immediately before the `ptrs` are always set to 0 and unused.

```
// m = c ^ Dp mod P
m  = bigmod.NewNat().Exp(t0.Mod(c, P), // t0 = c mod P
priv.Precomputed.Dp.Bytes(), P)
// m2 = c ^ Dq mod Q
m2 = bigmod.NewNat().Exp(t0.Mod(c, Q), // t0 = c mod Q
priv.Precomputed.Dq.Bytes(), Q)
```

Listing 3: DMP-vulnerable subroutine in Go's RSA (1.20) `Decrypt`. c is the attacker's challenge ciphertext that contains a `ptr`. t_0 functions as the AoP `a` in Section 5 because $t_0 = c \bmod p$ would activate the DMP if and only if $c < p$. Attacker can then extract p adaptively by observing DMP activations.

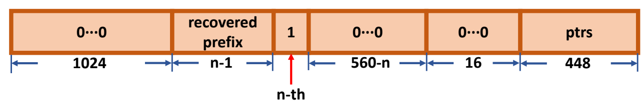


Figure 10: Challenge ciphertext construction to leak the n-th most significant bit of the p in Go's RSA-2048.

Assuming $p > q$, if the attacker observes no DMP activation from $t_0 = c \bmod p$, they can conclude that $c \geq p$ and the n-th bit of p is 0 with $1 - \frac{1}{2^{576-n}} \approx 1$ probability. On the other hand, if the attacker observes the DMP activation, they can conclude that $c < p$ and the n-th bit of p must be 1. Since $\frac{1}{2^{576-n}}$ becomes non-negligible as n approaches 576, we stop the attack at $n = 560$.

Experimental result. We now use the previous ciphertext construction strategy and the Coppersmith method to extract the full RSA secret key.[16] When targeting each of the 560 top bits of p, we collect 32 Prime+Probe latency data points to mitigate background noise. The median of the collected data is then compared to a profiled threshold: 742 ± 38 ticks when c triggers the DMP activation versus 664 ± 124 ticks when c does not trigger. We repeat the experiment targeting bit n if the collected data are outliers due to system noise. The end-to-end attack takes 49 minutes on average to finish. More details about compound eviction set generation and noise tolerance for Go's RSA are in Appendix B.

6.2 OpenSSL Diffie-Hellman Key Exchange

Our targeted OpenSSL DHKE implementation utilizes a window-based exponentiation algorithm. This creates a vulnerability given DMP: if an attacker crafts a malicious public key and correctly guesses the target window of the secret key, a multiplication subroutine will generate a `ptr` value. The attacker can then exploit DMP activations to adaptively extract the DH secret key.

[12] Updates to Go 1.20 cryptography: https://words.filippo.io/dispatches/go-1-20-cryptography/

[13] In Sections 6 and 7, DMP activation particularly refers to DMP dereferences to the attacker-chosen `ptr`.

[14] The attack does not apply to the RSA signature scheme because signatures are calculated as $s = h^d \bmod N$, where h is the message hash. Since hash is a one-way function, the attacker does not have precise control of h.

[15] t_0 is a Go `bigmod.Nat` whose internal representation is an array of 64-bit integers (on 64-bit machine).

[16] The implementation of the Coppersmith method used by our paper: https://github.com/mimoo/RSA-and-LLL-attacks

Cryptography	Online Time (minutes) ❶	❷	❸	Offline Time (minutes)
RSA-2048	5	18	26	∼ 0
DH-2048	5	6	127	∼ 0
Kyber-512	6	10	43	286
Dilithium-2	5	13	577	274

Table 1: Experimental results of four cryptographic attack PoCs. We show the mean of three runs of each PoC. Online time refers to the required time for a co-located attacker process, which includes ❶ standard eviction sets generation; ❷ compound eviction set finding; ❸ DMP leakage. Offline time is the post-processing (e.g. lattice reduction) time to complete secret key recovery. We do not include the time for the offline signature collection phase of Dilithium-2.

OpenSSL DHKE (1.1.1q) overview. DHKE allows two parties, Alice and Bob, to agree on a shared secret over an insecure channel [34]. The public parameters are a prime p and a generator g that generates a cyclic order-q subgroup of $\mathbb{Z}_p^*$. DHKE requires p to be a safe prime such that $q = \frac{p-1}{2}$.[17] Alice and Bob generate their own secret keys $x \in \mathbb{Z}_q$ and $y \in \mathbb{Z}_q$. Alice sends her public key $g^x \bmod p$ to Bob and Bob sends his $g^y \bmod p$ to Alice. They both compute the shared secret $(g^x)^y \bmod p = (g^y)^x \bmod p$. The security of DHKE relies on the computational Diffie–Hellman (CDH) assumption that given $g^x \bmod p$, $g^y \bmod p$, and g, it is computationally difficult to compute $g^{xy} \bmod p$.

PoC overview. Following [41,61], our threat model assumes that the victim (server) and attacker (client) do a DH key exchange. The victim (server) generates a random 2048-bit DH public parameter p and shares it with the attacker (client). The victim generates its own static secret key s. The attacker sends a challenge public key c to the victim, who computes $c^s \bmod p$ using the OpenSSL window-based exponentiation. The window size w is 6. The attacker extracts s window after window by observing DMP activations.

DMP-vulnerable subroutine in OpenSSL DHKE. The victim breaks s into k windows $s_0\|s_1\|...\|s_{k-1}$ with each window of size w. Listing 4 shows a simplified version of the algorithm that computes $c^s \bmod p$ window by window, where we replace most of the code with descriptive comments and only highlight the DMP-vulnerable subroutine `bn_mul_mont_fixed_top`. To start with, a variable `tmp` is initialized to c^{s_0}. At the end of each while loop iteration i (i starts from 1), $\texttt{tmp} = c^{s_0\|...\|s_i}$.

During iteration i, an invariant holds after Line 5: $\texttt{tmp} = (c^{s_0\|s_1\|...\|s_{i-1}})^{2^w}$. If the attacker already recovered the prefix $s_0\|s_1\|...\|s_{i-2}$, they can guess s_{i-1} and construct c strategically. If their guess is correct, `tmp` will contain a `ptr` after Line 5, triggering the DMP. Consequently, the subroutine `bn_mul_mont_fixed_top` is DMP-vulnerable, and `tmp` is the AoP `a` in Section 5.

```
1 // tmp = c^(s0)
2 while (bits > 0) {
3   for (i = 0; i < w; i++)
4     if (!bn_mul_mont_fixed_top(&tmp, &tmp, &tmp, mont, ctx))
5       goto err;
6   // bits -= w;
7   // tmp = tmp * c^(si)
8 }
```

Listing 4: `bn_mul_mont_fixed_top` is our DMP-vulnerable subroutine in OpenSSL DHKE (1.1.1q). The secret key s is broken into k windows $s_0\|s_1\|...\|s_{k-1}$ with a window size w. An attack who knows $s_0\|s_1\|...\|s_{i-2}$ can guess s_{i-1} and construct c such that if the guess is correct, `tmp` contains `ptr` after Line 5. The attacker can then extract s adaptively.

Challenge public key construction. Next, we show how to construct the challenge public key c. All multiplication is in Montgomery form, so every operand is pre-multiplied with a public constant R. Assume the attacker already recovered $s_0\|s_1\|...\|s_{i-2}$. To target s_{i-1}, the attacker makes a guess of its value and constructs c by solving the equation

$$(c^{s_0\|s_1\|...\|s_{i-1}})^{2^w} \cdot R \equiv \texttt{tmp} \mod p \tag{1}$$

where the 2048-bit attacker-controlled output buffer `tmp` contains a `ptr`.

Let E denote the exponent $(s_0\|s_1\|...\|s_{i-1}) * (2^w)$. We start by assuming that the exponent E is an odd number.

If E is an odd number, we first move R to the right-hand side of the equation by doing an inverse:

$$c^E \equiv R^{-1} \cdot \texttt{tmp} \bmod p \tag{2}$$

Since p is a safe prime, $gcd(p-1, E) = 1$ (E is odd), the modular inverse E^{-1} ($E^{-1} \cdot E \equiv 1 \mod (p-1)$) exists due to Fermat's little theorem, and c can be solved as $(\texttt{tmp} \cdot R^{-1})^{E^{-1}} \bmod p$ [39].

However, $E = (s_0\|s_1\|...\|s_{i-1}) * (2^w)$ is an even number. An even number can be factorized as an odd number multiplied by 2^n. In order to convert an even number to an odd number, we need to eliminate the 2^n. Tonelli-Shanks algorithm explains how to calculate the modular square root for n times, but only half of the elements in $\mathbb{Z}_p^*$ are quadratic residues, which means that a given number might not have recursive modular square roots of depth-n [74,81] . This problem can be overcome because `tmp` has 32 64-bit elements and only one needs to be the `ptr`. We can adjust any of the other 31 64-bit elements to ensure that $\texttt{tmp} \cdot R^{-1} \bmod p$ has recursive modular square roots of depth-n. Once the 2^n factor is eliminated, we can apply the odd-E case outlined above.

Experimental result. For a target window i, there are only 2^w (64) possible values of s_i. For each guess of s_i, we collect 32 Prime+Probe latency data points to mitigate background noise. We repeat an experiment if the collected data contains

[17] Why Diffie-Hellman prefers safe primes: https://www.johndcook.com/blog/2017/01/12/safe-primes-sylow-theorems-and-cryptography/

outliers due to system noise. We compare the median of our collected data with a profiled threshold to determine if our guess of s_i is correct. After testing all 64 values, we expect to observe 1 high Prime+Probe latency (correct guess) and 63 low Prime+Probe latencies (incorrect guesses). If the number of positive and negative measurements deviates from this, we redo the experiment for this window. When the challenge public key c triggers the DMP, the Prime+Probe latency is 701 ± 65 ticks, compared to 641 ± 10 when it does not. The experiment takes 2.3 hours to complete, and we extract the victim secret key s. Appendix B provides further details about compound eviction set generation and noise tolerance for OpenSSL DHKE.

7 Attacking Post-Quantum Cryptography

We demonstrate that the implementation of two CRYSTALS cryptographic primitives, Kyber and Dilithium, though designed to be constant-time, can leak secrets via the DMP side channel.[18] Kyber is an IND-CCA2-secure (secure against adaptive chosen-ciphertext attack) NIST-selected key encapsulation mechanism (KEM) [12]. Dilithium is a NIST-selected digital signature scheme [57]. Both Kyber and Dilithium rely on the hardness of Module-LWE (MLWE).

Notation: R denotes the ring $(\mathbb{Z}[x]/x^n+1)$. R_q denotes the ring $(\mathbb{Z}_q[x]/x^n+1)$. R_q^k denotes the space of length-k vectors whose elements are in R_q. $R_q^{k\times l}$ denotes the space of $k\times l$ matrices whose elements are in R_q. For a polynomial p, $p[i]$ denotes the i-th coefficient of p. For a vector $\mathbf{v}$, $\mathbf{v}[i]$ denotes the i-th polynomial of $\mathbf{v}$, and $\mathbf{v}[i][j]$ denotes the j-th coefficient of $\mathbf{v}[i]$. For a vector $\mathbf{v} \in R_q^k$ (or matrix $\mathbf{A} \in R_q^{k\times l}$), $\mathbf{v}^T$ (or $\mathbf{A}^T$) denotes its transpose. $\lceil x \rfloor$ denotes rounding x to the closest integer, rounding up in the case of ties. B_η and S_η denote the centered binomial and uniform random distribution respectively. A number sampled from B_η or S_η is within the range $[-\eta,\eta]$. When we say that $\mathbf{v} \in R_q^k$ is sampled from B_η (S_η), we mean that each coefficient of polynomials in $\mathbf{v}$ is sampled from B_η (S_η). B_τ denotes the set of sparse polynomials in R where τ coefficients are either -1 or 1 and the rest are 0.

7.1 Kyber

Kyber decapsulation relies on a decryption subroutine. Decryption failure leaks the Kyber secret key [35, 66–68, 75]. While Kyber does not expose decryption failures to the attacker, the attacker can use the DMP side channel to construct a decryption failure oracle and then extract the secret key.

Kyber overview. A KEM uses a public key encryption (PKE) scheme to secure symmetric key material. Kyber builds upon a PKE scheme called Kyber.CPAPKE, which is chosen-plaintext secure (CPA-secure). Kyber is a Fujisaki-Okamoto (FO) transformation of the underlying Kyber.CPAPKE, which turns a CPA-secure PKE into a IND-CCA2-secure KEM [38].

Kyber.CPAPKE key generation samples the secret key $\mathbf{s},\mathbf{e} \in R_q^k$ from B_{η_1}, with η_1 being a small integer. The public key consists of $\mathbf{t} \in R_q^k$ and a random $\mathbf{A} \in R_q^{k\times k}$, where $\mathbf{t} = \mathbf{A}\mathbf{s}+\mathbf{e}$.[19] Leaking either $\mathbf{s}$ or $\mathbf{e}$ breaks Kyber.

Kyber.CPAPKE encryption takes in the public key $(\mathbf{t},\mathbf{A})$, a 256-bit message M, and a seed r as the source of randomness. $M = M_0M_1...M_{255}$ is converted to a polynomial $m_p \in R_q$, where $m_p(x) = \sum_{i=0}^{255} M_i * \lceil \frac{q}{2} \rfloor * x^i$. Then, it samples $\mathbf{r} \in R_q^k$ from B_{η_1}, $\mathbf{e_1} \in R_q^k$ from B_{η_2}, and $e_2 \in R_q$ from B_{η_2}, with η_1 and η_2 being small integers. It computes $\mathbf{u} = \mathbf{A}^T\mathbf{r}+\mathbf{e_1}$, and $v = \mathbf{t}^T\mathbf{r}+e_2+m_p$. The ciphertext is $(\mathbf{u}, v)$.

Kyber.CPAPKE decryption takes in the ciphertext $(\mathbf{u}, v)$, and the secret key $(\mathbf{s},\mathbf{e})$. It computes $v - \mathbf{s}^T\mathbf{u} = m_p + \mathbf{e}^T\mathbf{r} + e_2 - \mathbf{s}^T\mathbf{e_1}$. Coefficients in m_p are either 0 or $\lceil \frac{q}{2} \rfloor$. Coefficients in $\mathbf{e}^T\mathbf{r}+e_2-\mathbf{s}^T\mathbf{e_1}$ are small integers. Decryption recovers the plaintext M by rounding each coefficient of $v-\mathbf{s}^T\mathbf{u}$ to 1 if the coefficient is closer to $\lceil \frac{q}{2} \rfloor$ than to 0; and to 0 otherwise.

Decryption failure occurs with negligible probability when processing normal ciphertexts. Let M' denote the decrypted plaintext. If decryption fails, resulting in $M'_i \neq M_i$ (the i-th bit is flipped), this happens only if the i-th coefficient of the error vector $(\mathbf{e}^T\mathbf{r}+e_2-\mathbf{s}^T\mathbf{e_1})[i] \geq \lceil \frac{q}{4} \rfloor$.

PoC overview. We target the Kyber-512 reference implementation, where $n = 256$, $q = 3329$, $k = 2$, $\eta_1 = 3$, and $\eta_2 = 2$. Our threat model assumes that the victim (server) and attacker (client) want to derive a shared secret using Kyber. The victim (server) generates a pair of static Kyber secret and public keys. The secret $\mathbf{s}$ has two ($k = 2$) polynomials, each with 256 coefficients. The attacker guesses a value for $\mathbf{s}[i][j]$ and then crafts a plaintext M containing a `ptr`. They encrypt M using the victim's public key and send the ciphertext to the victim for decryption.

DMP-vulnerable subroutine in Kyber. Kyber's DMP-vulnerable subroutine is `indcpa_dec`, the CPAPKE decryption function. It decrypts the challenge ciphertext that encrypts a plaintext M containing a `ptr`, and stores the decrypted M' into a buffer `buf`. If the decryption is successful, $M' = M$ and `buf` contains `ptr`. Otherwise, $M' \neq M$ and `buf` does not contain `ptr`.

Kyber is CCA secure. Decapsulation would reject a malformed ciphertext without exposing $M' = M$ or $M' \neq M$ to the attacker. However, the attacker can learn decryption failure or success by observing whether `ptr` is dereferenced by the DMP. This behavior is not an implementation issue but fundamental to the FO transform. As a result, subroutine `indcpa_dec` is DMP-vulnerable and `buf` is the AoP `a` in Section 5.

[18] CRYSTALS: Cryptographic Suite for Algebraic Lattices https://pq-crystals.org/index.shtml

[19] The security level of Kyber scales with k. A MLWE matrix from $R_q^{k\times k}$ is analogous to a $nk \times nk$ matrix in LWE.

Challenge ciphertext construction. We demonstrate how to construct a ciphertext $(\mathbf{u}, v)$ that allows the attacker to build a decryption failure oracle using DMP activations. Recall:

$$\begin{aligned} \mathbf{u} &= \mathbf{A}^T\mathbf{r} + \mathbf{e_1} \\ v &= \mathbf{t}^T\mathbf{r} + e_2 + m_p \end{aligned} \tag{3}$$

Suppose the attacker attempts to learn the first coefficient of the first polynomial in $\mathbf{s}$: $\mathbf{s}[0][0]$. They prepare a plaintext M with a `ptr` in $M_{0...63}$ and fill the rest with 0s: $M = \texttt{ptr}||00...00$. They manipulate other variables to ensure the following: if $0 < \mathbf{s}[0][0]$, $(\mathbf{u}, v)$ decrypts to M; otherwise, it decrypts to M' with the first bit flipped ($M'_0 = M_0 \oplus 1$). To achieve this, they can set $\mathbf{r} = (0,0)$ (a length-2 vector of degree-0 polynomials), $e_2 = \lceil \frac{q}{4} \rfloor$ (a degree-0 polynomial), and $\mathbf{e_1} = (1,0)$. This results in a ciphertext of $\mathbf{u} = (1,0)$, $v = m_p + e_2$.

Decryption computes $v - \mathbf{s}^T\mathbf{u}$:

$$\begin{aligned} v - \mathbf{s}^T\mathbf{u} &= m_p + e_2 - \mathbf{s}^T\mathbf{e_1} \\ &= m_p + e_2 - (\mathbf{s}[0], \mathbf{s}[1])^T(1,0) \\ &= m_p + \lceil \frac{q}{4} \rfloor - \mathbf{s}[0] \end{aligned} \tag{4}$$

The first entry of $v - \mathbf{s}^T\mathbf{u}$ is $m_p[0] + \lceil \frac{q}{4} \rfloor - \mathbf{s}[0][0]$, containing a deliberately introduced large error $\lceil \frac{q}{4} \rfloor - \mathbf{s}[0][0]$. Decryption would fail if $\lceil \frac{q}{4} \rfloor - \mathbf{s}[0][0] \geq \lceil \frac{q}{4} \rfloor$. The ciphertext construction ensures that decryption failure depends on the value of $\mathbf{s}[0][0]$:

- If $\lceil \frac{q}{4} \rfloor - \mathbf{s}[0][0] < \lceil \frac{q}{4} \rfloor$ $(0 < \mathbf{s}[0][0])$, $M' = M = \texttt{ptr}||00...00$, and `buf` activates the DMP.
- If $\lceil \frac{q}{4} \rfloor - \mathbf{s}[0][0] \geq \lceil \frac{q}{4} \rfloor$ $(0 \geq \mathbf{s}[0][0])$, $M' \neq M$ because the first bit is flipped ($M'_0 = M_0 \oplus 1$), and `buf` cannot activate the DMP.

The attacker can learn the exact value of $\mathbf{s}[0][0]$ by tuning e_2 and observing DMP activations. To trigger DMP activation on `buf` (e.g., eviction set construction), we employ the same method as the chosen-input attack from Section 5.1.

We now present a simplified version of our attack. Kyber includes an extra compression and decompression step. In [25], we detail how to overcome the compression when constructing the challenge ciphertext.

The secret key has $256 \times 2 = 512$ coefficients. Ideally, the attacker should be able to apply the same process above to target any coefficient of $\mathbf{s}$. However, due to findings in Section Section 4.3, the DMP cannot leak every coefficient of $\mathbf{s}$: If we break $\mathbf{s}$ into chunks of 64, the leading 8 and trailing 7 bits are not recoverable via the DMP. As a result, 392 out of 512 coefficients can be recovered by observing DMP activations. We feed the recovered 392 coefficients as 392 hints into the lattice reduction tool from May et al., to recover the entire secret key [59]. In [25], we provide more details about why certain coefficients are not recoverable, and how we use the lattice reduction tool.

Experimental result. In our PoC, there are 392 recoverable secret coefficients. We construct 8 challenge ciphertexts to adaptively learn each coefficient, as its potential value ranges from -3 to 3. See [25] for why we need 8 ciphertexts. For each ciphertext, we collect 32 Prime+Probe latency data points to mitigate background noise. We repeat the experiment if the data we collect contains outliers due to system noise. We compare the median of our collected data with a profiled threshold to determine the activation status of the DMP. When the ciphertext triggers the DMP (decryption succeeds), the Prime+Probe latency is 713 ± 22 ticks, compared to 616 ± 14 ticks when it does not (decryption fails). The experiment takes 59 minutes to complete. After that, we spend another 5 hours on lattice reduction to extract the entire secret key. More details about compound eviction set generation and noise tolerance for Kyber are in Appendix B.

7.2 Dilithium

Dilithium relies on the "Fiat-Shamir with Aborts" [57], and its security depends on the privacy of its nonce $\mathbf{y}$ [56]. Dilithium is secure against chosen-message attacks, meaning a polynomial-time attacker cannot learn secret information by observing signatures. However, Dilithium might generate data in $\mathbf{y}$ that resembles a pointer. By monitoring DMP activations, an attacker could obtain knowledge of $\mathbf{y}$, derive linear equations involving the secret key, and ultimately extract the entire secret key. Prior research has explored similar attacks that exploit side channels to learn intermediate values during Dilithium signing, allowing secret key reconstruction [15, 28, 49, 58, 77, 87].

Dilithium overview. *Dilithium key generation* samples the secret key $\mathbf{s_1} \in R_q^l$ from S_η and $\mathbf{s_2} \in R_q^k$ from S_η, with η being a small integer. The public key consists of $\mathbf{t} \in R_q^k$ and a random $\mathbf{A} \in R_q^{k \times l}$, where $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$. Leaking either $\mathbf{s_1}$ or $\mathbf{s_2}$ breaks Dilithium. Dilithium also has a public key compression, which we discuss in [25].

Dilithium signature generation uses rejection sampling to generate digital signatures [56]. In Algorithm 1 we present a simplified version that focuses on the part relevant to our attack. The algorithm generates a signature $(\mathbf{z}, c)$ of a message M using the secret key $\mathbf{s}_1$. $\mathbf{z}$ is initialized to $\perp$ (Line 2). In a while loop, the algorithm samples a private nonce $\mathbf{y}$, which is a length-l vector of polynomials with coefficients randomly sampled from $[-\gamma_1, \gamma_1]$ (Line 4). Then, the algorithm samples a random c from B_τ, and c depends on M (Line 5). c is a sparse polynomial with exactly τ number of 1 or -1, and the non-zero entries have randomized positions. The algorithm computes $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ (Line 6), but will only accept $(\mathbf{z}, c)$ if it leaks no secrets, and reject $(\mathbf{z}, c)$ otherwise. Note that $\mathbf{y}$ must be kept private. Leaking $\mathbf{y}$ leaks $\mathbf{s}_1 = \frac{\mathbf{z}-\mathbf{y}}{c}$. Leaking partial information of $\mathbf{y}$ might also compromise $\mathbf{s}_1$ [18].

PoC overview. The victim (a Dilithium signing server) generates a pair of Dilithium secret and public keys. Our threat

```
Sign(s₁,M)
z := ⊥
while z = ⊥ do
    y ← S^ℓ_γ1 // A length-l vector of random
        and small polynomials
    c ∈ B_τ // A sparse polynomial (depending
        on M) with τ number of 1 or -1
    z := y + cs₁
    // Reject z (set z to ⊥) if z leaks
        information about the secret key
    return (z, c)
end
```

Algorithm 1: The main body of the Dilithium sign algorithm is a while loop that creates a signature $(\mathbf{z}, c)$ of message M under the secret key $\mathbf{s}_1$. The algorithm returns $(\mathbf{z}, c)$ if it does not leak any secret information.

model assumes that the victim is a signing server. The attacker can choose arbitrary messages and request digital signatures from the victim. The attacker can parse the signatures offline and replay certain messages later.

We target CIRCL's implementation of deterministic Dilithium-2 (written in Go), where $n = 256$, $q = 8380417$, $k = l = 4$, $\gamma_1 = 2^{17}$, $\eta = 2$, and $\tau = 39$ [37].[20] Dilithium is deterministic when the private nonce $\mathbf{y}$ in Algorithm 1 is generated with deterministic randomness. Our attack is motivated by the observation: *the server might naturally produce data that resembles a* `ptr` *in* $\mathbf{y}$. While the exact value of $\mathbf{y}$ should remain secret, the underlying MLWE structure allows an attacker to approximate $\mathbf{y}$ through $\mathbf{z}$. If a `ptr` appears in $\mathbf{z}$, the attacker infers its presence within $\mathbf{y}$ and confirms this by observing DMP activations. Successful confirmation reveals partial knowledge of $\mathbf{s}_1$.

Our PoC consists of an offline and online signature collection phase. During the offline phase, the attacker sends m message to the server requesting signatures and collects m' pairs of $\{(\mathbf{z}, c), M\}$, where $\mathbf{z}$ contains a `ptr`. During the online phase, the attacker re-submits the collected m' messages to the server for signatures. The attacker can distinguish which pair $\{(\mathbf{z}, c), M\}$ causes the `ptr` to show up in $\mathbf{y}$ via DMP activations, and then derive a linear equation of $\mathbf{s}_1$. The attacker further uses the lattice reduction tool to recover $\mathbf{s}_1$ [59].

DMP-vulnerable subroutine in CIRCL Dilithium. The DMP-vulnerable subroutine is the $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ in Algorithm 1 Sign. CIRCL uses an array of unsigned 32-bit integers to represent a polynomial. Every coefficient of $\mathbf{y}$ and $\mathbf{z}$ is stored as an unsigned 32-bit integer. We pick $\mathbf{y}$ as the AoP `a` from Section 5. The range of coefficients in $\mathbf{y}$ is $[-2^{17}, 2^{17}]$, and that of coefficients in $c\mathbf{s}_1$ is $[-78, 78]$.

Let's take the first two 32-bit coefficients of $\mathbf{y}$ ($\mathbf{y}[0][0]$, $\mathbf{y}[0][1]$) and $\mathbf{z}$ ($\mathbf{z}[0][0]$, $\mathbf{z}[0][1]$) as an example. Assume that $\mathbf{z}[0][1] \parallel \mathbf{z}[0][0]$ forms a valid 64-bit `ptr`, pointing to the same 4GByte region where $\mathbf{y}$ lives. If we break `ptr` into two 32-bit halves ($\mathtt{ptr}_1 \parallel \mathtt{ptr}_0$), then $\mathbf{z}[0][1] = \mathtt{ptr}_1$ and $\mathbf{z}[0][0] = \mathtt{ptr}_0$. We can derive the range of ($\mathbf{y}[0][0]$, $\mathbf{y}[0][1]$):

$$\begin{aligned}\mathbf{y}[0][1] &= \mathbf{z}[0][1] - c\mathbf{s}_1[0][1] \in [\mathtt{ptr}_1 - 78, \mathtt{ptr}_1 + 78] \\ \mathbf{y}[0][0] &= \mathbf{z}[0][0] - c\mathbf{s}_1[0][0] \in [\mathtt{ptr}_0 - 78, \mathtt{ptr}_0 + 78]\end{aligned} \quad (5)$$

The takeaway from Equation (5) is that if $\mathbf{z}[0][1] \parallel \mathbf{z}[0][0]$ is a `ptr`, $\mathbf{y}[0][1] \parallel \mathbf{y}[0][0]$ might also be a `ptr`!

To elaborate, we know $\mathbf{z}[0][1] \parallel \mathbf{z}[0][0]$ forms a `ptr`. If we want $\mathbf{y}[0][1] \parallel \mathbf{y}[0][0]$ to also form a `ptr`, we only need $\mathbf{y}[0][1] = \mathbf{z}[0][1]$ or $c\mathbf{s}_1[0][1] = 0$. The value of $\mathbf{y}[0][0]$ is less important because $c\mathbf{s}_1[0][0]$ is small, variations in which will only cause $\mathbf{y}[0][1] \parallel \mathbf{y}[0][0]$ to map to the same or an adjacent cache line as `ptr`. As a result, $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ is DMP-vulnerable. If the attacker sets $\mathbf{y}$ as the AoP `a` from Section 5, they can learn $c\mathbf{s}_1[0][1] = 0$ by observing DMP activations. The same idea applies to all other coefficients of $\mathbf{y}$ and $\mathbf{z}$.

Offline and Online signature collection In the offline phase, the attacker sends m random messages for signatures. Recall that $\mathbf{z}$ is a length-4 vector of 256-degree polynomials. The attacker collects m' pairs of $\{(\mathbf{z}, c), M\}$ where for an $i \in [0, 3]$ and an even $j \in [0, 255]$, $\mathbf{z}[i][j+1] \parallel \mathbf{z}[i][j]$ forms a `ptr`, which lives in the same 4GByte region as $\mathbf{y}$.[21]

In the online phase, the attacker re-submits the m' messages collected offline. If the attacker observes a DMP activation, the attacker can deduce that $\mathbf{y}[i][j+1] = \mathbf{z}[i][j+1]$, and derive one linear equation of $\mathbf{s}_1$: $c\mathbf{s}_1[i][j+1] = 0$. After gathering at least 876 linear equations, the attacker uses the lattice reduction tool to recover $\mathbf{s}_1$ [59].

In [25], we discuss more details about our PoC including a theoretical bound of m and m', and how to loose some conditions above for the practicality of the PoC.

Experimental result. In our PoC, we request $m = 4 \times 10^9$ messages during the offline collection phase. We parse the signatures and collect $m' = 3 \times 10^5$ ones with the property that $\mathbf{z}[i][j+1] \parallel \mathbf{z}[i][j]$ forms a `ptr`. In the online phase, resubmitting the m' messages, we observe a Prime+Probe latency of 772 ± 152 ticks when the message triggers the DMP, compared to 657 ± 106 ticks when it does not. To minimize false positives, we accept a message as triggering the DMP only after observing 10 consecutive positive signals. The entire experiment takes 10 hours. An additional 5 hours are spent on lattice reduction to extract the full secret key. More details about compound eviction set generation and noise tolerance for CIRCL Dilithium are in Appendix B.

[20] CIRCL: Cloudflare's Interoperable Reusable Cryptographic Library https://github.com/cloudflare/circl/

[21] Both base addresses of $\mathbf{z}$ and $\mathbf{y}$ are 64-bit aligned, so that entries at even indexes are 64-bit aligned as well.

8 Countermeasures

This paper demonstrates that information disclosure through the Apple m-series DMP is significantly greater than previously believed, and puts constant-time cryptography at risk. A drastic solution would be to completely disable the DMP. However, as doing so will incur heavy performance penalties and is likely not possible on M1 and M2 CPUs,[22] in this section we discuss alternative defensive approaches.

Using Efficiency Cores. As pointed out by Augury [84], the DMP does not activate on code running on Icestorm cores. Thus, a sensible short-term security posture is to run all cryptographic code on Icestorm cores. This strategy is simple, general, and does not require user code changes. Yet, it is brittle because any future Apple part could silently enable the DMP on Icestorm cores. Finally, restricting cryptography to run on Icestorm cores will likely incur a significant performance penalty.

Blinding. An alternative solution is to apply cryptographic blinding-like techniques. For example, by instrumenting the code to add/remove masks to sensitive values before/after being stored/loaded from memory. These ideas could be applied in different ways depending on the sensitive program. For instance, in our attack on Diffie-Hellman Key Exchange, one can generate a random number to mask the secret key for every key exchange [45]. The major downside of this approach is that it requires potentially DMP-bespoke code changes to every cryptographic implementation, as well as heavy performance penalties for some cryptographic schemes.

Ad-Hoc Defenses. Finally, one can imagine point defenses that interfere with specific steps in the attack. For example, changing victims to better validate inputs or scheduling policies to forbid co-location [70]. The downside of these approaches is that they are ad-hoc and leave the root cause (the DMP) unaddressed.

Hardware Support. Longer term, we view the right solution to be to broaden the hardware-software contract to account for the DMP. At a minimum, hardware should expose to software a way to selectively disable the DMP when running security-critical applications. This already has nascent industry precedent. For example, Intel's DOIT extensions specifically mention disabling their DMP through an ISA extension [3]. Longer term, one would ideally like finer-grain control, e.g., to constrain the DMP to only prefetch from specific buffers or designated non-sensitive memory regions.

9 Conclusions

In this paper we showed that DMPs pose a significant security threat to modern software, breaking a wide variety of state-of-the-art cryptographic implementations. At a high level, if the attacker has the ability to secret-dependently write a pointer to memory, the DMP enables it to learn partial or complete information about that secret. While we demonstrate end-to-end attacks on four cryptographic implementations, more programs are likely at risk given similar attack strategies. Given our findings that DMPs also exist on the Apple M2/M3 and Intel 13th Generation CPUs, the problem seemingly transcends specific processors and hardware vendors and thus requires dedicated hardware countermeasures.

[22] We observe that setting the data independent timing (DIT) [1] bit disables the DMP behavior on M3, which is not the case with M1 and M2.

Acknowledgments

This work was partially supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; the Defense Advanced Research Projects Agency (DARPA) under contract numbers W912CG-23-C-0022 and HR00112390029; the National Science Foundation (NSF) under grant numbers 1954712, 1954521, 2154183, 2153388, and 1942888; the Alfred P. Sloan Research Fellowship; and gifts from Intel, Qualcomm, and Cisco.

References

[1] Arm Armv8-A Architecture Registers. https://developer.arm.com/documentation/ddi0595/2021-12/.

[2] ARM Cortex-A Series Programmer's Guide for ARMv8-A. https://developer.arm.com/documentation/den0024/a.

[3] Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[4] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *CSAW*, 2007.

[5] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, 2006.

[6] Onur Aciicmez, Jean-Pierre Seifert, and Cetin Kaya Koc. Predicting Secret Keys via Branch Prediction. *IACR*, 2006.

[7] Sam Ainsworth and Timothy M. Jones. Graph Prefetching Using Data Structure Knowledge. In *ICS*, 2016.

[8] Sam Ainsworth and Timothy M. Jones. An Event-Triggered Programmable Prefetcher for Irregular Workloads. In *ASPLOS*, 2018.

[9] Monjur Alam, Haider Adnan Khan, Moumita Dey, Nishith Sinha, Robert Callan, Alenka Zajic, and Milos Prvulovic. One&Done: A Single-Decryption EM-Based attack on OpenSSL's Constant-Time blinded RSA. In *USENIX Security*, 2018.

[10] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop Van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, 2016.

[11] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-Time implementations. In *USENIX Security*, 2016.

[12] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber algorithm specifications and supporting documentation (version 3.02). *NIST submissions*, 2021.

[13] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, 2014.

[14] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *CSF*, 2018.

[15] Alexandre Berzati, Andersson Calle Viera, Maya Chartouny, Steven Madec, Damien Vergnaud, and David Vigilant. Exploiting intermediate value leakage in dilithium: a template-based approach. In *CHES*, 2023.

[16] Abhishek Bhattacharjee. Breaking the Address Translation Wall by Accelerating Memory Replays. *IEEE Micro*, 2018.

[17] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. Hardware Prefetchers Leak: A Revisit of SVF for Cache-Timing Attacks. In *MICROW*, 2012.

[18] Leon Groot Bruinderink and Peter Pessl. Differential fault attacks on deterministic lattice signatures. *CHES*, 2018.

[19] David Brumley and Dan Boneh. Remote timing attacks are practical. In *USENIX Security*, 2005.

[20] Elad Carmon, Jean-Pierre Seifert, and Avishai Wool. Photonic side channel attacks against rsa. In *HOST*, 2017.

[21] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In *PLDI*, 2020.

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

[23] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a dsl for timing-sensitive computation. In *PLDI*, 2019.

[24] Mustafa Cavus, Resit Sendag, and Joshua J. Yi. Informed Prefetching for Indirect Memory Accesses. *ACM Trans. Archit. Code Optim.*, 2020.

[25] 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. `https://gofetch.fail`, 2024.

[26] Yun Chen, Ali Hajiabadi, Lingfeng Pei, and Trevor E. Carlson. New Cross-Core Cache-Agnostic and Prefetcher-based Side-Channels and Covert-Channels. In *ArXiV*, 2023.

[27] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher. In *ASPLOS*, 2023.

[28] Zhaohui Chen, Emre Karabulut, Aydin Aysu, Yuan Ma, and Jiwu Jing. An efficient non-profiled side-channel attack on the crystals-dilithium post-quantum signature. In *ICCD*, 2021.

[29] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. *ACM SIGPLAN Notices*, 2002.

[30] Don Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In *EUROCRYPT*, 1996.

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

[32] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't mesh around:Side-Channel attacks and mitigations on mesh interconnects. In *USENIX Security*, 2022.

[33] Peter J Denning. Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.

[34] Whitfield Diffie and Martin E Hellman. New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*. 2022.

[35] Jintai Ding, Scott Fluhrer, and Saraswathy Rv. Complete attack on rlwe key exchange with reused keys, without signal leakage. In *ACISP*, 2018.

[36] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[37] Armando Faz-Hernández and Kris Kwiatkowski. *Introducing CIRCL: An Advanced Cryptographic Library*. Cloudflare, June 2019. Available at `https://github.com/cloudflare/circl`. v1.3.3 Accessed May, 2023.

[38] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *CRYPTO*, 1999.

[39] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In *CHES*, 2015.

[40] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO*, 2014.

[41] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *CCS*, 2017.

[42] Google/LLVM. Speculative Load Hardening. `https://llvm.org/docs/SpeculativeLoadHardening.html`, 2018.

[43] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[44] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *NDSS*, 2017.

[45] Da Harkins. Dragonfly key exchange. RFC 7664, November 2015.

[46] Lorenz Hetterich and Michael Schwarz. Branch different-spectre attacks on apple silicon. In *DIMVA*, 2022.

[47] Tyler J Huberty, Stephan G Meier, and Mridul Agarwal. Content-directed prefetch circuit with quality filtering, February 6 2018. US Patent 9,886,385.

[48] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. *IACR*, 2015.

[49] Emre Karabulut, Erdem Alkim, and Aydin Aysu. Single-trace side-channel attacks on ω-small polynomial sampling: With applications to ntru, ntru prime, and crystals-dilithium. In *HOST*, 2021.

[50] Anirudh Mohan Kaushik, Gennady Pekhimenko, and Hiren Patel. Gretch: A Hardware Prefetcher for Graph Analytics. *ACM Trans. Archit. Code Optim.*, 2021.

[51] Taehun Kim, Hyeongjin Park, Seokmin Lee, Seunghee Shin, Junbeom Hur, and Youngjoo Shin. Devious: Device-driven side-channel attacks on the iommu. In *S&P*, 2023.

[52] 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. In *S&P*, 2019.

[53] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

[54] Adam Langley, Mike Hamburg, and Sean Turner. Rfc 7748: Elliptic curves for security, Jan 2016.

[55] 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. In *USENIX Security*, 2018.

[56] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In *ASIACRYPT*, 2009.

[57] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. Crystals-dilithium algorithm specifications and supporting documentation (version 3.1). *NIST submission*, 2021.

[58] Soundes Marzougui, Vincent Ulitzsch, Mehdi Tibouchi, and Jean-Pierre Seifert. Profiling side-channel attacks on dilithium: A small bit-fiddling leak breaks it all. In *SAC*, 2022.

[59] Alexander May and Julian Nowakowski. Too Many Hints - When LLL Breaks LWE. In *ASIACRYPT*, 2023.

[60] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. In *ArXiV*, 2019.

[61] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon attack: Finding and exploiting Most-Significant-Bit-Oracles in TLS-DH (E). In *USENIX Security*, 2021.

[62] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *ICISC*, 2005.

[63] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. Pkcs# 1: Rsa cryptography specifications version 2.2. RFC 8017, November 2016.

[64] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, 2006.

[65] 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. In *USENIX Security*, 2021.

[66] Yue Qin, Chi Cheng, and Jintai Ding. An efficient key mismatch attack on the nist second round candidate kyber. *Cryptology ePrint Archive*, 2019.

[67] Yue Qin, Chi Cheng, Xiaohan Zhang, Yanbin Pan, Lei Hu, and Jintai Ding. A systematic approach and analysis of key mismatch attacks on lattice-based nist candidate kems. In *ASIACRYPT*, 2021.

[68] Gokulnath Rajendran, Prasanna Ravi, Jan-Pieter D'Anvers, Shivam Bhasin, and Anupam Chattopadhyay. Pushing the limits of generic side-channel attacks on lwe-based kems-parallel pc oracle attacks on kyber kem and beyond. In *CHES*, 2023.

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

[70] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*, 2009.

[71] Keegan Ryan and Nadia Heninger. Fast practical lattice reduction through iterated compression. In *CRYPTO*, 2023.

[72] 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. In *ISCA*, 2021.

[73] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *FC*, 2017.

[74] Daniel Shanks. Five number-theoretic algorithms. In *Proceedings of the Second Manitoba Conference on Numerical Mathematics (Winnipeg)*, 1973.

[75] Muyan Shen, Chi Cheng, Xiaohan Zhang, Qian Guo, and Tao Jiang. Find the bad apples: An efficient method for perfect key recovery under imperfect sca oracles–a case study of kyber. In *CHES*, 2023.

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

[77] Hauke Steffen, Georg Land, Lucie Kogelheide, and Tim Güneysu. Breaking and protecting the crystal: Side-channel analysis of dilithium in hardware. In *PQCrypto*, 2023.

[78] 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.

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

[80] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete Information Flow Tracking from the Gates Up. In *ASPLOS*, 2009.

[81] Alberto Tonelli. Bemerkung über die auflösung quadratischer congruenzen. *Nachrichten von der Königl. Gesellschaft der Wissenschaften und der Georg-Augusts-Universität zu Göttingen*, 1891.

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

[83] Stephan Van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. Revanc: A framework for reverse engineering hardware page table caches. In *EuroSec*, 2017.

[84] 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 *S&P*, 2022.

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

[86] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. Meshup: Stateless cache side-channel attack on cpu mesh. In *S&P*, 2022.

[87] Ruize Wang, Kalle Ngo, Joel Gärtner, and Elena Dubrova. Single-trace side-channel attacks on crystals-dilithium: Myth or reality? *IACR*, 2023.

[88] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *USENIX Security*, 2022.

[89] Yingchen Wang, Riccardo Paccagnella, Alan Wandke, Zhao Gang, Grant Garrett-Grossman, Christopher W. Fletcher, David Kohlbrenner, and Hovav Shacham. DVFS frequently leaks secrets: Hertzbleed attacks beyond SIKE, cryptography, and CPU-only data. In *IEEE S&P*, 2023.

[90] Zixuan Wang, Mohammadkazem Taram, Daniel Moghimi, Steven Swanson, Dean Tullsen, and Jishen Zhao. Nvleak: Off-chip side-channel attacks via non-volatile memory systems. In *USENIX Security*, 2023.

[91] Chong Xiao, Ming Tang, and Sylvain Guilley. Exploiting the microarchitectural leakage of prefetching activities for side-channel attacks. *Journal of Systems Architecture*, 2023.

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

[93] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. *IACR*, 2016.

[94] 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.

[95] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO*, 2019.

[96] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. IMP: Indirect Memory Prefetcher. In *MICRO*, 2015.

[97] Xiangyao Yu, Christopher J. Hughes, and Nadathur Rajagopalan Satish. Hardware prefetcher for indirect access patterns, February 2017. US9582422B2.

[98] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. BunnyHop: Exploiting the Instruction Prefetcher. In *USENIX Security*, 2023.

A Standard eviction sets generation algorithm

We now briefly present the method to generate eviction sets covering all L2 sets. We start with generating eviction sets with a fixed page offset, which map to L2 sets differed by upper 6 bits. To this end, we sweep a pool of pages to identify new evict targets, and test if the fixed offset into one of them has conflicts with the current eviction set group. If there is no conflict, it means that this evict target maps to a new L2 set whose eviction set is not included in the current group. With this evict target, we use the techniques from Vila et al. [85] to generate a matching eviction set and add it into the group. Finally, for each of the 64 (2^6) eviction sets, we fix the upper 6 bits, and generate eviction sets for every combination of the lower 7 bits for a total of 8192 eviction sets.

B Compound Eviction Sets Generation and Noise Tolerance Tips

Compound eviction sets generation. To generate a compound eviction set ($\mathsf{EV}_{\texttt{a}}$, $\mathsf{EV}_{\texttt{ptr}}$), the attacker must solve two problems. First, they must identify the address of `a` as well as valid (and quiet) pages to search for `ptr`. Second, they must confirm `ptr` injection to `a`.

For the first problem, both DHKE-2048 (OpenSSL) and Kyber-512 (reference implementation) allocate `a` in the same 4GByte region as the `dyld` cache, which is an ideal target for `ptr`. The virtual address of the `dyld` shared library is only randomized by macOS at boot time and the attacker can recover it with another unprivileged process by running `vmmap`. RSA-2048 (Go) and deterministic Dilithium-2 (CIRCL) allocate `a` in a stable address region beyond `0x14000000000`.[23] Pages in this region are always valid even with ASLR, which makes it an ideal target for `ptr`.

For the second problem, in RSA-2048 (Go), as long as the ciphertext c is smaller than p and q, `ptr` will be preserved in `a`. In DHKE-2048 (OpenSSL), the first window of secret, s_0, is always 1. Hence, the attacker can inject the `ptr` to `a` for the first iteration by solving Equation (2) with $E = 1$. In Kyber-512 (reference), `ptr` can be injected by correctly encrypting message m with `ptr`. Dilithium-2 (CIRCL) is tricky as the attacker cannot confirm `ptr` injection to `y` (`a` in Dilithium), but has a pool of messages from which a subset correctly injects `ptr`. Moreover, the so-called semi-confirmation in Dilithium significantly increases the compound eviction set search space. To address this, the attacker can decouple $\mathsf{EV}_{\texttt{a}}$ and $\mathsf{EV}_{\texttt{ptr}}$ by first targeting `sig.z`, where the attacker can confirm `ptr` injection. Note that the compound eviction set to `sig.z` shares the same $\mathsf{EV}_{\texttt{ptr}}$ with that of `y`, thus having the right $\mathsf{EV}_{\texttt{ptr}}$ makes generating $\mathsf{EV}_{\texttt{a}}$ for `y` efficient.

Noise tolerance. We observed several sources of noise or failure when checking for the existence of `ptr` in `a`.

First, the background noise of the Prime+Probe channel could result in false positives. To address this, we take 32 latency observations (Sections 6 and 7) and apply the following strategies for our cryptography targets. To start off, during the attack process, the attacker also performs the background test accesses by sending random messages. Having these redundant measurements interleaved with normal test accesses establishes confidence that it is the DMP causing high latencies in the normal test accesses. Second, an attacker can detect errors in RSA-2048 (Go) and DHKE-2048 (OpenSSL), and is able to roll back and redo the experiment in such cases. In RSA (Go), if one bit is wrong 0/1, the ciphertext c will always be smaller/larger than p, resulting in the attacker recovering consecutive 1/0 bits, an unlikely pattern in practice. In DHKE-2048 (OpenSSL), if an erroneous s_{i-1} is chosen at the i-th window, the attacker will not observe any DMP signal for the next window, because the challenge c for subsequent windows is based on correctness of s_{i-1}. This method is not applicable for Kyber-512 (reference) and Dilithium-2 (CIRCL), because the recovery of each coefficient in Kyber/Dilithium is independent of the others. An attacker can always repeat the attack several times and take a majority vote.

Second, `a` may change its virtual address at runtime, rendering $\mathsf{EV}_{\texttt{a}}$ ineffective and causing false negatives. To detect this, the attacker must check known-good `ptr` injection to infer the validity of the current compound eviction set. As long as this happens infrequently, the attacker can then re-generate the compound eviction set.

Third, the interval between each load to `a` may be shorter than traversing $\mathsf{EV}_{\texttt{a}}$. One solution is to decrease the size of $\mathsf{EV}_{\texttt{a}}$ until it matches that of a standard eviction set. If traversing a standard eviction set is too expensive, a possible solution is to degrade the performance of the victim program [10].

[23] This specific address is a function of the target program.