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.

### 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 ≤ 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 + δ)` 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 $\delta$.

```c
uint64_t aop[N];
// set of test addresses
// Measure latency to
// to unique addresses // set of test addresses
for (i=0; i<N; i++) {
    *aop[i%N] = aop[i%N];
}
// Fill aop with pointers
// Fill aop with pointers
// or random values // or random values
```

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 $M = 256$, 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 $\sim 250$ 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 column).

As seen in Figure 2b, when the out-of-bounds region contains pointers, test accesses are $< 300$ cycles despite no architectural dereferences occurring into 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 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

\(^1\)We collect timing measurements by configuring and reading performance counters (PML2–PML7) for cycle counting via `kperf`.

\(^2\)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.
success rate (%)

The same patent suggests that this filter may be organized as a direct-mapped 128-entry or 256-entry structure. To corroborate the history filter reverse engineering, we design a new experiment where \( aop \) only contains a single pointer \( pt_r \). First, we access \( aop \), causing the DMP to dereference \( pt_r \). We then evict \( aop \) and the target line for \( pt_r \) 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 \( pt_r \). 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.

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 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 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. 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 \( aop[0], \ldots, aop[7] \). We run 7 more variants of this experiment, varying the single \( aop[i] \) access from \( i = 1, \ldots, 7 \) 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 \( i = 8, \ldots, 15 \). In this case, we observe that \( aop[8], \ldots, aop[15] \) 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 \( i = 0, \ldots, 7 \), 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 \( i = 8, \ldots, 15 \), 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

---

4We empirically verify this by subsequently timing an access to the other half and observing that its access latency corresponds to that of an L2 hit.
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 $\text{aop}$ is uncached. We then try testing different pointer values in the $\text{aop}$, 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 $\text{aop}$ entry and the value of the $\text{aop}$ entry (i.e., the pointer). We call the address of the $\text{aop}$ 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 $\text{aop}$ 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.
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.

**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 \((\log_2 4 \text{ GByte} = 32 \text{ bits})\)-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.

\(^5\)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.

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.

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.

![Success Rate](image)

**Figure 8:** We test four access patterns shown as the rows in Figure 1 on Apple M2 (left) and Intel 13th generation Raptor Lake (right).
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 \(\text{secret}\). 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 \(\text{secret}\) 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 \(\text{secret}\) to be a machine-sized word; i.e., 0x0000000000000000 or 0xFFFFFFFFFFFFFFFF based on the value of \(\text{secret}\). Next, for each loop iteration, Line 6 of Listing 2 computes a masked \(\delta\) based on 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 \(\text{secret}\).

```c
void ct-swap(uint64_t secret, uint64_t *a, uint64_t *b, size_t len) {
    uint64_t delta;
    uint64_t mask = ~(secret-1);
    for (size_t i = 0; i < len; i++) {
        delta = (a[i] & b[i]) & mask;
        a[i] = a[i] ^ delta;
        b[i] = b[i] ^ delta;
    }
}
```

Listing 2: Code snippet of constant-time swap. The contents of \(a\) and \(b\) is conditionally swapped based on \(\text{secret}\).

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 \(\text{secret}\). 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 \(\text{secret}\) from the victim, using microarchitectural side channels and the DMP.

**Chosen-Input Attack.** We now overview how the attacker uses the DMP to extract \(\text{secret}\). 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 \(\text{ptr}\) 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 \(\text{ptr}\) 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 \(\text{secret}\).

**Overcoming DMP Activation Criteria.** To correctly attribute the DMP’s activation to \(\text{ptr}\) 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 \(a\)\(^6\) (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 \(\text{ptr}\) was dereferenced by the DMP inside the victim process. However, it is not clear how to build eviction sets for \(\text{ptr}\)’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 \(\text{ptr}\) 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/175185154abf1a796e0f39567fe51c6e24b78d/
include/internal/constant_time.h.

---

\(^6\)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 (\(\text{len} > 1\)).

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 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 (EV\_a, EV\_ptr) of standard L2 eviction sets, where EV\_a (respectively EV\_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 (EV\_a, EV\_ptr), can evict both a and ptr’s target by priming all lines in EV\_ptr and continuously traversing EV\_a, and then probing/timing EV\_ptr. If the probe results in an L2 miss, the target of ptr filled the cache and displaced a line in EV\_ptr. This simultaneously implies that EV\_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 EV\_a or EV\_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, EV\_a only needs to be a superset of the standard eviction. We group 8\textsuperscript{10} standard eviction sets as one EV\_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 (EV\_a, EV\_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 EV\_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 EV\_ptr to monitor the DMP activation. We perform 3200 attack trials,\textsuperscript{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](image)

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

\textsuperscript{11}Based on Section 4.2, the attacker has to reset the history filter to achieve 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. We find that an attacker can craft ciphertexts to exploit this modular operation and extract a partial RSA secret key by observing DMP activations. 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 \( ed \equiv 1 \mod (p−1)(q−1) \). An RSA public key is \((N = p \cdot q, e)\). 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. 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 \( m = c^d \mod N \). Due to CRT, `Decrypt` breaks this exponentiation into two: \( m = c^{D_p} \mod p \) and \( m = c^{D_q} \mod q \), where \( D_p \) and \( D_q \) are CRT-related parameters.

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

1. if \( c < p \), \( t_0 = c \) contains the ptr and activates the DMP.
2. 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 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

```
// m = c ^ Dp mod P
m = bignod.NewNat().Exp(t0.Mod(c, P), // t0 = c mod P
// m2 = c ^ Dq mod Q
m2 = bignod.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 \( t_0 = c \mod p \) would activate the DMP if and only if \( c < p \). Attacker can then extract \( p \) adaptively by observing DMP activations.

Assuming \( p > q \), if the attacker observes no DMP activation from \( t_0 = c \mod p \), they can conclude that \( c \geq p \) and the \( n \)-th bit of \( p \) is 0 with \( 1 − \frac{1}{2^{512−n}} \approx 1 \) 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 \( \frac{1}{2^{512−n}} \) 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. 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.
### 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 standard evocation sets generation; compound evocation set finding; 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.

<table>
<thead>
<tr>
<th>Cryptography</th>
<th>Online Time (minutes)</th>
<th>Offline Time (minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA-2048</td>
<td>5 18</td>
<td>26</td>
</tr>
<tr>
<td>DH-2048</td>
<td>5 6</td>
<td>127</td>
</tr>
<tr>
<td>Kyber-512</td>
<td>6 10</td>
<td>43</td>
</tr>
<tr>
<td>Dilithium-2</td>
<td>5 13</td>
<td>577</td>
</tr>
</tbody>
</table>

---

**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 $\mathbb{Z}_p^\ast$. DHKE requires $p$ to be a safe prime such that $q = \frac{p-1}{2}$. Alice and Bob generate their own secret keys $x \in \mathbb{Z}_q$ and $y \in \mathbb{Z}_q$. Alice sends her public key $g^x \mod p$ to Bob and Bob sends his $g^y \mod p$ to Alice. They both compute the shared secret $(g^y)^x \mod p = (g^x)^y \mod p$. The security of DHKE relies on the computational Diffie–Hellman (CDH) assumption that given $g^x \mod p$, $g^y \mod p$, and $g$, it is computationally difficult to compute $g^{xy} \mod p$.

**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 $c^s \mod p$ 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 $c^s \mod p$ 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), `tmp = $c^{s_0||...||s_i}$`.

**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 $s_0||s_1||...||s_{w-2}$. To target $s_{w-1}$, the attacker makes a guess of its value and constructs $c$ by solving the equation

$$ (c^{s_0||...||s_{w-1}})^2 \mod p $$

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

Let $E$ denote the exponent $(s_0||s_1||...||s_{w-1}) \ast (2^w)$. 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:

$$ c^{E-1} \cdot \text{tmp} \mod p $$

Since $p$ is a safe prime, $\text{gcd}(p-1,E) = 1$ ($E$ 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 $(\text{tmp} \cdot R^{-1}) \mod p$ [39].

However, $E = (s_0||s_1||...||s_{w-1}) \ast (2^w)$ is an even number. An even number can be factorized as an odd number multiplied by $2^w$. 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 $\mathbb{Z}_p^\ast$ 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 `tmp \cdot R^{-1} \mod 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

---

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. 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 \( v \), \( v[i] \) denotes the \( i \)-th polynomial of \( v \), and \( v[j][i] \) denotes the \( j \)-th coefficient of \( v[i] \). For a vector \( v \in R_q^k \) (or matrix \( A \in R_q^{k \times l} \)), \( v^T \) (or \( A^T \)) denotes its transpose. \( \lfloor x \rfloor \) denotes rounding \( x \) to the closest integer, rounding up in the case of ties. \( B_{\eta} \) and \( S_{\eta} \) denote the centered binary and uniform distribution respectively. A number sampled from \( B_{\eta} \) or \( S_{\eta} \) is within the range \([-\eta, \eta]\). When we say that \( v \in R_q^k \) is sampled from \( B_{\eta} \) (\( S_{\eta} \)), we mean that each coefficient of polynomials in \( v \) is sampled from \( B_{\eta} \) (\( S_{\eta} \)). \( B_d \) denotes the set of sparse polynomials in \( R \) where \( t \) 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 \( s, e \in R_q^k \) from \( B_{\eta_1} \), with \( \eta_1 \) being a small integer. The public key consists of \( t = \text{AES} + e \). Leaking either \( s \) or \( e \) breaks Kyber.

**Kyber.CPAPKE encryption** takes in the public key \( (t, A) \), a 256-bit message \( M \), and a seed \( r \) as the source of randomness. \( M = M_0 M_1 \ldots M_{255} \) is converted to a polynomial \( m_p \in R_q \), where \( m_p(x) = \sum_{i=0}^{255} M_i \cdot \left[ \frac{q}{2} \right] + x^i \). Then, it samples \( r \in R_q^k \) from \( B_{\eta_1} \), \( e_1 \in R_q^k \) from \( B_{\eta_2} \), and \( e_2 \in R_q \) from \( B_{\eta_2} \), with \( \eta_1 \) and \( \eta_2 \) being small integers. It computes \( u = A^T r + e_1 \), and \( v = t^T r + e_2 + m_p \). The ciphertext is \((u, v)\).

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

**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 \( s \) has two \( (k = 2) \) polynomials, each with 256 coefficients. The attacker guesses a value for \( s[i][j] \) and then crafts a plaintext \( M \) containing a \( \text{ptr} \). 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 \( \text{indcpa_dec} \), the CPAKE decryption function. It decrypts the challenge ciphertext that encrypts a plaintext \( M \) containing a \( \text{ptr} \), and stores the decrypted \( M' \) into a buffer \( \text{buf} \). If the decryption is successful, \( M' = M \) and \( \text{buf} \) contains \( \text{ptr} \). Otherwise, \( M' \neq M \) and \( \text{buf} \) does not contain \( \text{ptr} \).

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 \( \text{ptr} \) is dereferenced by the DMP. This behavior is not an implementation issue but fundamental to the FO transform. As a result, subroutine \( \text{indcpa_dec} \) is DMP-vulnerable and \( \text{buf} \) is the AoP a in Section 5.

---

18 CRYSTALS: Cryptographic Suite for Algebraic Lattices https://pqcrystals.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.
We feed the recovered coefficients as hints into the lattice reduction tool. Suppose the attacker attempts to learn the first coefficient of the first polynomial in $s_0[0]$. They prepare a plaintext $M$ with a pointer in $M_{0..63}$ and fill the rest with 0s: $M = \text{ptr}[00..00]$. They manipulate other variables to ensure the following: if $0 < s_0[0]$, $(u, v)$ 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 $r = (0, 0)$ (a length-2 vector of degree-0 polynomials), $e_2 = \lfloor \frac{q}{4} \rfloor$ (a degree-0 polynomial), and $e_1 = (1, 0)$. This results in a ciphertext $(u, v, e_1)$, where $v = m_p + e_2$.

Decryption computes $v - s^Tu$:

$$v - s^Tu = m_p + e_2 - s^Te_1 = m_p + e_2 - (s[0], s[1])^T \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = m_p + \left( \frac{q}{4} \right) - s[0]$$

(4)

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

- If $\lfloor \frac{q}{4} \rfloor - s[0][0] < \lfloor \frac{q}{4} \rfloor (0 < s[0][0])$, $M' = M = \text{ptr}[00..00]$, and $buf$ activates the DMP.
- If $\lfloor \frac{q}{4} \rfloor - s[0][0] \geq \lfloor \frac{q}{4} \rfloor (0 \geq s[0][0])$, $M' \neq M$ 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 $s[0][0]$ by tuning $e_2$ and observing DMP activations. To trigger DMP activation on $buf$, we use 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 provide more details about why certain coefficients are not recoverable, and how we use the lattice reduction tool.

### Challenge ciphertext construction.

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

$$u = A^T r + e_1$$

$$v = t^T r + e_2 + m_p$$

Suppose the attacker attempts to learn the first coefficient of the first polynomial in $s$: $s[0][0]$. They prepare a plaintext $M$ with a pointer in $M_{0..63}$ and fill the rest with 0s: $M = \text{ptr}[00..00]$. They manipulate other variables to ensure the following: if $0 < s[0][0]$, $(u, v)$ 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 $r = (0, 0)$ (a length-2 vector of degree-0 polynomials), $e_2 = \lfloor \frac{q}{4} \rfloor$ (a degree-0 polynomial), and $e_1 = (1, 0)$. This results in a ciphertext $(u, v, e_1)$, where $v = m_p + e_2$.

Decryption computes $v - s^Tu$:

$$v - s^Tu = m_p + e_2 - s^Te_1 = m_p + \left( \frac{q}{4} \right) - s[0]$$

(4)

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

- If $\lfloor \frac{q}{4} \rfloor - s[0][0] < \lfloor \frac{q}{4} \rfloor (0 < s[0][0])$, $M' = M = \text{ptr}[00..00]$, and $buf$ activates the DMP.
- If $\lfloor \frac{q}{4} \rfloor - s[0][0] \geq \lfloor \frac{q}{4} \rfloor (0 \geq s[0][0])$, $M' \neq M$ 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 $s[0][0]$ by tuning $e_2$ and observing DMP activations. To trigger DMP activation on $buf$, we use 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 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 $y$ [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 $y$ that resembles a pointer. By monitoring DMP activations, an attacker could learn knowledge of $y$, 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 $s_1 \in R_q^t$ from $S_\eta$ and $s_2 \in R_q^\ell$ from $S_\eta$, with $\eta$ being a small integer. The public key consists of $t \in R_q^\ell$ and a random $A \in R_q^{\ell \times t}$, where $t = s_1 + s_2$. Leaking either $s_1$ or $s_2$ 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 $(z, c)$ of a message $M$ using the secret key $s_1$, $z$ is initialized to $\perp$ (Line 2). In a while loop, the algorithm samples a private nonce $y$, 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 $B_2^l$, and $c$ depends on $M$ (Line 5). $c$ is a sparse polynomial with exactly $t$ non-zero entries, and the non-zero entries have randomized positions. The algorithm computes $z = y + cs_1$ (Line 6), but will only accept $(z, c)$ if it leaks no secrets, and reject $(z, c)$ otherwise. Note that $y$ must be kept private. Leaking $y$ leaks $s_1 = \frac{z-y}{c}$. Leaking partial information of $y$ might also compromise $s_1$ [18].

**PoC overview.** The victim (a Dilithium signing server) generates a pair of Dilithium secret and public keys. Our threat
Algorithm 1: The main body of the Dilithium signature algorithm is a while loop that creates a signature \((z,c)\) of message \(M\) under the secret key \(s_1\). The algorithm returns \((z,c)\) if it does not leak any secret information.

1 \(\text{Sign}(s_1,M)\)
2 \(z := \bot\)
3 while \(z = \bot\) do
4 \(y \leftarrow \mathcal{S}_n^t\) // A length-1 vector of random and small polynomials
5 \(c \in \mathcal{B}_t\) // A sparse polynomial \((\text{depending on } M)\) with \(\tau\) number of 1 or \(-1\)
6 \(z := y + cs_1\)
// Reject \(z\) (set \(z\) to \(\bot\)) if \(z\) leaks information about the secret key
7 \(\text{return } (z,c)\)
8 end

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]. Dilithium is deterministic when the private nonce \(y\) 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 \(y\). While the exact value of \(y\) should remain secret, the underlying MLWE structure allows an attacker to approximate \(y\) through \(z\). If a ptr appears in \(z\), the attacker infers its presence within \(y\) and confirms this by observing DMP activations. Successful confirmation reveals partial knowledge of \(s_1\).

Our PoC consists of an offline and online signature collection phase. During the offline phase, the attacker sends \(m\) random messages to the server requesting signatures and collects \(m'\) pairs of \(\{(z,c),M\}\), where \(z\) 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 \(\{(z,c),M\}\) causes the ptr to show up in \(y\) via DMP activations, and then derive a linear equation of \(s_1\). The attacker further uses the lattice reduction tool to recover \(s_1\) [59].

DMP-vulnerable subroutine in CIRCL Dilithium. The DMP-vulnerable subroutine is the \(z = y + cs_1\) in Algorithm 1. CIRCL uses an array of unsigned 32-bit integers to represent a polynomial. Every coefficient of \(y\) and \(z\) is stored as an unsigned 32-bit integer. We pick \(y\) as the AoP \(a\) from Section 5. The range of coefficients in \(y\) is \([-2^{17}, 2^{17}]\), and that of coefficients in \(cs_1\) is \([-78, 78]\).

Let’s take the first two 32-bit coefficients of \(y\) \((y[0][0], y[0][1])\) and \(z\) \((z[0][0], z[0][1])\) as an example. Assume that \(z[0][1] \parallel z[0][0]\) forms a valid 64-bit ptr, pointing to the same 4GBYTE region where \(y\) lives. If we break ptr into two 32-bit halves (\(ptr_1 \parallel ptr_0\), then \(z[0][1] = ptr_1\) and \(z[0][0] = ptr_0\).

We can derive the range of \((y[0][0], y[0][1])\):

\[
y[0][1] = z[0][1] - cs[1][0][1] \in [ptr_1 - 78, ptr_1 + 78]
y[0][0] = z[0][0] - cs[1][0][0] \in [ptr_0 - 78, ptr_0 + 78]
\]

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

To elaborate, we know \(z[0][1] \parallel z[0][0]\) forms a ptr. If we want \(y[0][1] \parallel y[0][0]\) to also form a ptr, we only need \(y[0][1] = z[0][1]\) or \(cs[1][0][1] = 0\). The value of \(y[0][0]\) is less important because \(cs[1][0][0]\) is small, variations in which will only cause \(y[0][1] \parallel y[0][0]\) to map to the same or an adjacent cache line as ptr. As a result, \(z = y + cs_1\) is DMP-vulnerable. If the attacker sets \(y\) as the AoP \(a\) from Section 5, they can learn \(cs[1][0][1] = 0\) by observing DMP activations. The same idea applies to all other coefficients of \(y\) and \(z\).

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

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 \(y[i][j+1] = z[i][j+1]\), and derive one linear equation of \(s_1\): \(cs[1][j][j+1] = 0\). After gathering at least 876 linear equations, the attacker uses the lattice reduction tool to recover \(s_1\) [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 \(m' = 3 \times 10^8\) ones with the property that \(z[i][j+1] \parallel z[i][j]\) forms a ptr. In the online phase, resubmitting the \(m'\) messages, we observe a Prime+Probe latency of \(772 \pm 152\) ticks when the message triggers the DMP, compared to \(657 \pm 106\) 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.
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.

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


We now briefly present the method to generate eviction sets whose eviction set is not included in the current group. With a fixed page offset, which map to L2 sets differed by dyld 4GByte region as the valid (and quiet) pages to search for.

Dilithium significantly increases the compound eviction set ptr injects but has a pool of messages from which a subset correctly attacker cannot confirm m message ptr 512 (reference), first iteration by solving Equation (2) with $E$ is always 1. Hence, the attacker can inject the ciphertext is smaller than $p$, 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 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 $s_i$ 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.

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 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 $s_i$ 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 $EV_a$ 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 $EV_a$. One solution is to decrease the size of $EV_a$ 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].

### 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 ($EV_a$, $EV_{ptr}$), 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 `vmmmap`. RSA-2048 (Go) and deterministic Dilithium-2 (CIRCL) allocate $a$ in a stable address region beyond `0x1400000000`. 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 $EV_a$ and $EV_{ptr}$ by first targeting $sig.z$, where the attacker can confirm $ptr$ injection. Note that the compound eviction set to $sig.z$ shares the same $EV_{ptr}$ with that of $y$, thus having the right $EV_{ptr}$ makes generating $EV_a$ 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 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 $s_i$ 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 $EV_a$ 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 $EV_a$. One solution is to decrease the size of $EV_a$ 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].