

SEC DMP: Provably Secure Data Memory-Dependent Prefetching

Jens Sprengers

DistriNet, KU Leuven

jens.sprengers@kuleuven.be

Marton Bogнар

DistriNet, KU Leuven

marton.bognar@kuleuven.be

Lesly-Ann Daniel

EURECOM

lesly-ann.daniel@eurecom.fr

Frank Piessens

DistriNet, KU Leuven

frank.piessens@kuleuven.be

Abstract—Data memory-dependent prefetchers (DMPs) are a microarchitectural optimization recently introduced in mainstream desktop and server CPUs. Although DMPs offer performance improvements, they have been shown to enable side-channel leakage of secrets, even from constant-time code that has been hardened against Spectre attacks. Unfortunately, existing defenses either incur significant performance overhead or, as we show, fail to prevent all DMP leakage.

In this work, we address this gap by designing a defense that is both secure and efficient. First, we propose a classification of DMP leakage, categorize existing and novel attacks, and analyze which leakage classes are mitigated by existing defenses. We validate this analysis empirically by implementing the first open-source DMPs on a RISC-V processor, showing that the only existing defense that aims to preserve DMP performance fails to protect against entire leakage classes. Next, we propose SecDMP, a provably secure and efficient hardware-based defense. SecDMP requires secrets to be marked in memory and ensures that DMPs never encode these secrets in the microarchitecture—without compromising performance for public data. We formally prove that SecDMP introduces no leakage beyond a baseline semantics without DMPs. Notably, when combined with speculative-execution defenses, SecDMP restores complete microarchitectural security for constant-time code. Finally, our empirical evaluation shows that SecDMP prevents all DMP leakage while incurring minimal performance overhead and hardware cost.

I. INTRODUCTION

Microarchitectural side channels pose a significant challenge to secure software. Since the first timing attack [40], a growing body of work has shown that many microarchitectural components and optimizations leak sensitive data. Well-known leakage sources include instruction [1] and data caches [10], port contention [2], branch predictor state [72], and (classical) hardware prefetchers [12]. Additionally, modern CPUs employ speculative execution induced by mechanisms such as branch prediction and memory disambiguation, which Spectre attacks [32], [39] exploit to leak secrets during transient execution caused by mispredictions.

To defend against microarchitectural side-channel attacks, the most commonly adopted approach is to implement software in a constant-time fashion [3], [4], [11], [49]. This

approach requires that secret data is never used as operands of computations that may produce operand-dependent observable microarchitectural effects. Specifically, secrets must not be used as branch conditions, memory addresses, or operands of variable-time instructions (e.g., divisions). While constant-time programming itself does not protect against Spectre, defenses have been proposed in both software and hardware that secure constant-time code against these attacks [16], [20].

Data memory-dependent prefetchers (DMPs). Recent work has shown that new generations of Apple, Intel, and Arm CPUs implement a novel type of prefetcher, known as a data memory-dependent prefetcher (DMP) [6], [17], [64], [65]. Unlike classical prefetchers, which base their predictions on the history of accessed memory addresses, DMPs also consider the *loaded data* itself. This has severe security implications, as DMPs can encode secrets into microarchitectural state without the program ever explicitly (not even transiently) performing this encoding. For instance, GoFetch [17] showed that the DMP on modern Apple processors can leak secrets from constant-time cryptographic implementations. Mitigating this type of leakage is particularly challenging, as it does not rely on any specific property of the victim program aside from containing sensitive data in memory. The most practical proposed mitigations are to disable the DMP, either completely [17], [64], [65] or selectively [17], [54].

This paper. In this work, we propose a systematic and formal approach to analyzing and mitigating DMP leakage. We start by classifying how DMPs access and leak secret data. Our main insight is that DMPs can be activated not only on data *architecturally* loaded by the victim program, but also *indirectly* via (1) adjacent data within a cache block, (2) data loaded transiently, and (3) data loaded by prefetchers (either the DMP itself or other prefetchers). We analyze which commercial DMPs are prone to each leakage class based on prior reverse-engineering efforts, report which classes have been demonstrated in prior attacks, and identify the remaining unknown design points that could enable new attacks. Importantly, we show that selective disabling [54] does not mitigate all leakage classes, making it an error-prone defense whose security is hard to reason about. In addition to this informal analysis, we formally model existing DMP designs as extensions of a standard speculative-execution semantics.

SecDMP. To overcome the limitations of existing defenses, we propose SEC DMP, a new hardware-based defense mech-

anism for securing DMPs. SECDMP requires that secrets are marked in memory—a technique employed in many Spectre defenses [20], [26], [55], [71]—and prevents prefetches and state updates that are based on secret data. Using our formal model, we prove that SECDMP prevents all secret leakage through the DMP. Specifically, when combined with a baseline speculative semantics without a DMP, SECDMP *does not introduce additional leakage* beyond the baseline. In particular, when combined with a strong Spectre defense such as ProSpecT [20], SECDMP restores full microarchitectural security for constant-time cryptographic code.

On the practical side, we implement several DMPs modeled after commercial designs [17], [64], [65] and academic proposals [19], [73] on the Proteus RISC-V CPU [14]. Using our implementations, we demonstrate all DMP leakage classes through proof-of-concept (PoC) programs, based on existing and novel attacks. To the best of our knowledge, these are the first open-source DMP implementations and PoC DMP attacks on RISC-V. To assess SECDMP’s cost and effectiveness, we evaluate two configurations: SECDMP on a speculative out-of-order baseline, which only prevents leakage through the DMP; and SECDMP combined with ProSpecT [20], restoring full microarchitectural security for constant-time code. Results show that SECDMP incurs minimal hardware cost and performance overhead: it increases chip area by 0.5 % without affecting the critical path, and has an average performance overhead of 0.36 % on our benchmarks with a maximum of 3.3 %. Furthermore, we experimentally confirm that SECDMP eliminates all secret leakage across our benchmark programs.

Contributions. In summary, our contributions are:

- We classify how DMPs leak secrets (Section IV). Our classification reveals that current mitigations are insufficient or expensive, which we experimentally validate.
- We present a modular formal model of DMPs that we instantiate with existing designs (Section VI).
- We propose SECDMP, a hardware-based defense mechanism that prevents DMPs from leaking secrets (Section V), with a formal proof of security (Section VII).
- We implement the first open-source DMPs on a research processor, and implement and evaluate SECDMP and its combination with ProSpecT. Our results show that SECDMP is effective, incurring low hardware cost and minimal performance overhead (Section VIII).

Availability. All artifacts are available at <https://github.com/JensTheGrens/secdmp>. An extended version of this paper with the full formalization and security proofs is available at [61].

II. BACKGROUND

A. Microarchitectural Side-Channel Attacks

Microarchitectural side-channel attacks expose a victim’s secrets by observing secret-dependent effects on hardware components shared between attacker and victim, such as caches, branch predictors, and prefetchers [27], [62], [74]. Notably, a wide range of attacks expose a victim’s memory usage by (partially) leaking accessed memory ad-

dresses, which might depend on secret data. Prominent examples include cache attacks such as Flush+Reload [69] and Prime+Probe [44], address-translation attacks targeting TLBs and page walks [28], [29], [63], [65], and contention-based attacks on shared interconnects [45]. While most of these attacks leak information within a single CPU and at cache-block granularity, prior work has demonstrated leakage across CPUs [36], [47] and even recovery of fine-grained address bits [42], [57], [70].

1) *Constant-time programming*: Constant-time is a programming discipline addressing the root cause of side-channel leakage by ensuring that secrets do not influence observable microarchitectural behavior [4], and has become the standard practice for securing cryptographic software [3], [11], [49]. Concretely, it requires memory accesses, control flow, and variable-latency instruction operands to be secret-independent.

2) *Spectre*: Speculative execution improves performance by executing instructions ahead of time based on predictions. Modern processors include various speculation mechanisms [15], [68], such as conditional and indirect branch prediction [39], memory disambiguation [32], load value prediction [37], and load address prediction [38]. Instructions executed based on incorrect predictions, known as *transient* instructions, are architecturally reverted to preserve functional correctness. However, their microarchitectural effects remain and can be observed through side-channel attacks. Spectre attacks [39] exploit this by deliberately causing a victim program to mispredict and transiently execute instructions that produce secret-dependent microarchitectural effects.

3) *Spectre defenses*: Transient execution attacks showed that classical constant-time programming is insufficient: even architecturally constant-time programs may leak secrets via misspeculated execution [16]. As a result, the constant-time discipline has been extended to *speculative constant-time* [16], which additionally requires the protection of secrets during transient execution. However, enforcing speculative constant-time purely in software is extremely challenging. Consequently, a lot of research has focused on hardware-based defenses against leakage during speculation [33]. Among these, a particularly promising class relies on explicitly informing the hardware which data should be protected [20], [26], [55], [71]. This enables the hardware to restrict speculation on secrets while preserving full performance for public data.

B. Hardware Prefetchers

Prefetching is an optimization technique to reduce memory latency. Hardware prefetchers predict which addresses will be accessed by a program in the near future and bring the corresponding data into the cache before they are requested by the program. If the prediction is correct, prefetching increases performance as it avoids the penalty of a cache miss.

Hardware prefetchers can be categorized into two groups: *classical prefetchers* and *data memory-dependent prefetchers* (DMPs) [17], [52], [64]. Classical prefetchers are widely implemented on commercial processors and make predictions by monitoring a program’s accessed memory addresses. For

instance, a *classical stride prefetcher* detects array traversals with a constant stride, prefetching elements ahead in the array [64]. DMPs, however, have only recently been implemented on commercial processors following years of academic proposals [17], [64]. DMPs base their predictions not only on the accessed memory addresses but also on the *data itself* stored in memory. This enables them to identify irregular access patterns for which classical prefetchers fail [52]. For example, they can speed up pointer dereferences by scanning memory for pointers and prefetching their targets. In 2022, Vicarte et al. [64] demonstrated for the first time the existence of a DMP on a commercial processor, namely the Apple M1. As of now, several Apple [17], [53], [64], Intel [65], and Arm [6] processors are known to include a DMP.

Next, we identify three key mechanisms in existing DMP designs based on the literature [19], [73] and reverse-engineering [17], [53], [64], [65]. We discuss concrete examples of commercial and academic designs in Section IV-B.

1) *Address heuristics*: DMPs read data to derive target addresses to prefetch, optionally applying a heuristic to filter out invalid targets. For example, *pointer-chasing DMPs* scan cache blocks for values that resemble valid pointers and prefetch the corresponding addresses [64], [65], relying on *pointer heuristics* to distinguish pointers from other data.

2) *Recursion and chain prefetching*: DMPs typically read data loaded directly by the program [19], [64], [65], [73], but they may also read data loaded by a *prefetcher*. First, DMPs can scan data that they prefetched themselves, which is known as *recursive prefetching* [19]. For instance, such DMPs can recursively prefetch entire linked data structures ahead of time. Second, DMPs can also scan data prefetched by other prefetchers [17]; we call this *chain prefetching*.

3) *Activation patterns*: Some DMPs only activate after detecting a specific memory access pattern [65], [73]. They first train on the program’s memory accesses to learn a data-dependent pattern, then prefetch ahead in the learned pattern. These DMPs are often recursive, as data-dependent patterns require reading data ahead in the pattern to derive the next prefetch target. For example, the Intel DMP can traverse an array of pointers [65]: it first prefetches ahead in the array, then reads the returned pointer and prefetches its target address.

C. Security Risks of DMPs

The effects of prefetchers on microarchitectural state match those of load requests: prefetching a (virtual) address first triggers an address translation, and, if successful, fetches the data into the cache. As a result, any microarchitectural attack that can detect loaded addresses can be used to infer prefetch targets, including standard cache attacks. Additionally, some attacks exploit prefetchers’ *internal states* directly [18], [31]. As such, any secret data that influences the behavior of a prefetcher (i.e., the information analyzed to determine its internal state and prefetch targets) can potentially leak.

Vicarte et al. [52] were the first to warn about the security risks of DMPs, showing that the academic IMP DMP [73] can serve as a universal read gadget through out-of-bounds

prefetching. Later, Augury [64] identified the first commercial DMP on Apple processors and showed that it can leak data beyond the bounds of an array of pointers. Barthe et al. formalized this leakage behavior in LmSpec [9], detecting leakage in constant-time cryptographic implementations under this model. Subsequently, GoFetch [17] demonstrated how the Apple DMP can be exploited in practice to leak secret keys from constant-time code, performing attacks on several cryptographic primitives. Because Apple’s pointer heuristic is fairly strict, the GoFetch attacks are limited to specific scenarios where the attacker forces a pointer to appear in the victim’s memory only for some variants of the secret. Peek-a-Walk [65] showed that an attacker can deliberately mistrain the Intel DMP so that it activates during the victim’s execution and leaks data at an address of the attacker’s choosing within certain constraints. Their attack demonstrates that DMPs with less restrictive pointer heuristics have more potential to expose secrets. In this case, the impact is limited, as it is not possible to train the Intel DMP across processes [65].

To address the identified leakage caused by DMPs, a number of countermeasures were proposed in the literature. We discuss and analyze these in Section IV-D.

III. THREAT MODEL

In line with related work, we assume an attacker who can monitor all microarchitectural side channels introduced in Section II-A, including those induced by speculative execution. The attacker is abstract and may operate from an intra-process sandbox [65], a sibling SMT context, or another processor core. Our goal is to protect against side-channel leakage introduced by DMPs—specifically, we care about leakage relative to a baseline with DMPs disabled. In Section VII-C, we additionally show that full microarchitectural security is achieved when complementary countermeasures are in place for other leakage, either in software (speculative constant-time [16]) or in hardware (secure speculation defenses [20]).

IV. CLASSIFICATION OF DMP LEAKAGE

To understand the different ways in which DMPs may leak secret data, we propose a leakage classification based on existing DMP designs. We focus our analysis on designs described in the literature, consisting of the commercial designs of the Apple [17], [53], [64] and Intel [65] DMPs as reverse-engineered by prior work, and two academic designs [19], [73] with unique properties. First, we present our classification in Section IV-A and describe the existing DMP designs and how they fit into the leakage classes in Section IV-B. Next, we categorize existing DMP attacks into our leakage classes, devise new attack techniques, and demonstrate the leakage and attacks in practice (Section IV-C). For the demonstration, we implement several DMPs on a RISC-V processor. Finally, we describe existing defenses (Section IV-D) and provide an informal analysis and empirical validation of which leakage classes they mitigate (Section IV-E). Our results show that the most promising defense, selective disabling, does not mitigate all leakage, highlighting the need for secure DMP defenses.

TABLE I
LEAKAGE CLASSES APPLICABLE TO DIFFERENT DMP DESIGNS,
EXPLICITLY DOCUMENTED (●) OR PRESUMED (○).

DMP	Arch	Ind		
		Sub	Trans	Pref
Apple DMP [17], [64]	●	●	●	●
Intel DMP [65]	●		○	●
CDP [19]	●	●	○	●
IMP [73]	●		○	●

A. DMP Leakage Classes

All attacks that exploit DMPs involve at least three steps: (1) the DMP scans secret data, (2) the data is encoded into the microarchitecture, and (3) the attacker uses existing side-channel attacks to recover it [64]. As discussed previously, DMPs have the potential to leak any data they scan in step (1), making this the crucial step to reason about DMP leakage. We therefore classify DMP leakage based on the sources of data that the DMP scans for both training and prefetching.

1) *Architecturally accessed data (Arch)*: The *Arch* class represents all data that is architecturally accessed (via a non-transient instruction) by the program. This is the baseline source of data for any DMP: when the program architecturally loads data from memory, the DMP may scan it to derive prefetch targets, potentially leaking the data.

2) *Indirectly accessed data (Ind)*: In addition to *Arch*, DMPs can also scan and leak data that the program never architecturally accesses, divided into three subclasses.

a) *Sub-cache-block data (Sub)*: When accessing memory, the processor typically loads an entire cache block, e.g., a 64-byte cache block for a 4-byte load instruction. The data adjacent to the architecturally accessed value in the cache block may also be scanned by DMPs.

b) *Transient loads (Trans)*: Due to incorrect speculative execution, the processor may issue load requests that would never occur during architectural execution. Some DMPs may read the data accessed by such transient loads. We expect that no real-world DMP distinguishes between transient and non-transient loads, and thus leaks through the *Trans* class. Distinguishing them would require complex hardware and reduce performance, as at the time a speculative load is executed, it is not yet known whether it will be squashed.

c) *Prefetched data (Pref)*: Finally, DMPs can activate on data without any load instruction (architectural or transient) explicitly requesting it. This happens when the data was accessed through prefetching. There are two ways in which a DMP can scan data belonging to the *Pref* class: recursion and chain prefetching; reading data prefetched by itself or by another prefetcher, respectively.

B. DMP Designs and Classification

We now describe the four DMP designs of interest and categorize their leakage into the classes above. The results are summarized in Table I.

1) *Apple DMP*: The Apple DMP [17] is a pointer-chasing DMP. When a data block is loaded into the L1 cache, the DMP applies a pointer heuristic to every 64-bit value in the block, referred to as candidate pointers. The DMP scans all blocks loaded into the L1 cache for candidate pointers, including from transient loads, hence leaking through the *Arch*, *Sub*, and *Trans* classes. A prefetch request is then initiated for each candidate pointer that passes the heuristic, which checks whether bits 55–32 of the candidate pointer match those of the current block address. While GoFetch [17] does not report any recursive prefetching behavior, it does report the DMP triggering a next-line prefetcher, fetching two adjacent blocks.

GoFetch additionally observes behavior that is not explained by its model: looping through an array of random values triggers the DMP to dereference pointers located after the array, demonstrating *Pref* leakage. While the exact mechanism responsible for this leakage is unknown, we believe it to be chain prefetching, where a stride prefetcher first fetches data beyond the array, which the DMP then scans for pointers.

2) *Intel DMP*: The Intel DMP [65] is a pointer-chasing DMP requiring a specific activation pattern: accessing and dereferencing the pointers inside an array of pointers (AoP). During its training phase, the DMP learns the stride of array accesses issued in a loop (detected by storing and comparing the 10 lowest bits of the program counter (PC) of the load instructions), and checks whether the accessed array entries are dereferenced. The DMP thus analyzes loaded addresses and values, leaking through the *Arch* class. Whether the Intel DMP also reads transient loads is unknown, but we presume *Trans* leakage to be present (cf. Section IV-A).

When the DMP detects the AoP pattern and learns the stride k , it activates whenever it encounters a load instruction with the matching 10 lowest PC bits: it prefetches $k * 16$ entries ahead in the array, applies a pointer heuristic to the value when the prefetch returns, and dereferences the value if the heuristic succeeds. This recursive behavior thus enables leakage through the *Pref* class: the DMP can read data it prefetched.

Finally, the Intel DMP limits the stride by requiring the current entry and the entry to prefetch to be located in the same 256 KiB aligned memory region, and its pointer heuristic restricts the prefetching to user memory.

3) *CDP*: The content-directed data prefetcher (CDP) [19], predating commercial designs, is a recursive variant of the Apple DMP: it scans cache blocks for pointers from both load and prefetch requests, leaking through the *Arch*, *Sub*, and *Pref* classes. CDP’s design does not explicitly mention transient loads; however, since the DMP monitors all memory traffic for a specific cache level, it would also scan transient loads, capturing the *Trans* class. CDP applies a pointer heuristic to each candidate pointer of the scanned cache block, checking whether a configurable number of upper bits of the candidate pointer and the address at which it is stored match. CDP also features a configurable number of next lines to prefetch alongside any target address that passes the heuristic. To prevent recursive prefetching from continuing indefinitely, it has a configurable maximum recursion depth.

TABLE II
LEAKAGE CLASSES TARGETED BY EACH OF OUR TEST PROGRAMS.
PROGRAMS ADAPTED FROM EXISTING ATTACKS ARE HIGHLIGHTED.

	Program	Arch	Ind		
			Sub	Trans	Pref
Attacks	IMP Leak [52]				●
	Augury PoC [64]				●
	GoFetch PoC [17]	●			
	Peek-a-Walk [65]				●
	DMP-Spectre Recursion			●	●
General	Load Secret	●			
	Same Cache Block		●		
	Transient Load			●	

4) *IMP*: The indirect memory prefetcher (IMP) [73] monitors memory traffic for the pattern $A[B[i]]$, i.e., indexing one array with values from another. After detecting this pattern, IMP prefetches and reads values ahead in the first array, adds them to the base of the second array, and prefetches the resulting address. This recursive behavior enables leakage through the *Arch* and *Pref* classes. IMP’s design does not mention transient loads, but when implemented as described, it would not distinguish them from non-transient loads, leaking through the *Trans* class. In addition to the 2-level variant, the authors proposed a 3-level variant $A[B[C[i]]]$. Notably, IMP does not use a pointer heuristic to filter prefetch targets.

C. Exploiting Leakage Classes

We now illustrate how programs can leak secrets through each leakage class, classifying existing DMP attacks and devising new techniques. We demonstrate the leakage classes and attacks in practice by crafting simple test programs for the Proteus RISC-V processor [14]. To enable the demonstration, we extended Proteus with the four DMPs from Section IV-B, i.e., the Apple DMP [17], [64], Intel DMP [65], CDP [19] and IMP [73], modeling them as faithfully as possible after the descriptions in the literature. The test programs and results of the leakage demonstration are listed in Table II. For more details regarding the implementation and evaluation methodology, we refer to Section VIII.

1) *Classification of existing attacks*: We revisit the attacks from Section II-C and classify them according to our leakage classes. Vicarte et al. [52] showed that the 3-level IMP [73] can serve as a universal read gadget through out-of-bounds prefetching, hence leaking through the *Pref* class. Augury [64] showed that pointer-chasing DMPs that look for an AoP pattern can leak secrets located after the array’s bounds, also falling in the *Pref* class. These two cases highlight the dangers of the recursive behavior of pattern-activated DMPs: when they prefetch too far ahead in the pattern, they may access data out-of-bounds and use it to derive further prefetch targets, thereby leaking it. In GoFetch’s attacks [17], the DMP leaks whether an architecturally loaded secret resembles a valid pointer, leaking through *Arch*. In the Peek-a-Walk DMP attack [65], an attacker deliberately mistrains the Intel

DMP into activating during the victim’s runtime, causing it to load, scan, and leak a secret located at an attacker-influenced address, i.e., *Pref* leakage.

We demonstrate these attacks via the following programs: IMP Leak demonstrates a 2-level IMP attack, Augury PoC shows the out-of-bounds leakage of AoP DMPs, GoFetch PoC replicates GoFetch’s proof-of-concept attack on OpenSSL’s constant-time conditional swap, and Peek-a-Walk mistrains the Intel DMP into activating and leaking a secret.

2) *Novel attacks exploiting leakage classes*: We now devise novel attacks that rely on speculative execution and recursive prefetching, leaking through *Trans* and *Pref*, respectively.

a) *DMP-Spectre*: Attackers can exploit DMPs that are capable of leaking through the *Trans* class (see Table I) in a manner similar to Spectre attacks. While typical Spectre attacks require a gadget that transiently loads a secret and subsequently leaks it through a follow-up instruction [68], DMPs can replace this follow-up instruction. For example, by deliberately mistraining prediction mechanisms, the attacker can induce the victim to transiently load a secret, which the DMP can then scan and leak. We demonstrate the potential of these attacks with the DMP-Spectre program, which loops through a public array and triggers a transient load of a secret located beyond the array bounds due to branch misprediction.

b) *Recursion and chain prefetching*: In the case of a recursive pointer-chasing DMP, an attacker can embed the address of a secret in user input. The victim will then load this input, causing the DMP to interpret it as a pointer and fetch the secret data. Because the DMP is recursive, it reads the prefetched secret, potentially leaking it. This attack vector was also mentioned but not exploited by Schlüter et al. [54]. We demonstrate it for CDP via the Recursion test program. For chain prefetching, the attacker can mistrain the first prefetcher into loading a secret during the victim’s execution, causing the chained DMP to activate and read the secret.

3) *General tests*: Beyond the attacks described above, we run three general leakage tests for the *Arch*, *Sub*, and *Trans* classes. Unlike the attacks, these tests are not tailored to specific DMPs. Load Secret simply loads a secret, the simplest example of architectural leakage. Same Cache Block accesses public data sharing a cache block with a secret, demonstrating the *Sub* class. While our test deliberately co-locates secret and public data, this can occur naturally in real programs, e.g., when compilers reuse stack space by overwriting unused secrets with public data [7]. Finally, Transient Load shows *Trans* leakage by transiently loading a secret due to misspeculation.

D. Proposed DMP Mitigations

Given our new classification of DMP leakage, we now apply the same concept to mitigations. First, we briefly introduce defense proposals before classifying them in Section IV-E.

1) *Permanent disabling*: The simplest, often proposed defense is disabling the DMP for the entire program’s execution [17], [64], [65], e.g., via software-accessible DIT/DOIT bits [5], [35]. While permanent disabling addresses the vulnerabilities of DMPs, it imposes a potentially heavy performance

TABLE III
LEAKAGE CLASSES MITIGATED BY DIFFERENT DMP DEFENSES.

Defense	Arch	Ind		
		Sub	Trans	Pref
No Defense	✗	✗	✗	✗
Permanent Disabling	✓	✓	✓	✓
Selective Disabling	✓	✗	✗	✗
SECDMP	✓	✓	✓	✓

penalty [17], [54], [64]. For example, a program with only a small security-critical code region will lose all the performance benefits of DMPs, including when operating on public data.

2) *Selective disabling*: To address the performance cost of permanent disabling, some papers propose to selectively disable DMPs only when computing on secret data [17], [54]. Apple and Intel similarly recommend selectively setting the DIT and DOIT bits only during security-critical operations [5], [35]. Recently, Schlüter et al. implemented selective disabling of all prefetchers in PreFence [54], reporting a significant performance improvement over permanent disabling.

3) *Preventing secret extraction*: Some defenses attempt to prevent attackers from extracting secrets from the microarchitecture. For instance, PhantomFetch [75] flushes the prefetcher’s state on context switches. However, as prefetching affects many microarchitectural components, it is impractical to secure all leakage channels without incurring high overhead. Additionally, this defense is likely insecure in intra-process sandbox settings or when the attacker can continuously monitor the victim from an SMT thread or another core.

4) *Interleaving*: Pättschke et al. [51] proposed interleaving secret data with specific values in memory as a theoretical defense to prevent secrets from activating the DMP pointer heuristic. This defense only applies to DMPs with a pointer heuristic and must be adapted to each DMP’s specific heuristic [17], [19], [65]. Pättschke et al. also note the high performance cost and instead suggest using DMP disabling, which Sharma et al. [56] confirmed: their implementation of interleaving incurred a performance overhead of 274 %.

E. Classification of DMP Mitigations

Since flushing the prefetcher state is insecure in our threat model and interleaving is less secure and has much higher overhead than permanent disabling [51], [56], we focus on permanent and selective disabling. The theoretical analysis of this section is complemented by an experimental validation and performance measurements on Proteus [14] in Section VIII-B.

Trivially, with no defense, no leakage class is mitigated. Conversely, with permanent disabling, DMPs cannot scan any data, mitigating all their leakage. However, all performance improvements of the DMPs are lost.

Selective disabling [17], [54] consists of disabling the DMP during security-critical code regions (i.e., regions where the program architecturally accesses secret data), effectively preventing leakage of the *Arch* class. However, leakage of the *Ind* classes can still occur in public code regions where DMPs

are enabled. Schlüter et al. [54] acknowledge that selective disabling does not prevent leakage for recursive DMPs (*Pref*), and advise to disable DMPs when looping through public data if secret data is located after it. However, this additional disabling only prevents accidental leakage of the *Pref* class caused by out-of-bounds prefetching, without addressing other types of indirect leakage. Attackers can still deliberately mistrain predictors to trigger *Trans* and *Pref* leakage during public code regions, and *Sub* leakage can occur in non-trivial ways when public and secret data are mixed, e.g., on the stack. In conclusion, DMPs cannot be safely enabled in public code regions while secret data is accessible in memory.

This highlights a critical gap in DMP defenses: users either need to sacrifice security or suffer performance degradation from permanent DMP disabling.

V. OVERVIEW OF SECDMP

To overcome the inefficiency of permanent disabling and the insecurity of selective disabling, we propose SECDMP, a general defense strategy for securing DMP designs against data leakage. SECDMP relies on a hardware-software co-design: on the software side, developers mark all data that is secret, while on the hardware side, the DMP is prevented from using secret data for training or prefetching.

Importantly, SECDMP prevents all secret leakage through the DMP while imposing minimal performance overhead: it prevents only prefetches that depend on secret data without hindering any prefetches based on public data.

A. Software Changes

On the software side, application data must be partitioned into public and secret data. SECDMP can be combined with different methods for partitioning [20], [26], [55], [71], as long as secret data is correctly labeled.

Our security proof relies on a static memory partition, which we implement in Section VIII-A using coarse-grained static secret regions, similar to prior academic defenses [20], [26], [55], [71] and industry patents by Arm [24] and NVIDIA [13]. While coarse-grained static partitioning introduces several challenges, recent work has shown that it can be practical [60]; we discuss these challenges and solutions in Section IX-A.

B. Hardware Changes

On the hardware side, SECDMP ensures that data labeled as secret is never encoded in the DMP state or used for initiating prefetch requests. Concretely, SECDMP requires only two hardware modifications. The first is the aforementioned mechanism for identifying secret data in memory by associating loaded data with a taint bit. Second, the DMP itself must be modified to recognize this taint bit and ensure that it ignores tagged data. Importantly, these changes are local and lightweight. Unlike existing Spectre defenses based on taint tracking [20], [26], [55], [71], SECDMP itself does not require secrets to be tracked throughout the core, making adoption more realistic in complex CPU designs—though SECDMP can optionally be paired with dynamic secret-tracking schemes as an alternative to static partitioning.

C. Security of SecDMP

SECDMP closes all leakage channels introduced by DMPs for data marked as secret. We formally prove this property in [Section VII \(Theorems 1 and 2\)](#) for a static secret partitioning: SECDMP provides the same security guarantees as a baseline configuration without a DMP.

Programs may still leak secrets through other microarchitectural channels, such as standard side-channel attacks (e.g., exploiting caches or port contention) and Spectre attacks. To achieve comprehensive microarchitectural security, SECDMP must therefore be combined with existing defenses against such attacks. If a program does not leak secrets in the baseline configuration without a DMP, it will also not leak secrets when the baseline is extended with a SECDMP-secured DMP.

Importantly, even cryptographic programs that follow the strict speculative constant-time discipline, and are thus secure against standard side-channel and Spectre attacks, can remain vulnerable to DMP-based attacks. SECDMP closes this additional leakage channel and restores the security guarantees offered by speculative constant-time defenses. For instance, SECDMP can be combined with existing hardware-based [20], [21], [26], [55], [71] or software-based [8], [77] Spectre defenses; we show this formally in [Section VII-C](#).

VI. FORMALIZATION OF DMPs AND SECDMP

To formally reason about DMP security, we propose a formal semantics capturing DMP behavior. We start by introducing a baseline out-of-order semantics ($\{\cdot\}^{\text{BASE}}$, [Section VI-A](#)), which we extend to model DMP behavior ($\{\cdot\}^{\text{DMP}}$, [Section VI-B](#)). Finally, we formalize SECDMP as an extension of an arbitrary base semantics ($\{\cdot\}^{\text{B+SECDMP}}$, [Section VI-C](#)). We keep the formalization high-level; the full formalization can be found in the extended version [61].

A. Baseline Speculative Semantics ($\{\cdot\}^{\text{BASE}}$)

1) *Language*: The language in [Figure 1](#) defines our ISA. It contains instructions for moving data between registers and memory, as well as jumps and conditional branches. Direct assignments to the program counter are not allowed.

2) *Configurations*: A hardware configuration c is a tuple $\langle m, r, \beta, \mu \rangle$ where $m : \mathcal{V} \rightarrow \mathcal{V}$ is a memory mapping addresses to values; $r : \mathcal{R} \rightarrow \mathcal{V}$ is a register map of registers to values; β is a reorder buffer (ROB), a sequence of (possibly transient) instructions; and μ is the *microarchitectural context*. Similar to existing formal semantics [20], we assume that μ is the *observable part of the microarchitectural state*: attackers can observe μ at each execution step. Finally, we also refer to $\langle m, r \rangle$ as the *architectural configuration* and denote it α .

During the execution, we (partially) apply changes that occur in the ROB using a function $\text{apl}(\beta, r)$, which applies all effects of instructions in β to r [61]. When the value of an assignment in β is not resolved yet, the corresponding register is mapped to a special symbol $\perp$, meaning that it is undefined. Thus, apl generates a new register map where the value of some registers is undefined.

(Values) $a, v, \ell \in \mathcal{V}$ (Registers) $x, r \in \mathcal{R}$

$e ::= v \mid x \mid e_1 \otimes e_2$
 $inst ::= r \leftarrow e \mid \text{load } r \ e \mid \text{store } e_a \ e_v \mid \text{beqz } e \ \ell \mid \text{jmp } e$

Fig. 1. Syntax of assembly instructions: $\otimes$ is an abstract binary operator, $r \neq \text{pc}$. A program P is a partial mapping from locations to instructions and $P[\ell]$ denotes the instruction at location ℓ .

3) *Attacker strategies*: Our semantics is parameterized by a set of functions $\mathcal{F} = \{\text{next}, \text{predict}, \text{update}\}$, which model the next directive to execute, control flow predictions, and microarchitectural state updates, respectively. These functions can be adversarially chosen to model different attacker strategies. We call an *attacker strategy* $\mathcal{A} = (\mu_0, \mathcal{F})$ a choice of initial microarchitectural state μ_0 and functions $\mathcal{F}$. Given a program P and an attacker strategy $\mathcal{A} = (\mu_0, \mathcal{F})$, we let $\mathcal{A}\{P(\alpha)\}$ denote the sequence of observable microarchitectural states $\mu_0 \dots \mu_n$ produced by the execution of P from the initial configuration $\langle \alpha, \varepsilon, \mu_0 \rangle$ in a semantics parameterized by functions $\mathcal{F}$. For an n -step execution $\langle \alpha, \varepsilon, \mu_0 \rangle \rightarrow \dots \rightarrow \langle \alpha_n, \beta_n, \mu_n \rangle$, $\mathcal{A}\{P(\alpha)\} = \mu_0 \dots \mu_n$.

4) *Leakage model*: The function **update** models information leakage by updating the microarchitectural context μ . Intuitively, all arguments of the update function are leaked and become attacker-observable. Following the common constant-time leakage model, $\{\cdot\}^{\text{BASE}}$ leaks the *program counter* of fetched and executed instructions and the *memory addresses* accessed by (speculative) loads and stores.

5) *Semantics*: Our base semantics $\{\cdot\}^{\text{BASE}}$ models the execution of assembly programs at the microarchitectural level with a deterministic relation $c_1 \rightarrow c_2$, meaning that a step in the configuration c_1 produces a unique configuration c_2 . It captures direct and indirect control flow speculation (Spectre-v1 and Spectre-v2), similar to existing microarchitectural semantics [20], [30]. It is unknown how existing DMPs interact with other speculation mechanisms like memory disambiguation [32], load value [37], or load address [38] prediction.

The semantics uses *directives* to model the different stages of an out-of-order pipeline: fetch, execute i for $i \in \mathbb{N}$, and retire. At each step, the semantics selects the next directive d to execute using the function $d = \text{next}(\mu)$ and applies the corresponding rule for that directive, modeled as an auxiliary relation $c_1 \xrightarrow{d} c_2$. Semantics rules are standard; we give an excerpt of the complete rules [61] in [Appendix A](#).

B. DMP Extensions ($\{\cdot\}^{\text{DMP}}$ and $\{\cdot\}^{\text{B+DMP}}$)

To model the behavior of DMPs, we introduce two functions, **train** (μ, ℓ, m, a) and **prefetch** (μ, ℓ, m, a) , modeling training and prefetching operations. Both functions take as input the current microarchitectural state μ , the location ℓ of the instruction triggering the DMP, the address a loaded by the instruction, and the memory m . For generality, we give the DMP access to the whole memory: this allows us to model DMPs that perform recursive prefetches based on loaded values. Both functions return a new microarchitectural state μ' modeling the updates performed by the DMP. In general,

$$\begin{array}{c}
\text{EXECUTE-LOAD-DMP} \\
|\beta| = i - 1 \quad inst = \text{load} \times e@T \\
\boxed{\ell \triangleq \llbracket pc \rrbracket_{apl}(\beta, r)} \quad a \triangleq \llbracket e \rrbracket_{apl}(\beta, r) \\
v \triangleq m(a) \quad inst' \triangleq x \leftarrow v@T \quad \text{store} _ _ \notin \beta \\
\boxed{\mu_{train} \triangleq \text{train}(\mu, \ell, m, a)} \quad \boxed{\mu_{pref} \triangleq \text{prefetch}(\mu, \ell, m, a)} \\
\mu' \triangleq \text{update}(\mu, \mu_{train}, \mu_{pref}, a) \\
\hline
\langle m, r, \beta \cdot inst \cdot \beta', \mu \rangle \xrightarrow[\text{execute } i]{\text{DMP}} \langle m, r, \beta \cdot inst' \cdot \beta', \mu' \rangle
\end{array}$$

Fig. 2. Load evaluation for $\llbracket \cdot \rrbracket^{\text{DMP}}$. Boxes mark differences with $\llbracket \cdot \rrbracket^{\text{BASE}}$.

we leave these functions undefined: they can be instantiated to model different DMP implementations. This also means that an attacker strategy $\mathcal{A}$ includes **train** and **prefetch** functions.

1) *Modeling existing implementations* ($\llbracket \cdot \rrbracket^{\text{DMP}}$): In existing DMP implementations, the DMP is trained and activated on memory loads. To model this behavior, we extend the LOAD rule of $\llbracket \cdot \rrbracket^{\text{BASE}}$ with the **train** and **prefetch** functions, obtaining the **EXECUTE-LOAD-DMP** rule defined in Figure 2. To show that our model captures existing DMP designs, we now present a high-level model of the Apple DMP and include models of the Intel DMP, CDP, and IMP in Appendix B.

Figure 3 contains the model of the Apple DMP [17] described in Section IV-B1 (the gray box will be explained later and can be ignored for now). Since this DMP does not have a training phase, we only define the **prefetch** function. This function applies Apple’s pointer heuristic to all words (i.e., candidate pointers) in the loaded cache block (lines 2–7). If the heuristic passes, it dereferences the candidate pointer (line 8). To model the DMP’s next-line prefetching behavior, the function also prefetches the cache block following the pointer’s target in memory (line 8). With this behavior, our model captures leakage through both the *Arch* and *Sub* classes. It also captures leakage through the *Trans* class, as the semantics allows **prefetch** to be called on speculative loads. However, since the specific mechanism responsible for the Apple DMP’s *Pref* leakage is unknown (as mentioned in Section IV-B1), we do not capture this leakage class in our model.

2) *Generalization* ($\llbracket \cdot \rrbracket^{\text{B+DMP}}$): More generally, given an arbitrary hardware semantics $\mathcal{B}$, we define a **DMP-enhanced** semantics $\llbracket \cdot \rrbracket^{\text{B+DMP}}$. We assume $\mathcal{B}$ is equipped with two functions $\text{get_pc}(c, d)$ and $\text{get_addr}(c, d)$, which, given the current configuration and directive, respectively return the current program counter and the address being loaded (get_addr returns $\perp$ if no address is accessed). We also add a predicate $\text{dmp_activate}(\mu, \ell, d)$, which returns whether the DMP is activated at the current location ℓ and directive d . These functions are defined by the semantics, not attacker-controlled.

The semantics of $\llbracket \cdot \rrbracket^{\text{B+DMP}}$ is defined in Figure 4. Each rule follows the base semantics $\mathcal{B}$ to compute the next configuration. Additionally, they update the microarchitectural state to reflect whether the DMP was activated. If this is the case (i.e., dmp_activate is true), we trigger the DMP using the **train** and **prefetch** functions. To leave the order in which microarchitec-

```

1:  function prefetch( $\mu, \ell, m, a$ ):
2:      base :=  $a - a \bmod n$ 
3:      end := base + n - 1
4:      for address := base to end do
5:          if  $\pi_m(\text{address}) = L$  then
6:              target :=  $m(\text{address})$ 
7:              if pointerHeuristic(target, address) then
8:                   $\mu := \text{update}(\mu, \text{target}, \text{target} + n)$ 
9:      return  $\mu$ 

10: function pointerHeuristic(target, address):
11:     return target[55:32] = address[55:32]

```

Fig. 3. Model of the Apple DMP where n represents the number of words in a cache block.

$$\begin{array}{c}
\text{DMP-ACTIVATE} \\
c \triangleq \langle m, r, \beta, \mu \rangle \quad \ell \triangleq \text{get_pc}(c, d) \\
\text{dmp_activate}(\mu, \ell, d) \quad a^\perp \triangleq \text{get_addr}(c, d) \\
\langle m, r, \beta, \mu \rangle \xrightarrow[d]{\mathcal{B}} \langle m', r', \beta', \mu' \rangle \\
\mu_{train} \triangleq \text{train}(\mu, \ell, m, a^\perp) \quad \mu_{pref} \triangleq \text{prefetch}(\mu, \ell, m, a^\perp) \\
\mu'' \triangleq \text{update}(\mu, \mu', \text{activate}, \mu_{train}, \mu_{pref}, a^\perp) \\
\hline
\langle m, r, \beta, \mu \rangle \xrightarrow[d]{\mathcal{B+DMP}} \langle m', r', \beta', \mu'' \rangle \\
\\
\text{DMP-NO-ACTIVATE} \\
\ell \triangleq \text{get_pc}(c, d) \\
\neg \text{dmp_activate}(\mu, \ell, d) \quad \langle m, r, \beta, \mu \rangle \xrightarrow[d]{\mathcal{B}} \langle m', r', \beta', \mu' \rangle \\
\mu'' \triangleq \text{update}(\mu, \mu', \text{no-activate}) \\
\hline
\langle m, r, \beta, \mu \rangle \xrightarrow[d]{\mathcal{B+DMP}} \langle m', r', \beta', \mu'' \rangle
\end{array}$$

Fig. 4. Semantics of $\llbracket \cdot \rrbracket^{\text{B+DMP}}$, with $a^\perp \in \mathcal{V} \cup \{\perp\}$.

tural effects are combined unspecified, we pass the incoming microarchitectural state together with every computed state to **update**, which produces the final microarchitectural state. Note that, compared to the base semantics, the **DMP-ACTIVATE** rule only modifies the microarchitectural state; the architectural state and reorder buffer follow the base semantics.

C. Formalization of SecDMP ($\llbracket \cdot \rrbracket^{\text{SecDMP}}$ and $\llbracket \cdot \rrbracket^{\text{B+SecDMP}}$)

1) *Memory partitioning*: **SecDMP** assumes that memory is partitioned between public (low, L) and secret (high, H) data. We model this assumption using a memory partition π_m that maps memory addresses to their security level. Similar to prior work [20], we require this partition to be fixed (i.e., it cannot change over time), meaning that any secret data written to the public memory region is *declassified*. Our semantics does not make a distinction between declassified values and leakage; both are simply leaked to the microarchitectural state via **update**. We capture this with the rule **RETIRE-STORE-LOW** (cf. Figure 8 in Appendix A), which leaks the value v when storing it to a public address (i.e., $\pi_m(a) = L$). Note that declassification never occurs speculatively.

Given such a partition, we define the *public projection* of a memory m as the memory $m|_\pi$ such that for all addresses a :

$$m|_\pi(a) = \begin{cases} m(a) & \text{if } \pi_m(a) = L \\ \perp & \text{otherwise} \end{cases}$$

2) *Securing existing DMPs with SECDMP*: We first show how SECDMP can be applied to existing DMP designs by securing our model of the Apple DMP ($\{\cdot\}^{\text{DMP}}$). We modify the **prefetch** function as shown by the gray box in Figure 3: for any address *address* analyzed by the DMP, SECDMP first checks whether this address corresponds to public data ($\pi_m(\text{address}) = L$). If so, we proceed as in the insecure model and prefetch the corresponding target address if the pointer heuristic passes. Otherwise, we do not analyze the address as it contains secret data. Appendix B describes the secure models of the Intel DMP, CDP, and IMP.

3) *Generalization*: More generally, SECDMP can secure any DMP by restricting its behavior (**train** and **prefetch**) to public memory regions defined by $m|_\pi$:

$$\begin{aligned} \mu_{\text{train}} &\triangleq \text{train}(\mu, \ell, m|_\pi, a) \\ \mu_{\text{pref}} &\triangleq \text{prefetch}(\mu, \ell, m|_\pi, a) \\ \mu' &\triangleq \text{update}(\mu, \text{activate}, \mu_{\text{train}}, \mu_{\text{pref}}, a) \end{aligned}$$

SECDMP can therefore use the entire public memory for training and prefetching.

The individual SECDMP models for the Apple DMP, and those for the Intel DMP, CDP, and IMP in Appendix B, are naturally subsumed by a more abstract maximally-leaking model where **train** and **prefetch** in Figure 2 are kept abstract and restricted to the public memory projection $m|_\pi$. We call this model $\{\cdot\}^{\text{SECDMP}}$. Additionally, given an arbitrary hardware semantics $\{\cdot\}^B$, we define a SECDMP-enhanced semantics, $\{\cdot\}^{B+\text{SECDMP}}$, as in Figure 4, but with **train** and **prefetch** restricted to the public memory projection $m|_\pi$.

VII. SECURITY THEOREMS AND PROOFS

In this section, we prove that SECDMP instantiations do not leak more than what is already leaked (or declassified) by the baseline semantics they extend. The full formalization and proofs can be found in the extended version of our paper [61].

A. Definitions

1) *Public equivalence*: We extend the memory partition to registers by defining a partition $\pi = (\pi_m, \pi_r)$, where π_m maps memory addresses to their security level and π_r maps registers to their security level. Two architectural configurations α, α' are *public-equivalent* (or π -equivalent) according to a partition π , written $\alpha =_\pi \alpha'$, if they agree on their program counter, public memory, and public registers.

2) *Microarchitectural security*: Given a hardware semantics $\{\cdot\}$, a program P , a security partition π , and two initial architectural configurations α, α' , we say that P leaks on α, α' if α, α' are π -equivalent and there exists an attacker strategy $\mathcal{A}$ that produces distinct observations from them. Formally:

Definition 1

Given a hardware semantics $\{\cdot\}$, a program P leaks on α, α' under partition π , written $\text{leak}_{P,\pi}^{\{\cdot\}}(\alpha, \alpha')$, if:

$$\alpha =_\pi \alpha' \wedge \exists \mathcal{A}. \mathcal{A}\{P(\alpha)\} \neq \mathcal{A}\{P(\alpha')\}.$$

Conversely, if no pair of π -equivalent configurations leaks, we say that P is *secure* for security partition π on $\{\cdot\}$.

3) *Comparing the security of hardware semantics*: In the following, we compare the security guarantees of different hardware semantics (e.g., with and without DMPs). To do so, we define a relation $\sqsupseteq$ where $\{\cdot\}^{\text{OPT}} \sqsupseteq \{\cdot\}^B$ means that $\{\cdot\}^{\text{OPT}}$ does not leak any secret information that is not already leaked by $\{\cdot\}^B$. Formally:

Definition 2: $\{\cdot\}^{\text{OPT}} \sqsupseteq \{\cdot\}^B$

A hardware semantics $\{\cdot\}^{\text{OPT}}$ is at least as secure as a hardware semantics $\{\cdot\}^B$, written $\{\cdot\}^{\text{OPT}} \sqsupseteq \{\cdot\}^B$, if for every program P , security partition π , and pair of initial architectural configurations α, α' :

$$\text{leak}_{P,\pi}^{\{\cdot\}^{\text{OPT}}}(\alpha, \alpha') \implies \text{leak}_{P,\pi}^{\{\cdot\}^B}(\alpha, \alpha').$$

For example, a sequential hardware semantics $\{\cdot\}^{\text{SEQ}}$ is (typically) more secure than a speculative hardware semantics $\{\cdot\}^{\text{SPEC}}$ ($\{\cdot\}^{\text{SEQ}} \sqsupseteq \{\cdot\}^{\text{SPEC}}$) since sequential execution does not leak through speculation. Note that if $\{\cdot\}^{\text{OPT}} \sqsupseteq \{\cdot\}^B$, then all programs that are secure on $\{\cdot\}^B$ are also secure on $\{\cdot\}^{\text{OPT}}$.

B. Security Proof

1) *Security of $\{\cdot\}^{\text{SECDMP}}$* : We start by showing that our speculative semantics equipped with SECDMP does not leak more than $\{\cdot\}^{\text{BASE}}$. Formally:

Theorem 1

$$\{\cdot\}^{\text{SECDMP}} \sqsupseteq \{\cdot\}^{\text{BASE}}$$

Proof. We leave the full proof to our extended version [61] and sketch the main idea here. Fix two publicly equivalent initial configurations and assume an attacker $\mathcal{A}^{\text{SECDMP}}$ distinguishes them in $\{\cdot\}^{\text{SECDMP}}$; let $n + 1$ be the first step where the resulting traces diverge (i.e., the microarchitectural states are different). Without loss of generality, we assume this attacker to be equipped with a maximally-leaking **update** function that leaks all its arguments to the microarchitectural state—i.e., it represents the strongest possible leakage allowed by the semantics.¹ We then construct a baseline attacker $\mathcal{A}^{\text{BASE}}$ that distinguishes the same two configurations in $\{\cdot\}^{\text{BASE}}$ at the same step by simulating the $\{\cdot\}^{\text{SECDMP}}$ execution. By induction over the small step semantics of $\{\cdot\}^{\text{SECDMP}}$, we show

¹In particular, this attacker captures, in a non-destructive way, all leakage piled up by our semantics rules. This assumption does not restrict the generality of the proof since a divergence produced by any other **update** function is also captured by such a maximally-leaking **update** function.

that we can construct pairs of executions in $\{\cdot\}^{\text{BASE}}$ that (1) match the architectural state and reorder buffer of executions in $\{\cdot\}^{\text{SECDMP}}$ at every step, and (2) ensure that whenever the $\{\cdot\}^{\text{SECDMP}}$ executions diverge, the baseline executions also diverge. Intuitively, because $\{\cdot\}^{\text{SECDMP}}$ restricts **train** and **prefetch** to the public projection $m|_{\pi}$, any divergence caused by SECDMP must arise from data already leaked in the baseline model (program counters, accessed addresses, and public memory); thus $\{\cdot\}^{\text{BASE}}$ can always produce the same leakage as $\{\cdot\}^{\text{SECDMP}}$. $\square$

2) *Instantiating SecDMP implementations*: It is easy to see that the SECDMP implementations proposed in Section VI-C2 and Appendix B are instances of $\{\cdot\}^{\text{SECDMP}}$: they only use public memory for training and prefetching. Therefore, it directly follows from Theorem 1 that these SECDMP implementations are as secure as the baseline speculative semantics.

C. Generalization

We generalize our results for an implementation of SECDMP combined with an arbitrary baseline B . Specifically, we show that under natural assumptions on the base semantics $\{\cdot\}^{\text{B}}$, the SECDMP-enhanced semantics $\{\cdot\}^{\text{B+SECDMP}}$ is at least as secure as the base semantics $\{\cdot\}^{\text{B}}$.

1) *Assumptions over $\{\cdot\}^{\text{B}}$* : We give a high-level summary and leave formal definitions to the extended version [61].

- A1** The microarchitectural state μ is updated only through **update**, and never directly from the architectural state or reorder buffer.
- A2** The semantics cannot encode information from μ into the architectural state or reorder buffer without using **predict** or **next**.
- A3** The base semantics respects a constant-time leakage model, exposing at least the program counter and accessed memory addresses, and it declassifies values written to public memory.

A1 and **A2** ensure that leakage and attacker-controlled speculation are modeled by dedicated interfaces in the semantics—i.e., the attacker can neither observe nor influence execution outside the behavior exposed by these interfaces.

2) *Security of $\{\cdot\}^{\text{B+SECDMP}}$* : Under these assumptions, we can show that $\{\cdot\}^{\text{B+SECDMP}}$ is at least as secure as (or does not leak more than) the base speculative semantics $\{\cdot\}^{\text{B}}$. Formally:

Theorem 2

Given a base speculative semantics $\{\cdot\}^{\text{B}}$ satisfying **A1–3**, $\{\cdot\}^{\text{B+SECDMP}} \sqsupseteq \{\cdot\}^{\text{B}}$

Proof. We leave the full proof to the extended version [61] and sketch the main idea here. The proof follows the same structure as that of Theorem 1. Fix two publicly equivalent initial configurations and assume an attacker $\mathcal{A}^{\text{B+SECDMP}}$ distinguishes them in $\{\cdot\}^{\text{B+SECDMP}}$. We then construct a baseline attacker $\mathcal{A}^{\text{BASE}}$ that simulates the $\{\cdot\}^{\text{B+SECDMP}}$ execution in $\{\cdot\}^{\text{B}}$ and distinguishes the same two configurations. We use

A2 to match the architectural states and reorder buffers of the corresponding executions in $\{\cdot\}^{\text{B}}$ and $\{\cdot\}^{\text{B+SECDMP}}$ by matching their predictions and directives. We use **A1** to ensure that all baseline microarchitectural changes pass through **update**, allowing us to reproduce any divergence originating in the underlying baseline step by choosing an update function for $\{\cdot\}^{\text{B}}$ that produces different results in the two executions. Finally, **A3** ensures that public memories remain equal before the divergences and that differences in the program counters or loaded addresses supplied to SECDMP are already observable in the baseline. Thus, we can always construct an attacker for $\{\cdot\}^{\text{B}}$ that distinguishes the same pair of initial configurations as $\{\cdot\}^{\text{B+SECDMP}}$. $\square$

3) *Instantiation with sequential CT*: As a corollary of Theorem 2, we can instantiate **B** with a sequential semantics in the constant-time leakage model, denoted $\{\cdot\}^{\text{SEQCT}}$. This semantics corresponds to an ISA semantics that executes instructions sequentially without speculation while leaking only the program counter, accessed memory addresses, and declassifies values written to public memory. It is easy to see that $\{\cdot\}^{\text{SEQCT}}$ satisfies **A1–3**. Therefore, we can conclude that $\{\cdot\}^{\text{SEQCT+SECDMP}}$ is at least as secure as $\{\cdot\}^{\text{SEQCT}}$.

4) *Instantiation with ProSpecT*: As a final step, we combine SECDMP with ProSpecT [20], a state-of-the-art speculative-constant-time defense, to obtain a system that provides strong security guarantees against both speculative execution attacks and DMP attacks for constant-time programs. ProSpecT satisfies **A1–3**, therefore, by Theorem 2, we can conclude that $\{\cdot\}^{\text{PROSPECT+SECDMP}}$ does not leak more than $\{\cdot\}^{\text{PROSPECT}}$. Furthermore, from [20], we know that $\{\cdot\}^{\text{PROSPECT}}$ is at least as secure as a baseline sequential semantics in the constant-time leakage model (i.e., $\{\cdot\}^{\text{PROSPECT}} \sqsupseteq \{\cdot\}^{\text{SEQCT}}$). Combining these, we obtain that $\{\cdot\}^{\text{PROSPECT+SECDMP}} \sqsupseteq \{\cdot\}^{\text{SEQCT}}$.

In conclusion, $\{\cdot\}^{\text{PROSPECT+SECDMP}}$ provides strong end-to-end security guarantees against both speculative execution and DMP attacks for constant-time programs.

VIII. IMPLEMENTATION AND EVALUATION

In this section, we empirically evaluate the security, performance, and hardware cost of SECDMP, permanent disabling, and selective disabling. We also consider the combination of SECDMP and ProSpecT [20] to evaluate the cost of integrating SECDMP into existing Spectre defenses that rely on secret partitioning, protecting constant-time code in the presence of both speculative execution and DMPs.

A. Implementation

Our implementation extends the Proteus RISC-V processor [14], on which ProSpecT [20] was also implemented. Proteus is a configurable research processor that includes an out-of-order pipeline and advanced microarchitectural features such as caches, speculative execution, branch prediction, and a prefetching module. We also build on the Proteus ecosystem [50], containing benchmarks and tools for evaluation.

1) *Proteus+DMP*: First, we implement the four DMPs described in Section IV-B, creating, to the best of our knowledge, the first open-source DMP implementations. These will act as the four baselines for our evaluation in Section VIII-B; we will repeat the evaluation for each baseline and refer to them as Proteus+DMP.

We model the DMPs as faithfully as possible after the designs described in the literature, resulting in the leakage classes from Table I (including all *Trans* leakage). The only exception is the Apple DMP: since the mechanism responsible for its *Pref* leakage is unknown, we do not model it. As a consequence, our models of the Apple DMP and CDP are very similar, the main difference being that CDP is recursive and the Apple DMP is not. This enables sharing most of their implementation code. For IMP, we implement the default 2-level variant $A[B[i]]$. Since CDP and IMP feature several configurable parameters, we initialize them with the recommended values from the original papers [19], [73], e.g., a maximum recursion depth of 3 for CDP and a prefetch lookup table with 16 entries for IMP.

2) *Permanent and selective disabling*: Next, we implement prototypes of the defenses on Proteus+DMP. To model permanent disabling, we use the original version of Proteus with no DMP. For selective disabling, we add a control and status register (CSR) to toggle the DMP from software. To prevent instructions from executing speculatively with incorrect CSR values, each CSR write in software must be followed by a fence instruction.

3) *SecDMP*: Our SECDMP implementation partitions the memory into public and secret regions, similar to ProSpeCT [20], and prevents the DMP from scanning data marked as secret. The secret regions are determined by CSRs (four CSRs defining two regions in our implementation). To match the static partitioning of SECDMP’s security proof, the CSRs must be configured at program start and remain unchanged throughout the program. Notably, SECDMP requires only the DMP to distinguish public and secret data, requiring no secret tracking in the processor pipeline.

4) *ProSpeCT+SecDMP*: For the combination of ProSpeCT and SECDMP, we add the SECDMP extension to the upstream ProSpeCT implementation.

B. Evaluation

We run the evaluation on each prototype for all four DMPs, configuring the Proteus RISC-V processor with a 4-way 16-set cache, a 64-cycle miss penalty, a 32-entry reorder buffer, 8 arithmetic units, 3 load units, and 2 multiplication/division units. The experiments are run in Verilator [58] simulations. Numerical results are available in the extended paper [61] and in <https://github.com/JensTheGrens/secdmp>.

1) *Security*: We evaluate DMP security by running the leakage tests and attacks from Table II. Since ProSpeCT’s original implementation has been evaluated against Spectre-PHT [20], we additionally run five Spectre-PHT tests, each with a different leakage source (i.e., load, store, branch, jump, and division). For selective disabling, we modify the programs

TABLE IV
SECURITY EVALUATION RESULTS: (1) IMP LEAK, (2) AUGURY PoC, (3) GoFetch PoC, (4) PEEK-A-WALK, (5) DMP-SPECTRE, (6) RECURSION, (7) LOAD SECRET, (8) SAME CACHE BLOCK, (9) TRANSIENT LOAD, (A) PHT-LOAD, (B) PHT-STORE, (C) PHT-BRANCH, (D) PHT-JUMP, (E) PHT-DIVISION. LEAKAGE ON APPLE DMP (◻), INTEL DMP (◻), CDP (◻), IMP (◻), OR NONE (✓).

Defense	DMP									Spectre				
	1	2	3	4	5	6	7	8	9	A	B	C	D	E
Proteus+DMP	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻
Permanent Disabling	✓	✓	✓	✓	✓	✓	✓	✓	✓	◻	◻	◻	◻	◻
Selective Disabling	◻	◻	✓	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻
SECDMP	✓	✓	✓	✓	✓	✓	✓	✓	✓	◻	◻	◻	◻	◻
ProSpeCT	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻	◻
ProSpeCT+SECDMP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

to disable DMPs before each security-critical code region, and re-enable them at the end of each region. For SECDMP and ProSpeCT, we annotate all secrets, which are then moved to a separate memory region by the compiler and linker. We write the region’s boundaries to the CSRs at program start.

Our security evaluation methodology is based on non-interference testing, similar to prior work [20], [66], [67]. We craft two variants of each program, P_1, P_2 , that only differ in their secret values, and run them on each prototype. If P_1, P_2 exhibit a difference in any observable signal, we consider the secret as leaked. For the DMP attacks (1–6), we monitor all signals observable to an attacker, e.g., memory bus addresses, program counter, tag bits of instruction and data caches, and branch predictor state. For the three general DMP leakage tests and the Spectre tests, we additionally monitor internal DMP signals, considering data as leaked once the DMP has analyzed it. This keeps the leakage tests independent of any specific DMP’s activation criteria. Our results in Table IV confirm that:

- Our DMP tests leak secrets exclusively through the DMPs: each program leaks on at least one Proteus+DMP baseline,² while permanent disabling prevents all leakage.
- Selective disabling only prevents architectural leakage: it is secure for the GoFetch PoC and Load Secret programs, which only leak through the *Arch* class, and is insecure for programs that leak through other classes.
- SECDMP correctly mitigates all DMP secret leakage.
- ProSpeCT alone leaks through all our programs: while it prevents Spectre-PHT leakage, each Spectre test still architecturally loads a secret before the speculative access, which leaks the secret through the DMPs.
- ProSpeCT+SECDMP prevents all secret leakage from both Spectre-PHT and DMPs.

2) *Performance*: A meaningful benchmark for evaluating the performance overhead of the defenses should (1) compute on secret data and (2) see a significant performance increase from DMPs. The results then show for each defense whether

²All DMP leakage tests and attacks leak through their targeted DMPs. While the Augury PoC was originally demonstrated on the Apple DMP [64], it only leaks on the Intel DMP in our evaluation because we do not model the Apple DMP’s unknown *Pref* leakage mechanism.

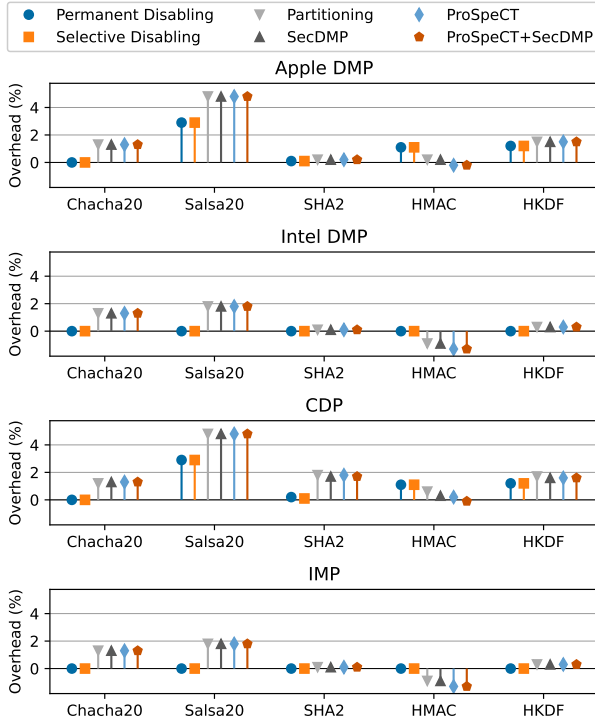


Fig. 5. HACl* performance overhead per DMP, relative to Proteus+DMP.

deploying it to strengthen security preserves the performance gained from data-dependent prefetching. For this purpose, we create *DMP-Bench*, inspired by SpectreGuard [26], which combines cryptographic and general-purpose benchmarks.

a) Cryptographic benchmarks: First, we evaluate the performance of several primitives from the HACl* cryptographic library [78]. As SECDDMP and ProSpeCT require secret partitioning, we manually annotate secrets as in Section VIII-B1. We partition five cryptographic primitives: Chacha20, Salsa20, SHA2 (256-bit variant), HMAC, and HKDF.³ In Section IX-A, we discuss how future work can overcome the partitioning challenges that arise for more complex programs. To isolate the performance overhead of the partitioning itself, we evaluate the partitioned programs on the baseline hardware configurations without any defense enabled. For selective disabling, we disable DMPs before calling each cryptographic function and re-enable them upon return.

The results in Figure 5 show that the defenses incur only a small overhead on the cryptographic benchmarks. The highest overhead of permanent disabling is only 2.9% (on Salsa20 for both the Apple DMP and CDP), indicating that these benchmarks barely benefit from DMPs in the first place. Selective disabling has the same overhead as permanent disabling, since it cannot re-enable DMPs during cryptographic functions. Additionally, the cost of secret partitioning ranges from -0.9% to +4.8%, with an average of +1.15% over

³We verify the partition correctness through a Proteus plugin. For Salsa20, we observe unnecessary secret spills introduced by the compiler, which we remove by patching the assembly.

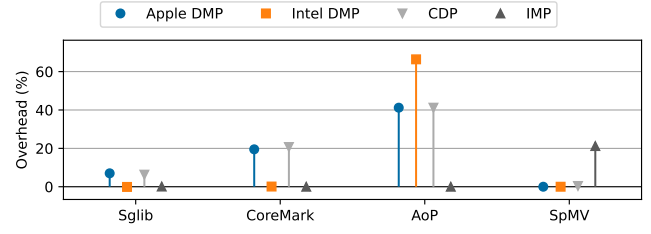


Fig. 6. Permanent disabling performance overhead on the different work benchmarks per DMP, relative to Proteus+DMP.

all DMPs and benchmarks, and varies by DMP: the Apple DMP and CDP perform worse on the partitioned programs. Finally, the overheads of SECDDMP and ProSpeCT stem solely from the partitioning method used; they add no additional cost, as SECDDMP only blocks secret-dependent prefetches and ProSpeCT only stalls speculative execution on secret data.

b) Work functions: The second component of DMP-Bench consists of public workloads, where no secrets are annotated for SECDDMP and ProSpeCT, and selective disabling never disables the DMP. In the following, we focus only on the results for permanent disabling, summarized in Figure 6, because the other defenses do not affect performance for public workloads (see the full results in the extended version [61]).

First, we evaluate the performance of the Embench benchmark suite [46] and CoreMark [25]. From Embench, only *sglib-combined* (Sglib in Figure 6) benefits from our DMPs: permanent disabling incurs a performance overhead of 7.0% for the Apple DMP and 6.3% for CDP. The other Embench benchmarks gain no performance from the DMPs (cf. [61]). For CoreMark, we observe a more significant performance impact: permanent disabling has an overhead of 20.5% for the Apple DMP and 20.7% for CDP. Neither *sglib-combined* nor CoreMark triggers the Intel DMP or IMP, as neither contains their required activation patterns. We therefore evaluate two additional workloads. For IMP, we take the SpMV benchmark from HPCG [23], which was evaluated in the IMP paper [73], and port it to Proteus. For the Intel DMP, we create a custom benchmark, AoP, that contains the array-of-pointers activation pattern. Figure 6 shows that both gain a significant performance increase from their respective DMPs: permanent disabling incurs an overhead of 21.2% on SpMV for IMP, and an overhead of 66.4% on AoP for the Intel DMP and 41.2% for both the Apple DMP and CDP.

c) DMP-Bench: Finally, we combine each work benchmark with each cryptographic primitive to obtain *DMP-Bench*, resulting in 20 work/crypto combinations. We run each combination in four configurations, varying the fraction of time spent in the work function at 25%, 50%, 75%, and 90%, for a total of 80 programs evaluated on each defense prototype for each DMP. The results are shown in Figure 7, reporting the geometric mean over all work/crypto combinations. The full results are available in the extended version [61].

Permanent disabling incurs a smaller overhead when most of the time is spent encrypting; this overhead increases as

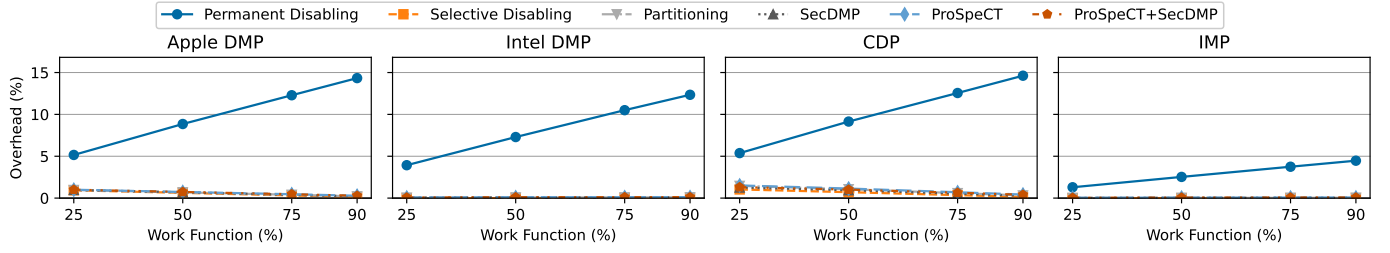


Fig. 7. Performance overhead on DMP-Bench, geometric mean over all work/crypto combinations per DMP, relative to Proteus+DMP.

TABLE V
HARDWARE AREA COST EVALUATION RESULTS.

Defense	Area (mm ²)			
	Apple DMP	Intel DMP	CDP	IMP
Proteus	0.6723	0.6723	0.6723	0.6723
Proteus+DMP	0.6858	0.7064	0.6906	0.8717
Selective Disabling	0.6861	0.7079	0.6910	0.8735
SECDDMP	0.6880	0.7102	0.6929	0.8758
ProSpeCT	0.7022	0.7228	0.7070	0.8881
ProSpeCT+SECDDMP	0.7031	0.7253	0.7080	0.8908

the work function’s share grows, with a maximum of 59.9% overhead at 90% work on AoP-Salsa20 for the Intel DMP. Neither selective disabling nor SECDDMP impacts DMP performance for the public work functions, solving the performance drawbacks of permanent disabling. Selective disabling has an average overhead of 0.26% over all DMPs and benchmarks. Its overhead is small at 25% work (at most 2.0% on AoP-Salsa20 for CDP) and decreases to near zero at higher work percentages, as it only disables DMPs during the crypto functions. SECDDMP’s average overhead is 0.36%; the overhead is highest at 25% work (at most 3.3% on CoreMark-Salsa20 for CDP) and also approaches zero as the work percentage increases. Again, the overhead stems solely from the secret partitioning method used; SECDDMP itself adds no overhead as it does not hinder prefetching on public data. Finally, ProSpeCT adds at most 0.5% over the partitioning cost.

3) *Hardware cost*: We use EVAL-HD [22] to measure the critical path and area overhead of the original version of Proteus, Proteus extended with the DMPs, and the hardware defenses. The critical path remains constant at 12.43 ns across all prototypes. The results in Table V show that:

- IMP introduces a substantial area increase of 29.7% caused by its large prefetch lookup table of 16 entries, while the Apple DMP, Intel DMP, and CDP introduce a 2.0%, 5.1%, and 2.7% area increase, respectively.
- Selective disabling and SECDDMP introduce a negligible area overhead of at most 0.2% and 0.5%, respectively.
- ProSpeCT alone has a small area overhead (highest is 2.4% for the Apple DMP). Extending ProSpeCT with SECDDMP increases area by at most 0.3%, since ProSpeCT itself already relies on secret partitioning.

IX. DISCUSSION AND FUTURE WORK

A. Practicality of Secret Partitioning

Our SECDDMP implementation uses a coarse-grained static memory partition, which comes with two main challenges. Importantly, these challenges stem from the concrete partitioning method used, not from SECDDMP’s design itself.

The first challenge is handling dynamic allocation of secret data on the heap, which can be solved by splitting the heap into a public and a secret region and differentiating between them at allocation time (either inferred from annotations or by providing a secure heap allocation function) [55], [60].

The second challenge is partitioning the stack. Coarse-grained partitioning cannot precisely separate public and secret data on the stack, often requiring the entire stack to be marked as secret [60]. In Section VIII-B, we avoid stack-resident secrets entirely by relocating them to a global memory region. While this approach works for simple programs, more complex programs risk the compiler spilling secret registers on the stack [20], [55], [60]. In addition, it may not be trivial to modify programs to change stack variables into global variables while maintaining functional correctness [60]. To address these issues, Song et al. recently proposed SecSep [60], an assembly type system that tracks secrets and automatically rewrites programs to partition secret and public data on the stack. While initially designed to support secure speculation defenses such as ProSpeCT [20], this technique directly applies to SECDDMP as well. Future work could integrate SecSep into compilers, potentially building on existing compiler-based secret-tracking mechanisms [48], [59], [76].

B. Marking Secrets

Similar to other microarchitectural defenses [20], [26], [55], [71], we achieve a good security/performance trade-off by requiring software to inform hardware about where secrets are located in memory. Hence, our work can be seen as additional evidence that extending the hardware-software interface with mechanisms to mark secrets in memory is a powerful enabler for secure and efficient defenses. Furthermore, we believe that our design could be extended to secure other optimizations in the memory subsystem. For instance, silent store suppression [41], which leaks whether a written value matches the one already present in memory [52], could be secured by never suppressing writes to addresses holding secret data.

C. Relation to ProSpeCT

ProSpeCT is a Spectre countermeasure with a formal proof of security [20]. Our design and formalization are inspired by ProSpeCT: both mitigations rely on secrecy tracking. However, while ProSpeCT guarantees security for constant-time programs in the presence of speculation, it fails to protect constant-time programs against leakage from DMPs.

SECDMP is motivated by this gap: we show that leakage induced by DMPs can be generally mitigated by reusing the same secrecy-tracking mechanism. Our formal proof goes beyond that of ProSpeCT: we not only show that the combination of SECDMP and ProSpeCT mitigates both leakage channels, but also that SECDMP eliminates all secret leakage through DMPs for an arbitrary baseline.

Our implementation shares the secret-tracking mechanism used by ProSpeCT and reuses the existing implementation for the combined evaluation of ProSpeCT+SECDMP. An important difference is that, contrary to ProSpeCT, which needs precise taint tracking through the pipeline, SECDMP only requires taint tracking in memory and the DMP, making it more realistic for complex implementations.

D. Reverse-Engineering of Commercial DMPs

The concrete DMP models in Section VI-B are based on prior reverse-engineering efforts and are only as precise as these efforts. For instance, the specific mechanisms responsible for the Apple DMP's *Pref* leakage reported by GoFetch [17] are unknown, so our model does not capture this leakage class. Similarly, it is unclear whether the Intel DMP leaks data during speculation (*Trans* class) or only architecturally. Importantly, our security proofs hold even if the current models are imprecise, as SECDMP's guarantees are established over an abstract DMP model that we believe can accurately capture any commercial design. Leveraging model-based testing [43] could be a promising approach to refining our models.

Another direction for future work is to reverse-engineer the commercial Arm DMP and analyze its security implications in terms of our leakage classes. This is of particular interest because Arm's security report [6] indicates that the DMP is recursive, potentially enabling the recursive attack from Section IV-C2 on real hardware.

E. Secure Selective Disabling

As discussed in Section IV-E, it is fundamentally difficult to predict when public code regions may trigger DMPs into scanning secrets, especially for the *Pref* and *Trans* classes. Consequently, secure selective disabling is only possible if all secrets are removed from memory before re-enabling DMPs. Concurrent work by Häuser et al. [34] makes a similar observation and, as a solution, suggests unmapping all secret data when exiting security-critical code regions. While this approach secures selective disabling across all leakage classes by making secrets inaccessible to DMPs, the authors acknowledge that it can introduce significant performance overhead.

X. CONCLUSION

SECDMP is a powerful and provably secure defense against side-channel leakage of DMPs. By requiring secrets to be tagged in memory, SECDMP ensures that no secrets leak through the DMP—without compromising the performance gained from prefetching based on public data. We propose a leakage classification to categorize DMP leakage and show that SECDMP is the only defense that mitigates all leakage classes across all DMP designs while introducing minimal performance overhead. Additionally, our open-source DMP implementations enable empirical validation of our results and can be used for future research on DMP security.

ACKNOWLEDGMENT

This research was partially funded by the Internal Funds KU Leuven, the Cybersecurity Research Program Flanders, and the Research Foundation – Flanders (FWO) under grant number G081322N. This work has benefited from a government grant managed by the National Research Agency under France 2030 with reference ANR-22-PECY-0009.

ETHICS CONSIDERATIONS

This research analyzes existing vulnerabilities in processors but does not demonstrate any attacks on production systems, only on the Proteus research core. We propose a defense that may help secure future products, but do not expose unknown vulnerabilities in current CPUs. However, as our research presents the first concrete demonstration that selective disabling does not prevent all DMP leakage, we notified Intel about an ambiguity in their documentation regarding selectively setting the DOIT bit on January 27, 2026.

REFERENCES

- [1] O. Aciğer, “Yet another microarchitectural attack: exploiting i-cache,” in *ACM Workshop on Computer Security Architecture (CSAW)*, 2007.
- [2] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, “Port contention for fun and profit,” in *IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [3] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub, “Jasmin: High-assurance and high-speed cryptography,” in *ACM Conference on Computer and Communications Security (CCS)*, 2017.
- [4] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, “Verifying constant-time implementations,” in *USENIX Security Symposium*, 2016.
- [5] Apple Inc., “Writing ARM64 code for apple platforms,” <https://developer.apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms>, 2025.
- [6] Arm, “Arm CPU security bulletin: CVE-2024-7881,” <https://developer.arm.com/documentation/110326/latest>, 2025.
- [7] S. Arranz Olmos, G. Barthe, L. Blatter, B. Grégoire, and V. Laporte, “Preservation of speculative constant-time by compilation,” *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 9, no. POPL, 2025.
- [8] S. Arranz Olmos, G. Barthe, C. Chuengsatiansup, B. Gregoire, V. Laporte, T. Oliveira, P. Schwabe, Y. Yarom, and Z. Zhang, “Protecting cryptographic code against spectre-rsb: (and, in fact, all known spectre variants),” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.
- [9] G. Barthe, M. Böhme, S. Cauligi, C. Chuengsatiansup, D. Genkin, M. Guarnieri, D. Mateos Romero, P. Schwabe, D. Wu, and Y. Yarom, “Testing side-channel security of cryptographic implementations against future microarchitectures,” in *ACM Conference on Computer and Communications Security (CCS)*, 2024.

- [10] D. J. Bernstein, "Cache-timing attacks on AES," Citeseer, 2005.
- [11] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, 2012.
- [12] S. Bhattacharya, C. Rebeiro, and D. Mukhopadhyay, "Hardware prefetchers leak: A revisit of SVF for cache-timing attacks," in *Annual IEEE/ACM International Symposium on Microarchitecture Workshops (MICROW)*, 2012.
- [13] D. D. Boggs, R. Segelken, M. Cornaby, N. Fortino, S. Chaudhry, D. Khartikov, A. Mooley, N. Tuck, and G. Vreugdenhil, "Memory type which is cacheable yet inaccessible by speculative instructions," Patent US10642744B2, May, 2020, filed June 28, 2018; granted May 5, 2020.
- [14] M. Bognar, J. Noorman, and F. Piessens, "Proteus: An extensible RISC-V core for hardware extensions," in *RISC-V Summit Europe*, 2023.
- [15] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security Symposium*, 2019.
- [16] S. Cauligi, C. Disselkoben, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.
- [17] B. Chen, Y. Wang, P. Shome, C. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, "GoFetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers," in *USENIX Security Symposium*, 2024.
- [18] Y. Chen, L. Pei, and T. E. Carlson, "AfterImage: Leaking control flow data and tracking load operations via the hardware prefetcher," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.
- [19] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.
- [20] L. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk, and F. Piessens, "ProSpecT: Provably secure speculation for the constant-time policy," in *USENIX Security Symposium*, 2023.
- [21] D. Davoli, M. Bognar, L.-A. Daniel, B. Grégoire, F. Piessens, and T. Rezk, "dfence: Fine-grained speculation barriers for efficient and effective hardware-software protection in the spectre era," in *ACM Conference on Computer and Communications Security (CCS)*, 2026.
- [22] J. De Meulemeester, Q. Norga, F. Piessens, I. Verbauwhede, and M. Bognar, "Hardware cost evaluation in systems security," in *ACM Conference on Reproducibility and Replicability (ACM REP)*, 2025.
- [23] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications (IJHPCA)*, vol. 30, no. 1, pp. 3–10, 2016.
- [24] W. R. Earnshaw, "Speculation-restricted memory region type," Patent US11809316B2, Nov., 2023, filed May 9, 2019; granted Nov 7, 2023.
- [25] EEMBC, "CoreMark," <https://www.eembc.org/coremark/>, 2009.
- [26] J. Fustos, F. Farshchi, and H. Yun, "SpectreGuard: An efficient data-centric defense mechanism against spectre attacks," in *Design Automation Conference (DAC)*, 2019.
- [27] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.
- [28] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: defeating cache side-channel protections with TLB attacks," in *USENIX Security Symposium*, 2018.
- [29] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *Network and Distributed System Security Symposium (NDSS)*, 2017.
- [30] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.
- [31] L. Hetterich, F. Thomas, L. Gerlach, R. Zhang, N. Bernsdorf, E. Ebert, and M. Schwarz, "ShadowLoad: Injecting state into hardware prefetchers," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.
- [32] J. Horn, "Speculative execution, variant 4: Speculative store bypass," <https://bugs.chromium.org/p/project-zero/issues/detail?id=1528>, 2018.
- [33] G. Hu, Z. He, and R. B. Lee, "SoK: Hardware defenses against speculative execution attacks," in *International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021.
- [34] M. Häuser and K. Schneider, "Secret types require os-backed secrecy code sections," in *Modeling and Verification for Secure and Performant Cyber-Physical Systems*, 2025.
- [35] Intel Corporation, "Data operand independent timing instruction set architecture guidance," <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html>, 2025.
- [36] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2016.
- [37] J. Kim, J. Chuang, D. Genkin, and Y. Yarom, "FLOP: Breaking the Apple M3 CPU via false load output predictions," in *USENIX Security Symposium*, 2025.
- [38] J. Kim, D. Genkin, and Y. Yarom, "SLAP: Data speculation attacks via load address prediction on Apple silicon," in *IEEE Symposium on Security and Privacy (S&P)*, 2025.
- [39] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.
- [40] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology - CRYPTO*, 1996.
- [41] K. M. Lepak and M. H. Lipasti, "On the value locality of store instructions," in *International Symposium on Computer Architecture (ISCA)*, 2000, pp. 182–191.
- [42] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "MemJam: A false dependency attack against constant-time crypto implementations," *International Journal of Parallel Programming*, vol. 47, no. 4, pp. 538–570, 2019.
- [43] O. Oleksenko, C. Fetzter, B. Köpf, and M. Silberstein, "Revizor: testing black-box CPUs against speculation contracts," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.
- [44] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Cryptographers' Track at the RSA conference (CT-RSA)*, 2006.
- [45] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *USENIX Security Symposium*, 2021.
- [46] D. Patterson, J. Bennett, M. Bennett, H. Chelin, D. Harris, J. Hellar, W. Jones, K. Moron, P. Savini, R. Shepherd, R. Simar, Z. Susskind, and S. Wallentowitz, "Embentch IOT 2.0 and DSP 1.0: Modern embedded computing benchmarks," *Computer*, vol. 58, no. 5, pp. 37–47, 2025.
- [47] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *USENIX Security Symposium*, 2016.
- [48] P. Philippe, M. Sabt, and P.-A. Fouque, "GnuZero: A compiler-based zeroization static detection tool for the masses," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2025.
- [49] T. Pornin, "BearSSL - Constant-Time Crypto," <https://bearssl.org/constanttime.html>, 2018.
- [50] Proteus ecosystem contributors, "The Proteus ecosystem," <https://github.com/proteus-core/ecosystem>.
- [51] A. Patschke, J. Wichelmann, and T. Eisenbarth, "Zebrafish: Mitigating memory-centric side-channel leakage via interleaving," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2025.
- [52] J. R. Sanchez Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher, "Opening pandora's box: A systematic study of new ways microarchitecture can leak private data," in *International Symposium on Computer Architecture (ISCA)*, 2021.
- [53] T. Schlüter, A. Choudhary, L. Hetterich, L. Trampert, H. Nemat, A. Ibrahim, M. Schwarz, C. Rossow, and N. O. Tippenhauer, "Fetch-Bench: Systematic identification and characterization of proprietary prefetchers," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.
- [54] T. Schlüter and N. O. Tippenhauer, "PreFence: A fine-grained and scheduling-aware defense against prefetching-based attacks," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2025.

- [55] M. Schwarz, M. Lipp, C. A. Canella, R. Schilling, F. Kargl, and D. Gruss, "ConTeXt: A generic approach for mitigating spectre," in *Network and Distributed System Security Symposium (NDSS)*, 2020.
- [56] R. K. Sharma, D. Grossman, and D. Kohlbrenner, "SplittingSecrets: A compiler-based defense for preventing data memory-dependent prefetcher side-channels," in *Microarchitecture Security Conference (uASC)*, 2026.
- [57] F. Sieck, Z. Zhang, S. Berndt, C. Chuengsatiansup, T. Eisenbarth, and Y. Yarom, "TeeJam: Sub-cache-line leakages strike back," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 1, pp. 457–500, 2023.
- [58] W. Snyder, "Verilator," <https://www.veripool.org/verilator/>.
- [59] L. Soares, M. Canesche, and F. M. Q. a. Pereira, "Side-channel elimination via partial control-flow linearization," *ACM Transactions on Programming Languages and Systems*, vol. 45, no. 2, 2023.
- [60] S. Song, T. Dong, K. Nwabueze, J. Zanders, A. Erbsen, A. Chlipala, and M. Yan, "Securing cryptographic software via typed assembly language," in *ACM Conference on Computer and Communications Security (CCS)*, 2025.
- [61] J. Sprengers, M. Bognar, L.-A. Daniel, and F. Piessens, "SecDMP: Provably secure data memory-dependent prefetching (extended version)," <https://arxiv.org/abs/2610.37454>, 2026.
- [62] J. Szefer, "Survey of microarchitectural side and covert channels, attacks, and defenses," *Journal of Hardware and Systems Security*, vol. 3, no. 3, pp. 219–234, 2019.
- [63] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, "TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering," in *USENIX Security Symposium*, 2022.
- [64] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *IEEE Symposium on Security and Privacy (S&P)*, 2022.
- [65] A. Wang, B. Chen, Y. Wang, C. W. Fletcher, D. Genkin, D. Kohlbrenner, and R. Paccagnella, "Peek-a-Walk: Leaking secrets via page walk side channels," in *IEEE Symposium on Security and Privacy (S&P)*, 2025.
- [66] H. Winderix, M. Bognar, L. Daniel, and F. Piessens, "Libra: Architectural support for principled, secure and efficient balanced execution on high-end processors," in *ACM Conference on Computer and Communications Security (CCS)*, 2024.
- [67] H. Winderix, M. Bognar, J. Noorman, L. Daniel, and F. Piessens, "Architectural mimicry: Innovative instructions to efficiently address control-flow leakage in data-oblivious programs," in *IEEE Symposium on Security and Privacy (S&P)*, 2024.
- [68] W. Xiong and J. Szefer, "Survey of transient execution attacks and their mitigations," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, 2021.
- [69] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security Symposium*, 2014.
- [70] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: a timing attack on OpenSSL constant-time RSA," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.
- [71] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious ISA extensions for side channel-resistant and high performance computing," in *Network and Distributed System Security Symposium (NDSS)*, 2019.
- [72] J. Yu, T. Jaeger, and C. W. Fletcher, "All your PC are belong to us: Exploiting non-control-transfer instruction BTB updates for dynamic PC extraction," in *International Symposium on Computer Architecture (ISCA)*, 2023.
- [73] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: indirect memory prefetcher," in *International Symposium on Microarchitecture (MICRO)*, 2015.
- [74] J. Zhang, C. Chen, J. Cui, and K. Li, "Timing side-channel attacks and countermeasures in CPU microarchitectures," *ACM Computing Surveys (CSUR)*, vol. 56, no. 7, 2024.
- [75] X. Zhang, B. Lv, Y. Lu, and K. Bu, "PhantomFetch: Obfuscating loads against prefetcher side-channel attacks," 2025.
- [76] Z. Zhang and G. Barthe, "CT-LLVM: Automatic large-scale constant-time analysis," *Cryptology ePrint Archive*, 2025.
- [77] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Ultimate SLH: taking speculative load hardening to the next level," in *USENIX Security Symposium*, 2023.
- [78] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Haci*: A verified modern cryptographic library," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

APPENDIX A

SEMANTICS OF $\{\cdot\}^{\text{BASE}}$ (EXCERPT)

We define a representative subset of rules in Figure 8 and leave the full semantics to the extended version [61].

Following the behavior of out-of-order processors, our semantics fetches program instructions in order with the fetch directive and places them in the ROB. Instructions in the ROB are marked with a tag (denoted $inst@T$) that tracks speculations: when an instruction $inst$ contains a predicted value, the tag records the location l of the instruction that produced the prediction; otherwise, the tag is ε . For instance, after fetching a branch or jump instruction, the rule **FETCH-PREDICT-BRANCH-JMP** predicts the next location l' and updates pc in the ROB accordingly.

The directive `execute  $i$` executes the i -th instruction in the ROB, effectively capturing out-of-order execution. For instance, the rule **JMP-COMMIT** models the execution of an indirect jump when the prediction is correct. It evaluates the actual jump target l' , ensures that it matches the predicted target l , and replaces the tag l_0 with ε to mark the instruction as resolved. The rule **EXECUTE-LOAD** executes a `load` instruction: it computes the address a , retrieves the corresponding value $m(a)$ from the memory, and updates the ROB. Note that our semantics allows loads to be executed speculatively (i.e., after a control-flow prediction), however, it does not allow for speculation about store-to-load dependencies nor store-to-load forwarding (i.e., the rule ensures that there is no preceding store instruction in the ROB).

Instructions are retired in order using the retire directive. Here, the effects of the instruction are committed to the architectural state (r and m). For instance, the rule **RETIRE-STORE-LOW** retires a store instruction on top of the ROB and updates the memory.

APPENDIX B

ADDITIONAL DMP FORMALIZATIONS

A. Formal Model of the Intel DMP

Figure 9 contains our Intel DMP model, which is informed by the prior reverse-engineering work [65] described in Section IV-B2. As explained there, we assume the Intel DMP can be trained and activated on speculative loads (i.e., the *Trans* class), which our speculative semantics captures by default. The gray boxes in Figure 9 depict the SECDDMP defense, and are ignored for the insecure model.

1) *Insecure model*: The `train` function in Figure 9 models the Intel DMP's training phase, which looks for an array-of-pointers dereferencing pattern. For each load, it remembers the address by appending it to μ (line 6) and later checks if subsequent loads dereference this address's value (lines 2–5). The training phase thus checks whether an architecturally loaded value equals a later load address, leaking through the *Arch* class. Since the pattern the DMP looks for occurs in a loop, it additionally remembers the location of the instruction that triggered the initial load, and internally matches this with the corresponding addresses to derive the stride value.

$$\begin{array}{c}
\text{FETCH-PREDICT-BRANCH-JMP} \\
\frac{\ell \triangleq \llbracket \text{pc} \rrbracket_{\text{apl}(\beta, r)} \quad P[\ell] \in \{\text{beqz } e _, \text{jmp } e\} \quad \ell' \triangleq \text{predict}(\mu) \quad \mu' \triangleq \text{update}(\mu, \ell)}{\langle m, r, \beta, \mu \rangle \xrightarrow{\text{fetch}} \langle m, r, \beta \cdot \text{pc} \leftarrow \ell' @ \ell, \mu' \rangle} \\
\\
\text{JMP-COMMIT} \\
\frac{|\beta| = i - 1 \quad \text{inst} = \text{pc} \leftarrow \ell @ \ell_0 \quad P[\ell_0] = \text{jmp } e \quad \ell' \triangleq \llbracket e \rrbracket_{\text{apl}(\beta, r)} \quad \ell' = \ell \quad \text{inst}' = \text{pc} \leftarrow \ell @ \varepsilon \quad \mu' \triangleq \text{update}(\mu, \ell', \ell_0)}{\langle m, r, \beta \cdot \text{inst} \cdot \beta', \mu \rangle \xrightarrow[\text{execute } i]{} \langle m, r, \beta \cdot \text{inst}' \cdot \beta', \mu' \rangle} \\
\\
\text{EXECUTE-LOAD} \\
\frac{|\beta| = i - 1 \quad \text{inst} = \text{load } x \ e @ T \quad a \triangleq \llbracket e \rrbracket_{\text{apl}(\beta, r)} \quad v \triangleq m(a) \quad \text{inst}' \triangleq x \leftarrow v @ T \quad \text{store } _ _ \notin \beta \quad \mu' \triangleq \text{update}(\mu, a)}{\langle m, r, \beta \cdot \text{inst} \cdot \beta', \mu \rangle \xrightarrow[\text{execute } i]{} \langle m, r, \beta \cdot \text{inst}' \cdot \beta', \mu' \rangle} \\
\\
\text{RETIRE-STORE-LOW} \\
\frac{\beta = \text{store } a \ v @ \varepsilon \cdot \beta' \quad \mu' = \text{update}(\mu, a, v) \quad \pi_m(a) = L}{\langle m, r, \beta, \mu \rangle \xrightarrow{\text{retire}} \langle m[a \mapsto v], r, \beta', \mu' \rangle}
\end{array}$$

Fig. 8. Evaluation rules for $\llbracket \cdot \rrbracket^{\text{BASE}}$ (excerpt).

The **prefetch** function represents the activation of the DMP after a stride has been learned (lines 9–10). Using the stride, it calculates an address ahead in the pattern (line 11), performs the check that limits the maximum stride value (line 12), loads the candidate pointer at this address (line 14), applies the pointer heuristic (line 15), and finally dereferences the candidate pointer if the heuristic passes (line 16). It thus loads and subsequently leaks data located ahead in the pattern, i.e., through the *Pref* class.

Intel’s pointer heuristic (lines 18–22) succeeds only if the upper 17 bits of the candidate pointer are all 0, restricting prefetching to user memory [65]. Additionally, if linear address masking (LAM)⁴ is enabled, the heuristic only checks if the bits at indices 63 and 47 are 0.

2) *Applying SecDMP*: To secure the Intel DMP with SEC DMP, we modify it as depicted by the gray boxes in Figure 9. We modify both the **train** and **prefetch** functions, ensuring the DMP neither trains on nor dereferences secrets.

B. Formal Model of CDP

Figure 10 contains the model of CDP [19] described in Section IV-B3. The gray box depicts the SEC DMP defense, which is ignored for the insecure model.

1) *Insecure model*: Since CDP has no training phase, we only require the **prefetch** function. To model CDP’s recursive behavior, we define **prefetchRec**, which takes the current recursion depth *depth* as an argument. If *depth* exceeds CDP’s

⁴LAM is a recently added feature allowing software to embed metadata in the highest-order bits of virtual addresses.

```

1: function train( $\mu, \ell, m, a$ ):
2:    $\text{previousLoads} := \text{getLoadHistory}(\mu)$ 
3:   for  $\langle \ell', a' \rangle$  in  $\text{previousLoads}$  do
4:     if  $\pi_m(a') = L$  then
5:        $\mu := \text{update}(\mu, \ell', a', a, m(a') = a)$ 
6:        $\mu := \text{appendToLoadHistory}(\mu, \langle \ell, a \rangle)$ 
7:   return  $\mu$ 
8: function prefetch( $\mu, \ell, m, a$ ):
9:    $\text{stride} := \text{predictStride}(\mu, \ell)$ 
10:  if  $\text{stride} \neq 0$  then
11:     $\text{address} := a + \text{stride} \times 16$ 
12:    if  $\text{address}[47:18] = a[47:18]$  then
13:      if  $\pi_m(\text{address}) = L$  then
14:         $\text{target} := m(\text{address})$ 
15:        if pointerHeuristic( $\text{target}$ ) then
16:           $\mu := \text{update}(\mu, \text{target})$ 
17:    return  $\mu$ 
18: function pointerHeuristic( $\text{target}$ ):
19:  if LAM then
20:    return  $\text{target}[63] = 0$  and  $\text{target}[47] = 0$ 
21:  else
22:    return  $\text{target}[63:47] = 0$ 

```

Fig. 9. Model of the Intel DMP. **getLoadHistory** extracts the history of load addresses from μ that was appended with **appendToLoadHistory**. **predictStride** returns the stride learned during the training phase, or zero if no activation pattern is detected.

configurable maximum *maxRecursionDepth*, the function returns (lines 4–5). Otherwise, it applies CDP’s pointer heuristic to every word in the loaded cache block (lines 6–11). If the heuristic passes, it prefetches the target and the next *nextLinePrefetches* adjacent cache blocks, thereby leaking the target (lines 12–13). Because every word in the loaded cache block is scanned, the model captures CDP’s *Arch*, *Sub*, and *Trans* leakage (the latter because, in our semantics, speculative loads trigger prefetchers by default). Finally, the function calls **prefetchRec** recursively on each prefetched target (line 14), capturing the *Pref* leakage class.

2) *Applying SecDMP*: To secure CDP with SEC DMP, we modify the model to never scan secret data (gray box in Figure 10): at line 9, it checks whether the address holds public data and only proceeds with the prefetch if it does.

C. Formal Model of IMP

Figure 11 depicts the model of IMP (cf. Section IV-B4) as described in [73]. The gray boxes depict the SEC DMP defense, and are ignored for the insecure model.

1) *Insecure model*: The **train** function models IMP’s training phase, which looks for the indirect array access pattern: indexing one array (i.e., the indirect array) using the values of another array (i.e., the index array). First, IMP looks for a regular stride access pattern that loads the index values of the

```

1: function prefetch( $\mu, \ell, m, a$ ):
2:   return prefetchRec( $\mu, \ell, m, a, 0$ )

3: function prefetchRec( $\mu, \ell, m, a, depth$ ):
4:   if  $depth > maxRecursionDepth$  then
5:     return  $\mu$ 
6:    $base := a - a \bmod n$ 
7:    $end := base + n - 1$ 
8:   for  $address := base$  to  $end$  do
9:     if  $\pi_m(address) = L$  then
10:       $target := m(address)$ 
11:      if pointerHeuristic( $target, address$ ) then
12:        for  $line := 0$  to  $nextLinePrefetches$  do
13:           $\mu := \text{update}(\mu, target + n \times line)$ 
14:           $\mu := \text{prefetchRec}(\mu, \ell, m,$ 
             $target + n \times line, depth + 1)$ 
15:   return  $\mu$ 

16: function pointerHeuristic( $target, address$ ):
17:   return  $target.highBits(compareBits)$ 
     $= address.highBits(compareBits)$ 

```

Fig. 10. Model of CDP where n represents the number of words in a cache block. CDP has a configurable $maxRecursionDepth$, number of $nextLinePrefetches$ and number of $compareBits$ for the pointer heuristic.

index array. For this purpose, line 2 stores the current program counter and address.

Next, lines 3–9 detect the indirect pattern itself. Since indirect accesses are not part of the index array’s stride pattern, IMP first checks that the current load matches no regular stride pattern (line 3). If so, it gets two earlier addresses a_1 and a_2 that are part of a common stride pattern (line 4), i.e., potentially two different accesses into the index array. IMP then checks whether the current load of address a is an access into the indirect array that uses the value stored at a_2 as index. To do so, it computes one candidate indirect base address by subtracting this index from a , and another by subtracting the value stored at a_1 (presumably an earlier index access) from a previously loaded address a_c that also did not match a regular stride pattern (lines 7–8). If both candidates match, a and a_c are likely accesses belonging to the same indirect pattern, and IMP registers the pattern at line 9. Since this state update depends on the previously loaded values at a_1 and a_2 , it causes leakage through the *Arch* class. As the size of the indirect array’s elements is unknown, IMP considers several shift values s when computing the candidates (line 6).

The following lines, i.e., lines 10–13, verify detected indirect patterns: for a load matching no regular stride pattern (line 3), IMP checks, for each learned pattern (with indirect base address b and shift s) and for a recent index access a_i (with program counter ℓ') into that pattern’s index array, whether the current address a equals the indirect base address b plus the index $m(a_i)$ shifted by s (lines 10–12). If so, a

```

1: function train( $\mu, \ell, m, a$ ):
2:    $\mu := \text{update}(\mu, \langle \ell, a \rangle)$ 
3:   if not isStrideAccess( $\mu, \ell, a$ ) then
4:     for  $\langle \ell', a_1, a_2 \rangle$  in  $getIndexAddresses(\mu)$  do
5:       if  $\pi_m(a_1) = L$  and  $\pi_m(a_2) = L$  then
6:         for  $s$  in  $shifts$  do
7:           for  $a_c$  in  $getCandidates(\mu)$  do
8:             if  $a - (m(a_2) \ll s)$ 
               $= a_c - (m(a_1) \ll s)$ 
              then
9:                $\mu := \text{setIndirectPattern}(\mu, \ell', s,$ 
                 $a - (m(a_2) \ll s))$ 
10:            for  $\langle \ell', s, b, a_i \rangle$  in  $getIndirectPatterns(\mu)$  do
11:              if  $\pi_m(a_i) = L$  then
12:                if  $a = b + (m(a_i) \ll s)$  then
13:                   $\mu := \text{increaseConfidence}(\mu, \ell')$ 
14:                 $\mu := \text{appendCandidate}(\mu, a)$ 
15:   return  $\mu$ 

16: function prefetch( $\mu, \ell, m, a$ ):
17:   if isConfident( $\mu, \ell$ ) then
18:      $address := a + \text{getDistance}(\mu, \ell)$ 
19:     if  $\pi_m(address) = L$  then
20:        $\langle s, b \rangle := \text{getIndirectPattern}(\mu, \ell)$ 
21:        $target := b + (m(address) \ll s)$ 
22:        $\mu := \text{update}(\mu, address, target)$ 
23:   return  $\mu$ 

```

Fig. 11. Model of IMP. *isStrideAccess* reports whether a load belongs to a regular stride pattern, and *getIndexAddresses* returns the earlier address pairs of such a pattern along with the associated program counter. *appendCandidate* records loads matching no stride pattern, retrieved by *getCandidates*. *setIndirectPattern* registers a detected indirect pattern, accessed via *getIndirectPatterns* and *getIndirectPattern*. *increaseConfidence* and *isConfident* track whether a pattern was observed often enough to activate, and *getDistance* returns how far ahead in the index array to prefetch. IMP is parameterized by its shift values (*shifts*); its activation confidence and maximum prefetch distance are abstracted by *isConfident* and *getDistance*.

is likely an access into the corresponding indirect pattern, and IMP increases the pattern’s confidence (line 13), again leaking through *Arch*.

Finally, line 14 records the current address a as a candidate indirect address, since it matched no regular stride.

The *prefetch* function activates IMP upon loading an entry of the index array of a learned, sufficiently confident indirect pattern, identified by the program counter of the index loads (line 17). It then loads an entry further ahead in the index array, shifts it, adds it to the indirect base address, and prefetches the result (lines 18–22), hence capturing IMP’s *Pref* leakage. Since our semantics allows speculative loads by default, the model also captures the *Trans* leakage class.

2) *Applying SecDMP*: The gray boxes in Figure 11 depict the modifications required to secure IMP with SEC DMP: they ensure that *train* never analyzes secret data and that *prefetch* never derives prefetch targets from secrets.

APPENDIX C

ARTIFACT APPENDIX

The SECDMP artifact contains all implementations, benchmarks, and scripts from SECDMP’s implementation and evaluation (cf. [Section VIII](#)). It consists of the source code of the Proteus RISC-V processor, the four open-source DMP implementations (the Apple DMP, Intel DMP, CDP, and IMP, see [Section IV-B](#)), and the implementation of SECDMP and of the defenses we compare against (i.e., permanent disabling, selective disabling, ProSpeCT and ProSpeCT+SECDMP). It additionally contains all security and performance benchmarks and all evaluation scripts, including the scripts to generate the paper’s figures and tables. A Dockerfile provides the complete build environment for reproducing the evaluation.

A. Description & Requirements

1) *How to access:* The artifact is available at <https://github.com/JensTheGrens/secdmp> and has been archived on Zenodo at <https://doi.org/10.5281/zenodo.23691560>. It contains a comprehensive README.md file, describing all artifacts and containing detailed instructions for reproducing the evaluation results, and is released under the MIT license.

2) *Hardware dependencies:* None.

3) *Software dependencies:* The artifact includes a Dockerfile containing all necessary dependencies. As such, the only requirement is Docker and approximately 20 GB of disk space for the Docker container, Docker build cache, additional storage for temporary files during the Docker build process, and the space occupied by the compiled benchmarks.

4) *Benchmarks:* All benchmarks are included in the artifact repository. The security evaluation uses our nine DMP leakage programs (IMP Leak, Augury PoC, GoFetch PoC, Peek-a-Walk, DMP-Spectre, Recursion, Load Secret, Same Cache Block, Transient Load) and the five Spectre-PHT tests (PHT-Load, PHT-Store, PHT-Branch, PHT-Jump, PHT-Division). The performance evaluation uses five cryptographic primitives (Chacha20, Salsa20, SHA2, HMAC, HKDF) from the HACLS* cryptographic library, the Embench benchmark suite, CoreMark, the SpMV benchmark from HPCG, our Array-of-Pointers (AoP) benchmark, and finally DMP-Bench, which combines each work benchmark with each cryptographic primitive at four work percentage variations.

B. Artifact Installation & Configuration

Perform the following steps to install and configure the artifact and its dependencies:

- 1) Install Docker following the instructions at <https://docs.docker.com/engine/install/>;
- 2) Clone the artifact repository from <https://github.com/JensTheGrens/secdmp>;
- 3) Build the Docker image (takes approximately 20 compute-minutes, detailed instructions are provided in the artifact’s README.md).

The `verify_installation.sh` script runs a minimal working example in the container to verify the installation, taking about one compute-minute.

Afterwards, perform the following two steps to build all the RISC-V processor prototypes (Proteus+DMP, permanent disabling, selective disabling, SECDMP, ProSpeCT, and ProSpeCT+SECDMP, for each of the four DMPs) and to compile all benchmarks used in the evaluation. This takes about 75 compute-minutes.

- 1) Build the prototypes by executing the `build_processor_sims.sh` script;
- 2) Compile the benchmarks by running `make`.

C. Major Claims

The main claims are that SECDMP mitigates all DMP secret leakage (C1) while introducing minimal performance overhead (C2) and low hardware cost (C3), improving over the state-of-the-art DMP defenses, i.e., permanent disabling and selective disabling, as they incur high performance overhead (C2) and fail to mitigate all DMP leakage (C1), respectively. Additionally, SECDMP can be combined with strong speculative-execution defenses such as ProSpeCT to obtain full microarchitectural security against both DMP leakage and Spectre attacks (C1). Each claim is explained in detail below.

- (C1): SECDMP mitigates all DMP secret leakage, while the selective disabling defense only prevents architectural DMP leakage. Moreover, combining SECDMP with ProSpeCT prevents leakage from both Spectre-PHT (which ProSpeCT’s existing implementation has been evaluated against) and DMPs. This claim is supported by (E1) and reported in [Table IV, Section VIII-B1](#).
- (C2): SECDMP (and selective disabling) preserves the performance benefit of the DMPs, whereas permanently disabling them incurs a significant overhead (up to 66.4 % on our work benchmarks and 59.9 % on DMP-Bench). SECDMP’s own overhead stems solely from the secret partitioning method used (at most 4.8 % and an average of 1.15 % on the cryptographic benchmarks). This is supported by experiments (E2), (E3) and (E4), whose results are reported in [Figures 5 to 7, Section VIII-B2](#).
- (C3): SECDMP’s hardware cost is negligible: it increases the area by at most 0.5 %, compared to 0.2 % for selective disabling, and leaves the critical path unchanged at 12.43 ns. Combining SECDMP with ProSpeCT increases the area by only 0.3 %. This claim is supported by (E5) and reported in [Table V, Section VIII-B3](#).

D. Evaluation

All experiments share the preparation steps described in [Appendix C-B](#), which need to be performed only once. Moreover, each experiment outputs its results to the `results/` directory. The artifact includes an `expected_results/` directory, which contains our own output, and which should be compared against the generated output in `results/`. Since the simulation is deterministic, results obtained with the provided Docker image match `expected_results/` exactly, except for the hardware evaluation, whose numbers can fluctuate negligibly between runs.

Experiments (E1)–(E5) can be run individually as described in the artifact’s `README.md`. Afterwards, follow the `README.md`’s instructions to generate the paper’s figures and tables, which can be compared against the figures and tables provided in `expected_results/`. The artifact additionally includes the `run_secdmp_eval.sh` script, which runs the entire evaluation with one command. Running all experiments is expected to take around 17 compute-hours.

1) *Experiment (E1): Security* [5 human-minutes + 20 compute-minutes]: verifies non-interference of all programs on all prototypes, checking attacker-observable and internal DMP signals (the latter for leakage tests only; see [Section VIII-B1](#)).

[Execution] Run `source .venv/bin/activate && python3 security_eval.py ; deactivate`.

[Results] The `results/security_evaluation/` directory contains a table marking each program as secure or leaking, per DMP, for each prototype. The outcome should match `expected_results/` and correspond to [Table IV](#).

2) *Experiment (E2): HACL* cryptographic performance* [5 human-minutes + 2 compute-minutes]: measures the cycle counts of the five HACL* primitives on every prototype and verifies that the secret annotations do not spill.

[Execution] Run the command `python3 performance_eval_hacl_crypto.py` followed by `./check_crypto_partition.sh`, which prints no exceptions if the partitioning is correct.

[Results] `results/performance_evaluation/hacl_crypto/` reports the cycle counts and relative overheads of every prototype, supporting (C2) and corresponding to [Figure 5, Section VIII-B2](#).

3) *Experiment (E3): Work benchmarks* [5 human-minutes + 2 compute-hours]: measures Embench, CoreMark, AoP, and SpMV on every prototype.

[Execution] Run `./performance_eval_embench.sh`, `./performance_eval_coremark.sh`, `./performance_eval_array_of_pointers.sh` and `./performance_eval_spmv.sh`.

[Results] The corresponding subdirectories of `results/performance_evaluation/` support (C2) and correspond to [Figure 6, Section VIII-B2](#).

4) *Experiment (E4): DMP-Bench* [5 human-minutes + 6 compute-hours]: measures the 20 work/crypto combinations at four work percentages, for each DMP on every prototype.

[Execution] Run the command `python3 performance_eval_dmp_bench.py`.

[Results] `results/performance_evaluation/dmp_bench/` supports (C2) and corresponds to [Figure 7](#).

5) *Experiment (E5): Hardware cost* [5 human-minutes + 8–24 compute-hours, depending on RAM]: synthesizes all prototypes with EVAL-HD, reporting area and critical path.

[Execution] Run `./hardware_eval.sh`.

[Results] `results/hardware_evaluation/` supports (C3) and corresponds to [Table V](#); this is the only experiment whose numbers fluctuate between runs. These fluctuations are negligible, as they occur only in the fourth decimal place (i.e., at the 10^{-4} mm² level).