PROBE+DETECT+MITIGATE (PDM): Enabling Cloud Tenants to Self-Defend against Microarchitectural Attacks

Arash Daneshmand[2,1]✉, Hugo Kermabon-Bobinnec[1], Lingyu Wang[2,1]✉, Makan Pourzandi[3], Suryadipta Majumdar[1], and Yosr Jarraya[3]

[1]Security Research Centre (SRC), Concordia University, Montreal, QC, Canada
[2]School of Engineering, University of British Columbia Okanagan, Kelowna, BC, Canada
[3]Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada
{hugo.kermabonbobinnec,arash.daneshmand, suryadipta.majumdar}@concordia.ca
lingyu.wang@ubc.ca
{makan.pourzandi,yosr.jarraya}@ericsson.com

Abstract

Microarchitectural attacks represent a critical security concern in public cloud environments, as they can cause information leakage between cloud tenants with conflicting interests. Existing solutions usually require provider-level resources, such as hardware performance counters or host processes, which may be inaccessible to cloud tenants. The lack of awareness among cloud tenants may persuade cloud providers to postpone the deployment of vendor patches, as evidenced by patched-yet-active threats, such as PRIME+PROBE and Spectre variants. In this paper, we propose PDM, a solution that enables cloud tenants to independently detect and mitigate microarchitectural attacks without providers' help. First, PDM introduces tenant-based detection based on an interesting observation, i.e., probing the memory space of victim applications using the popular FLUSH+RELOAD attack technique can actually enable detection. Second, PDM achieves efficient tenant-based mitigation by selectively triggering obfuscation and in-memory encryption techniques upon detection. Third, we tackle several key challenges including (i) attacks not involving evictions (e.g., Spectre), (ii) the need for source code or binary instrumentation, (iii) benign noises from the victim or co-resident tenants, and (iv) the tradeoff between accuracy, delay, and overhead. Our experiments show that PDM allows tenants to detect and mitigate various microarchitectural attacks, including PRIME+PROBE and Spectre, in an accurate (e.g., $\geq$99.72% TPR and $\leq$0.13% FPR on our testbed, and $\geq$98.63% TPR and $\leq$0.83% FPR on AWS Fargate), timely (e.g., 7ms lead time for triggering mitigation), efficient (e.g., $\leq$2.47% overhead on SPEC CPU 2017), and robust (against both noises and evasive attacks) manner.

1 Introduction

Microarchitectural attacks exploit the shared CPU cache and insecure hardware optimization to uncover private information, through either analyzing the victim's memory access patterns [30, 69, 86] or directly accessing confidential data [48, 56]. Such attacks are particularly concerning inside a multi-tenant public cloud environment where different tenants with conflicting interests may potentially share the same physical host. The shared resources [42, 44, 58, 89, 92] and/or hardware vulnerabilities [8, 74, 75, 81] may enable an attacker to steal sensitive information about other co-resident tenants, such as their passwords and cryptographic keys.

While cloud tenants are responsible for securing their own data (e.g., "security is a shared responsibility between the provider and the tenant" [5]), they may feel helpless when it comes to defending against microarchitectural attacks. As illustrated on the left side of Fig. 1, cloud tenants typically lack direct access to the host or hardware-level information, such as hardware performance counters (HPCs)[1] and system processes, which is usually necessary for existing solutions to detect microarchitectural attacks [12, 18, 30, 32, 53, 75, 80]. On the other hand, preventing microarchitectural attacks with existing code/binary hardening techniques, such as constant time code [41], inserting memory fences [15, 61, 66, 78] or in-memory encryption [70, 83], may incur significant overhead if applied non-selectively, while page table modification solutions like marking pages as uncacheable [73] or using execute-only memory [35] typically require access to the host or hypervisor, which again may be infeasible for the tenants.

The lack of means for dealing with microarchitectural attacks may cause cloud tenants to lose interest and decide to simply live with what providers offer, e.g., dedicated L1 and L2 cache in EC2 [4] and dedicated EC2 instances [6]. However, these either only provide limited protection against microarchitectural attacks [4], come with additional costs [6], or become infeasible altogether (e.g., in serverless platforms like AWS Fargate [7, 81]). Moreover, knowing such a lack of interest or awareness among cloud tenants, cloud providers may be less motivated to deploy all the

✉ Corresponding authors.
Arash Daneshmand is currently at UBCO; this work was done at Concordia.

[1]For instance, in Amazon AWS, HPCs are only accessible in *dedicated* instances (regardless of the instance size), and are disabled on *shared* instances due to their security implications [28].

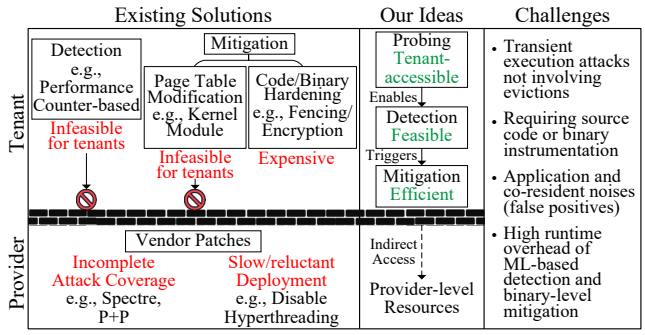


Figure 1: Motivation and Our Ideas

latest vendor patches since these are known to result in considerable performance penalty [10]. In addition, the timely development of vendor patches also proves to be challenging [61, 63], and each resolved vulnerability usually leads to the discovery of new ones. As a result, microarchitectural attacks remain a critical security concern in today's cloud environments, as evidenced by patched-yet-active threats like PRIME+PROBE on shared last-level caches [42, 58, 92] and Spectre [9, 61, 63, 65, 75, 81, 84].

Therefore, there is a need for enabling cloud tenants to defend themselves against microarchitectural attacks. This can (i) raise tenants' awareness of microarchitectural attacks, and draw further public attention to this active threat; (ii) pressure vendors and providers to more timely develop and deploy vendor patches; (iii) provide an additional layer of protection that can work in tandem with existing provider-level detection and mitigation solutions.

Our Ideas. As illustrated in the middle of Fig. 1, our main ideas are twofold. First, we can enable a cloud tenant to independently detect microarchitectural attacks by applying similar probing techniques used in such attacks. For instance, the popular FLUSH+RELOAD [86] attack involves repetitively flushing the cache, whose effect may be directly observed as slow memory reads by a tenant who probes (accesses) the memory. Second, we can then leverage this newly gained detection capability to selectively trigger attack mitigation only when a potential attack is detected. This can avoid the undue overhead incurred by constantly applying tenant-level mitigation solutions such as obfuscation and encryption.

Challenges. There are several challenges toward realizing those ideas. First, the previous idea of monitoring unexpected cache misses can only detect malicious evictions, but many microarchitectural attacks do not involve a malicious eviction, e.g., Spectre [48] variants. Second, injecting the probing procedure into the victim's process may require modifying the victim's source code (impractical) or binary instrumentation (expensive [21]). Third, as microarchitectural attacks occur at the hardware level, a tenant-level solution is naturally subject to various background noises (e.g., those coming from the application workload or co-resident tenants sharing the same CPU), which can cause significant false positives in detection and mitigation. Finally, both machine learning (ML)-based detection and binary-level mitigation may become too expensive in this context when applied at runtime.

To address those challenges, we introduce PDM, a solution for cloud tenants to defend themselves against microarchitectural attacks without cloud providers' help. First, we develop a series of probing schemes to enable cloud tenants to independently detect various types of microarchitectural attacks (Section 3.1). Second, we integrate PDM as a thread into the victim's binary to avoid the need for source code modification or binary instrumentation, and enhance PDM's robustness against potential noises through feature engineering (Section 3.2). Third, we balance between the overhead and accuracy through selectively triggering different mitigation solutions, i.e., a fast model triggers immediate and non-disruptive mitigation, and a slow model verifies the attack, and then either deactivates the mitigation (false positive), or triggers more drastic mitigation (confirmed attack) (Section 3.3).

The following summarizes our main contributions:

- We design PDM as a true tenant-based solution that requires no special support or access from the cloud provider. This enables cloud tenants to independently detect and mitigate the threat, giving them more control and an additional layer of protection. This can also raise cloud tenants' awareness of microarchitectural attacks, and attract more public attention to this important threat.
- We tackle several key challenges in realizing PDM: (i) covering various types of microarchitectural attacks, (ii) avoiding the need for source code modification or binary instrumentation, (iii) handling noises from various sources, and (iv) balancing between the overhead and accuracy.
- We implement and evaluate PDM using both an in-house multi-tenant cloud testbed (using KVM-based virtualization and Kubernetes-based containerization) and a public cloud (AWS Fargate), against various proof-of-concept microarchitectural attacks, diverse sources of noises, and slow evasive attacks. All our artifacts are publicly available.[2]
- Our evaluation demonstrates that PDM is effective against various attacks including PRIME+PROBE and Spectre, with high accuracy (e.g., $\geq$99.72% TPR and $\leq$0.13% FPR on our testbed, and $\geq$98.63% TPR and $\leq$0.83% FPR on AWS Fargate), short delay (e.g., 7ms lead time for triggering mitigation), low overhead (e.g., 2.47% overhead on SPEC CPU 2017), and reasonable robustness (against both noises and evasive attacks). Finally, in comparison to state-of-the-art solutions, PDM achieves (i) comparable detection accuracy, despite its lack of access to provider-level resources, and (ii) far less mitigation overhead (negligible when idle).

2 Preliminaries

This section gives background and defines our threat model.

[2] https://github.com/arashd021/pdm/

2.1 Microarchitectural Attacks

Cache Side-Channel Attacks. Those attacks exploit the shared CPU cache by measuring memory access latencies to determine whether a target cache line was brought into the cache by the victim. This can expose the victim's secret-dependent memory access patterns, and potentially enable the recovery of sensitive information such as cryptographic keys. These attacks typically begin by evicting target addresses to clear data previously cached by other processes. FLUSH+RELOAD [86] and FLUSH+FLUSH [30] use the *clflush* instruction to directly flush targets. EVICT+RELOAD [31] fills the cache set with congruent addresses to evict targets, while PRIME+PROBE variants [22, 42, 58, 69, 71, 92] do the same without shared memory. Next, the attacker infers victim access patterns: in FLUSH+RELOAD and EVICT+RELOAD, fast reloads indicate cache hits from the victim; in FLUSH+FLUSH, slow flushes indicate such hits; in PRIME+PROBE, slow accesses indicate victim-induced cache misses. RELOAD+REFRESH [13] exploits replacement policies instead of actively evicting victim's data. The attacker refreshes the replacement ages so that only their own line gets evicted if the victim touches the target. The victim's data will be kept in the last-level cache but is evicted from L1 and L2 [85].

Disabling memory sharing can mitigate some of those attacks, but leads to increased memory usage and performance degradation, and is ineffective against PRIME+PROBE variants that do not require shared memory [22, 42, 58, 69, 71, 92]. Randomized secure caches [24] can mitigate the latter, but require hardware changes. Last-level cache partitioning can also achieve this, though at the expense of performance [57].

Transient Execution Attacks. Those attacks exploit modern CPU optimizations that allow instructions to be executed speculatively or out-of-order. Specifically, mispredictions lead to *transient execution* whose effect is rolled back, while the microarchitectural effects (e.g., inside the cache) remain, allowing attackers to extract victim's data. For instance, in Spectre variants, an attacker having access to presumably safe segments of the program can divert normal execution toward a series of instructions (i.e., gadgets) that read secret data. This is achieved by forcing conditional branch misprediction (PHT [48]), indirect branch misprediction (BTB [48]), return address misprediction (RSB [49, 60]), or memory read misprediction (STL [34]). The data being read can then be extracted using a cache covert channel. Since Spectre [48] and Meltdown [56] appeared, there has been a constant stream of similar attacks [9, 14, 60, 74, 75, 77, 84].

Transient execution attacks have traditionally been kept under embargo until mitigated by the vendor. Examples of vendor mitigations include KPTI [29] for Meltdown, and preventing speculation with memory fence instructions [66] for Spectre. However, most such solutions are either incomplete (leaving some attack vectors unprotected) or too aggressive (causing performance degradation) [14, 15, 61, 63]. Finally, new speculative execution paths and branch target injection techniques are regularly discovered by researchers to defeat existing mitigation solutions [65, 84].

2.2 Threat Model

Following the literature on microarchitectural attacks, the attacker is assumed to be a regular cloud tenant who is co-resident with the victim on the same physical host [91]. The co-residency leads to sharing the CPU, memory, and last-level cache (LLC) between the victim and attacker. The attacker can only execute unprivileged code within his/her own cloud instance, with no privileged instructions or elevated access. The cloud provider is assumed to have disabled direct access by tenants to host or hardware-level information, including HPCs due to their own security implications. The tenants are assumed to trust the provider, but are also aware of the risks that the provider may struggle to keep up with the constant stream of newer attack variants, or be reluctant to deploy all the mitigation solutions, e.g., due to the implied overhead of disabling resource sharing or hardware optimization techniques. Finally, we only consider realistic cloud environments which are naturally noisy so a data leakage can only occur at a relatively low rate, e.g., real-world Spectre-PHT leaks at 41B/s [48, 73], and cross-VM FLUSH+RELOAD [44] and PRIME+PROBE [42, 58, 92] can both take seconds or minutes.

Table 1: PDM's Security Coverage

PDM's Security Coverage	Attack Category	Impact on the Victim's Cache Occupancy State
Fully Covered (Detection/Mitigation)	Eviction-based Cache Attacks [30, 42, 58, 69, 86, 92]	Data evicted out of all levels of cache
	Eviction-based Cache Attacks (non-inclusive cache) [72]	Data evicted (demoted) from L1 to L3
	Cache Attacks Exploiting Replacement Policies [13]	Data's replacement ages are refreshed (evicted from L1 and L2)
	Spectre-type Data Leakage [9, 14, 34, 48, 60, 84]	Data cached due to attacker's speculative access
Partially Covered (Detection/No Mitigation)	Non-secrecy Misuse of Spectre (e.g., Control Flow Integrity [25])	Data cached due to attacker's speculative access
	Memory DoS [88]	Data evicted out of all levels of cache
Not Covered (Out of Scope)	MDS [77], L1TF [76], Rowhammer [47], TLB [27], and Port Contention [3], etc.	No impact

Security Coverage. First, Table 1 shows the attacks affecting the victim's cache occupancy state (i.e., data distribution within the cache-memory hierarchy) that PDM can detect and mitigate *(fully covered)*. Specifically, eviction-based cache attacks increase cache misses across different levels of cache, while Spectre-type data leakages increase cache hits by pulling victim data into L1 through speculative accesses. Since PDM targets the speculative-access phase (not the covert channel data exfiltration), coverage remains effective

against variants using other types of covert channel [11, 20]. Second, the attacks leveraging Spectre for intents other than direct data leakage and memory DoS are detected but not mitigated *(partially covered)* by PDM, since they target integrity or availability rather than confidentiality. Finally, the attacks that do not alter the victim's cache occupancy state are out of scope *(not covered)*.

To place such attack coverage in context, we examined 169 microarchitectural attack papers published over the past decade at major security venues (IEEE S&P, USENIX Security, ACM CCS, and NDSS). We found that 28% target Spectre and 37% target cache-based side-channels, indicating that PDM covers a substantial portion of the common attacks despite the fact that it only has access to tenant-level resources.

3 Methodology

We provide an overview before detailing PDM's components.

Overview. As shown in Fig. 2, PDM consists of three major components, namely, *probing*, *detection*, and *mitigation*. First, the probing component monitors the victim's memory space for potential attacks, and captures unexpected cache hits or misses as features for detection. Second, the detection component has three sub-components: (i) *Injection* extracts secret-dependent addresses from the victim process and spawns a new thread to gain the control and visibility necessary for probing, detection, and mitigation. (ii) *Detection engine* first organizes the detection features received from the probing component as multivariate time series, and then performs a two-stage detection process to handle false positives, which is conditionally triggered using an inference switch when secrets were accessed or evicted. (iii) *Noise handling* deals with both victim noise and co-resident noise by collecting extra detection features, such as the load on cache and the victim's secret access frequency. Finally, upon detecting an attack, the mitigation component is activated to mitigate the attack through cache obfuscation or in-memory encryption.

3.1 Probing

This section designs a series of probing schemes to monitor the victim's memory space and collect data for detection.

Scheme 1: Access+Reload. As mentioned in Section 1, a straightforward way to detect microarchitectural attacks is to focus on abnormal cache misses caused by the attacker's eviction operations. Attackers employ such eviction operations to clear footprints of previous access patterns, such that the victim's subsequent access patterns can be inferred through cache hits. When an attacker evicts [31], flushes [30, 86], or primes [42, 58, 69, 92] the victim's data out of the cache, we can detect the attack by probing (accessing) the same memory addresses, since the access will be slow due to the cache misses caused by the attack.

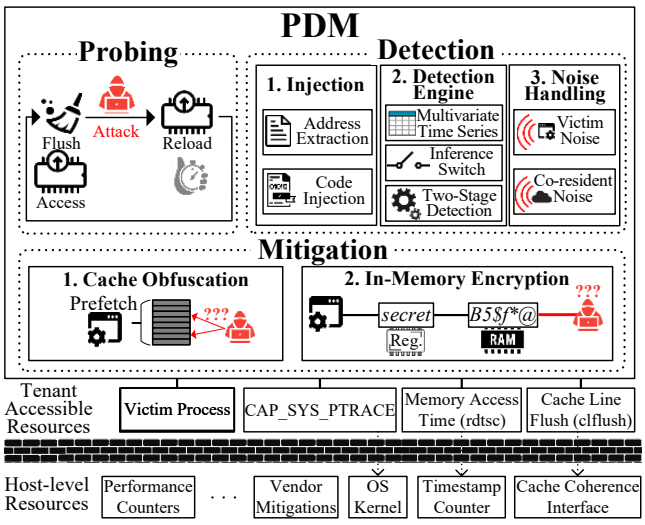


Figure 2: Overview of PDM

However, a challenge arises here since we typically need to probe not just one memory address, but a range of addresses, namely, *probing range*. During the delay before we can return to probe the same address again, noises may arise (e.g., from benign applications or hardware prefetchers) to cause false detection results. To address this, we learn from the attacker, i.e., we *clear any existing footprints*, by adding another access operation (*Access*) before our probing access (*Reload*), with a wait interval in-between as the time window for detection.

Specifically, as Fig. 3 (top-left) shows, each probing cycle of Scheme 1 starts with a memory access operation (*Access Memory[i]* in the figure). It then waits for a given time interval (*Wait*), before performing the second access operation at the same address (*Reload Memory[i]*). Next, it measures the reload time (*Measure Time*), and compares the result to a pre-defined threshold (*Slow?*). If the comparison indicates a cache miss, a detection counter[3] is incremented (*Miss++*); otherwise, this probing cycle ends. Fig. 3 (middle-left) illustrates how this scheme can detect the eviction step (*Flush*) of cache side-channel attacks, i.e., the flush will be indicated by a slow reload.

Scheme 2: Flush+Reload. As Scheme 1 only monitors for cache misses, it cannot detect transient execution attacks which rely on speculative memory access instead of evictions (e.g., Spectre [48]). To address this, we borrow the probing technique used in the popular FLUSH+RELOAD [86] attack for detection, i.e., we first flush the memory (*Flush*) before accessing it (*Reload*), with a wait interval in-between as the window for detection. Any speculative memory accesses occurring inside this window can then be detected as unexpected cache hits. Specifically, as Fig. 3 (top-right) shows, Scheme 2 starts with a memory flushing operation (*Flush Memory[i]*),

[3]To monitor variations in the victim's cache occupancy state across all levels of the cache-memory hierarchy, we use four detection counters with different thresholds for each scheme (not shown in the figure). These will then be used as features for our detection classifier, as detailed in Section 3.2.2.

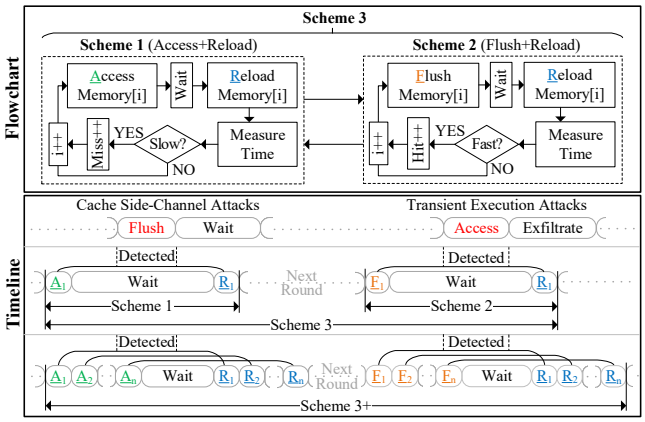


Figure 3: PDM Probing Schemes (Flowchart and Timeline)

waits for a given time interval (*Wait*), and finally performs an access operation on the same address (*Reload Memory[i]*). It then measures the access time (*Measure Time*) and compares the result to the cache hit threshold (*Fast?*) to increment the detection counter (*Hit++*). Fig. 3 (middle-right) illustrates how a transient execution attack's unauthorized access can be detected under this scheme as a fast reload.

Scheme 3: Scheme 1+Scheme 2. As both Scheme 1 and Scheme 2 can only detect a subset of attacks, Scheme 3 naturally combines those two building block schemes. Specifically, as Fig. 3 (top) shows, Scheme 3 basically alternates between the two previous schemes for the probing of each address, i.e., it first performs Scheme 1 (*Access+Reload[i]*) on every address in the probing range, and then it switches to Scheme 2 (*Flush+Reload[i]*). Fig. 3 (middle) illustrates how Scheme 3 can detect both types of attacks.

Scheme 3+: Scheme 3 Multiplexed. Although Scheme 3 is sufficient for most use cases (as demonstrated in Section 4.3.2), we further extend it through multiplexing the probing during wait intervals. Specifically, as illustrated in Fig. 3 (bottom), Scheme 3+ leverages each wait interval to sequentially perform the first probing operation (i.e., access or flush) on multiple addresses until the total elapsed time since the first operation matches the original wait interval. It then performs the reload (second access operation) on all those addresses to complete the probing cycle. Those addresses are selected using a 4KiB stride to avoid prefetching and ensure spatial independence across probing operations. This multiplexing design allows Scheme 3+ to cover up to 512KiB using a single probing thread (if the required probing range is insufficient to fill the entire wait interval, the thread sleeps for the remaining duration). This scheme gives tenants more flexibility in covering a larger probing range with slightly more CPU usage, as will be demonstrated in Section 4.3.2.

3.2 Detection

This section details how PDM injects itself into the victim process to detect attacks while handling noises.

3.2.1 Injection

The injection of PDM takes two steps.

Address Extraction. PDM applies existing approaches to locate both the secret-dependent memory access instructions (for cache side-channel attacks) and the secrets themselves (for transient execution attacks). First, the former can be achieved through manually profiling a cryptographic library, i.e., triggering the encryption while probing memory addresses using FLUSH+RELOAD (addresses showing a high cache hit rate may be secret-dependent [31]). However, since this approach can be time-consuming and lead to over-annotation, we instead rely on existing automated cache side-channel vulnerability detectors [79, 82, 87] to scan cryptographic implementations for identifying suspicious *leak sites* (i.e., secret-dependent memory access instructions that actually allow leaking secrets). Second, to locate the secrets and track their propagation, we leverage dynamic binary instrumentation (DBI) based on Intel Pin [40] for manually annotating secret offsets within the binary, and apply dynamic taint analysis (DTA) based on `libdft` [46] for tracking secret propagation [83].

Address extraction is performed offline, when instrumentation overhead is acceptable. Extracted addresses are stored as offsets within the victim binary and its shared libraries, ensuring resilience against ASLR. These offsets define PDM's *probing range* (detailed in Section 3.1).

Code Injection. Next, PDM obtains access to the addresses in the probing range. Dynamic binary instrumentation can achieve this but is expensive. On the other hand, cross-process memory access is not feasible since each process is assigned a dedicated virtual address space. Using the `mmap` syscall to map a read-only copy of the victim binary [31] can overcome this limitation, but it only provides visibility into static addresses.

Instead, we take the opposite approach, i.e., making the victim process load PDM into its memory on our behalf, and run PDM as a thread. This approach has two benefits: (i) PDM and the victim share the exact same memory (including dynamically allocated addresses); (ii) running PDM as a separate thread minimizes performance impact (Section 4.3.2).

Specifically, we compile PDM as a shared library (`.so` object) that will spawn a new thread once loaded (using `__attribute__ ((constructor))`), pinned to the same core as the victim. We inject the library into the victim in two ways (for victim applications given as binaries and running processes, respectively): (i) using `LD_PRELOAD` to load the library at startup; (ii) injecting the library at runtime using *ptrace* [51]. The runtime injection approach starts by attaching to the victim process and locating the `__libc_dlopen_mode` function. It then injects a small payload at the `libc` entrypoint to trigger `dlopen`, and updates the instruction pointer to the injected code. When execution resumes, the victim loads PDM thread as a shared library.

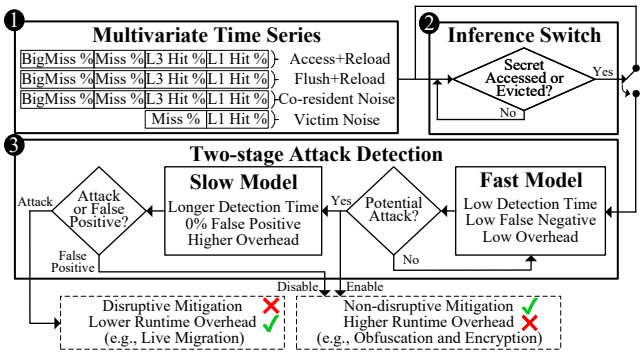


Figure 4: PDM Detection Engine

3.2.2 Detection Engine

The PDM detection engine organizes its detection features as a multivariate time series, and employs an inference switch to trigger a two-stage detection process for balancing between accuracy, overhead, and lead time.

Multivariate Time Series. We employ a multivariate time series to enable the analysis of dependencies between all the features, including detection counters (Section 3.1) and noise-handling features (Section 3.2.3). Specifically, as illustrated in Fig. 4 (upper left), the detection counters *Access+Reload* and *Flush+Reload* capture the ratio of probes that hit each level of the cache-memory hierarchy, separated with different thresholds (*BigMiss* refers to latencies exceeding 450 cycles, which typically occurs due to cache coherence, cross-slice L3 access delays, interconnect contention, TLB misses, and page table walks [39]). Other features collected for noise handling will be detailed in Section 3.2.3.

To enhance the accuracy of the classifier and mitigate unwanted fluctuations in the time series data, we compute the moving average and moving standard deviation over all the features and then normalize them using the RobustScaler technique. The constructed data point is then given to the encoder layer of a transformer-based classifier to make the detection decision. Since PDM is designed as an online security tool, one requirement here is to ensure that the classifier remains lightweight and efficient enough for online inference. To achieve this, we minimize the size (depth, number of layers, number of heads) of our classifier while maintaining sufficient accuracy, using the *Optuna* [2] parameter tuning library. We then import the classifier into the PDM thread using the *ONNX Runtime* [68], which is configured to only spawn one extra thread to minimize the overhead.

Inference Switch. Although our classifier is lightweight and executed in a single *ONNX* thread, constantly performing ML inference would still introduce unnecessary overhead and delay. To avoid this, we introduce an inference switch that conditionally triggers ML inference depending on whether the secrets have been accessed or evicted. As illustrated in Fig. 4 (upper right), the inference switch is turned off, i.e., it will not trigger the detection process, when both of the following conditions are true: (i) all *Access+Reload* probes return L1 cache hits; (ii) all *Flush+Reload* probes return L3 cache misses. Those conditions together indicate the absence of any ongoing attack, and hence performing ML inference would cause unnecessary overhead. Otherwise (if at least one of those conditions is false), the inference switch is turned on to pass detection features to the fast model.

Two-Stage Detection. Another challenge is to balance between the detection lead time and accuracy. Since real-world microarchitectural attacks may cause data leakage in a matter of seconds [42, 44, 48, 58, 73, 92], a shorter detection lead time is necessary to activate the mitigation earlier. However, this also requires more frequent detection decisions to be made based on less input data, which naturally results in more false positives and unnecessary mitigation. To address this challenge, we introduce a two-stage detection process composed of both a fast detection classifier and a slower but more accurate classifier, as illustrated in Fig. 4 (middle).

The fast classifier is tuned for the least amount of false negatives (i.e., not missing any attacks), while the slow classifier is tuned for the least amount of false positives (i.e., verifying the former's detection results), as demonstrated in Section 4.3. Mitigation is activated as soon as the fast classifier detects an attack (for a shorter detection lead time). If the slow classifier identifies this as a false positive, the mitigation is deactivated on the affected addresses (to reduce unnecessary overhead). Otherwise (if the attack is confirmed), the mitigation stays active (to provide additional response time), while the tenants are alerted to migrate their workloads and the cloud provider is notified to apply additional protection.

3.2.3 Noise Handling

Victim Noise. As PDM probes memory addresses, concurrent victim accesses may shift the baseline, i.e., a benign increase in cache hits due to normal workload can trigger false positives or hide attacks. To address this, PDM introduces a new component to identify legitimate victim accesses. Specifically, the offsets at which the process is accessing secrets are identified using dynamic binary instrumentation [40] and a dedicated probing thread is spawned to monitor those offsets using probing Scheme 2 (Section 3.1). This enables the collection of extra features, as shown in Fig. 4 (*Victim Noise*), that determine whether the victim has executed relevant code before accessing secrets. In addition, PDM dynamically adjusts its *Flush* and *Reload* wait interval based on the victim's access frequency to preserve probing resolution.

Co-resident Noise. In multi-tenant clouds, even with isolation measures such as Linux `cgroups`, Amazon Firecracker [81], and dedicated L1/L2 caches [4], tenants on the same host still share underlying resources like the CPU and last-level cache. Consequently, cache-intensive workloads from co-resident tenants and from the victim's other applications can both generate noise. Such co-resident noise has a similar effect as victim noise in hiding attack patterns or generating false

positives. Unlike victim noise, co-resident noise causes an increase in cache misses instead of hits. Nonetheless, the access patterns of co-resident applications are inherently different from those of eviction-based attacks, i.e., eviction-based attacks mostly follow a repetitive and predictable access pattern and affect specific cache lines, while cache intensive applications generally exhibit more temporal and spatial variability.

Based on such observations, PDM introduces a noise handling thread to detect cache-intensive workloads, following a similar technique used by the PRIME+PROBE attacks [42, 58, 69, 92] to probe addresses mapped to different L3 cache sets. Since physical addresses are evenly distributed across slices [62], a single probe per set can serve as a representative sample of L3 cache and enables detection of heavy system load. The collected features are appended to the multivariate time series (Fig. 4), while PDM dynamically adjusts its *Access* and *Reload* wait interval based on noise intensity.

3.3 Mitigation

This section describes how PDM achieves efficient attack mitigation by only triggering mitigation solutions upon detection. PDM leverages several user-space mitigation approaches, including obfuscation (for cache side-channel attacks), in-memory encryption (for transient execution attacks), and migration (for confirmed attacks). Those solutions are chosen to meet two unique requirements: (i) deployable at the tenant level, and (ii) *non-disruptive* in the sense that false positives in detection will not disrupt the victim application.

Mitigation for Cache Side-Channel Attacks. Since those attacks aim at inferring the victim's memory access patterns, a well-known mitigation solution is to obfuscate such patterns to increase the noise observed by attackers, and hence reduce their attack success rates [23, 64]. PDM adopts a two-stage obfuscation approach. First, the access and flushing operations performed by PDM during probing already introduce an obfuscation effect. Second, upon detecting an attack, PDM intensifies its obfuscation effect by aggressively prefetching the memory pages containing the addresses under attack, following existing approaches [23, 64]. This creates overwhelming noises to significantly reduce the attacker's ability to infer the victim's memory access patterns,[4] while being non-disruptive (the prefetching actually enhances the application's performance by reducing its cache misses [64]).

Mitigation for Transient Execution Attacks. To mitigate transient execution attacks at the tenant level, we cannot employ existing solutions like preventing unsafe speculative execution [15, 61, 66, 78] (which requires offline code analysis), marking memory pages *uncacheable* [73], or using execute-only memory [35] (which require access to the host

[4]While such security-by-obscurity mitigation approaches can eventually be defeated by attackers, e.g., using better signal processing techniques, PDM only relies on those approaches for a short time before its slow classifier can confirm the attack.

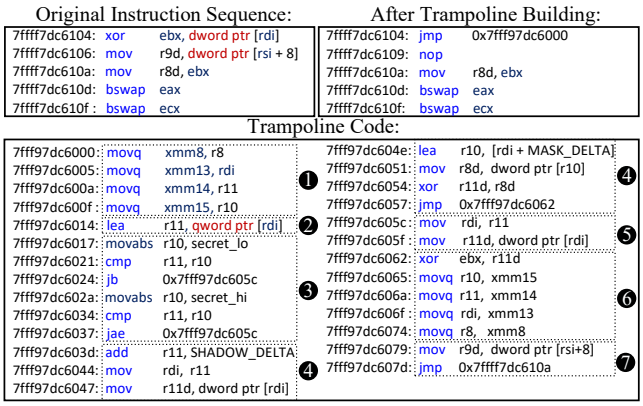


Figure 5: PDM Mitigation: Trampoline Building

or hypervisor, respectively). Instead, PDM leverages the *in-memory encryption* approach [70, 83]. Specifically, upon attack detection, PDM immediately encrypts the memory content stored at the affected addresses using XOR masking with a cryptographically secure pseudorandom mask. It then intercepts memory reads (and writes) that involve under-attack addresses, and decrypts (and encrypts) the data on the fly. Applying this solution to only a handful of victim memory addresses allows PDM to keep a low overhead on the victim process. This solution is also non-disruptive since the only impact to a mistakenly detected application is a slight delay (which will be evaluated in Section 4.4).

To avoid the overhead of instrumenting every memory access at runtime or locating all secret-accessing instructions through static analysis, PDM makes the application *self-declare* instructions accessing secrets and *self-modify* them to integrate the on-the-fly encryption and decryption logic. Specifically, we leverage the `mprotect` system call to mark pages containing secrets as `PROT_NONE` such that any access to these pages triggers a `SIGSEGV` signal [16]. This signal is then handled using a custom signal handler installed in the application to build a trampoline for each faulting instruction using inline binary rewriting [83]. To allow the trampoline to access the encrypted secrets without lifting the page protection, a `RW` encrypted shadow copy of the secret is allocated, and any secret access in the trampoline is redirected to the corresponding offset in the shadow page.

The trampoline is built by allocating a `RW` memory within ±1.5 GiB of the faulting instruction pointer (`RIP`), enabling 5-byte `jmp rel32` branches that are limited to ±2 GiB. If a nearby allocation is not possible, a generic allocation is used anywhere in the address space, requiring a 13-byte absolute indirect jump (e.g., `mov r11, imm64; jmp r11`). Fig. 5 illustrates how PDM builds a trampoline in the following seven steps for an `xor` load instruction accessing a secret.

First, the trampoline saves the scratch registers (`r8`, `r10`, `r11`, and `rdi`) into `xmm` registers to avoid clobbering (step ❶). The faulting address cannot be hard-coded into the trampoline since the accessed address may vary across runs. Instead, a

`lea` instruction is generated to compute the effective address at runtime (step ❷). If the operand is RIP-relative, the effective address is derived from a constant offset. Otherwise, the original addressing form is reconstructed by emitting the required `REX`, `ModR/M`, `SIB`, and displacement bytes to replicate the CPU's own decoding.

Next, in step ❸, the effective address is emitted unchanged if it falls outside a secret page (jump to step ❺). Otherwise, the corresponding shadow address is extracted by generating an `add` instruction to apply the offset between the secret and the shadow page. For loads, the encrypted shadow value is fetched into a scratch register, unmasked, and then placed in the destination register (step ❹). For stores, a fresh mask is generated using an AES-based PRNG [83], the value is masked in a scratch register with this fresh mask, and the resulting ciphertext is written to the shadow address. Scratch registers are restored after the instruction (step ❻). To replace the faulting instruction with the trampoline, we generate either a 5-byte `jmp rel32` or a 13-byte indirect `jmp` depending on the trampoline distance. As shown in Fig. 5 (top left), since both `xor` and `mov` are smaller than 5 bytes, both (6 bytes in total) must be *stolen* [83] to insert a 5-byte jump. The stolen `mov` instruction is copied verbatim into the trampoline, followed by a jump back to the original instruction sequence (step ❼). Finally, the faulting instruction is replaced with the jump; residual bytes are padded with `nops`; the instruction cache is flushed, and the trampoline page is set to `RX`.

During a transient execution attack, the CPU does not immediately deliver exceptions and instead suppresses them while speculating. Therefore, any speculative access to the protected memory pages by the attacker will not invoke the signal handler, and no trampoline is built. As a result, the attacker will not be able to invoke the decryption logic, and any transiently received memory content will remain encrypted, which mitigates the attack. On the other hand, any legitimate memory accesses by the victim will invoke the handler and trampoline building for decryption, ensuring non-disruptive mitigation. Appendix A provides further details.

Mitigation for Confirmed Attacks. As mentioned in Section 3.2.2, once the slow classifier of PDM has confirmed a detected attack, more drastic mitigation measures such as migrating the workload and notifying the cloud provider could be taken by the tenants. A challenge is that major cloud providers like Amazon AWS, Google Cloud, and Microsoft Azure still lack support for *tenant-initiated* live migration. To overcome this limitation, tenants can utilize user-space checkpoint/restore tools such as CRIU [19] to migrate containers while maintaining their network connection states. Since such migration could take some time (e.g., CRIU takes around 3.5 seconds to migrate a container with 11 tasks [19]), the aforementioned intermediate mitigation approaches are essential for providing security during the transition period.

4 Evaluation

We evaluate PDM to answer three research questions:

RQ1: What is PDM's detection (i) accuracy (in our testbed, and in a real cloud deployment) and (ii) overhead (w.r.t. application response time and resource consumption), and (iii) how does it compare to existing detection solutions?

RQ2: How effective is PDM in mitigating (i) cache side-channel attacks and (ii) transient execution attacks, and (iii) how does it compare to existing mitigation solutions?

RQ3: How robust is PDM in handling (i) victim noise, (ii) co-resident noise, and (iii) evasive attacks?

4.1 Evaluation Environment

Testbed. We evaluate PDM using an in-house testbed (where more attacks can be safely performed). Our testbed utilizes a physical server with an Intel Xeon Gold 5120 (Skylake) CPU with 19.25MB L3 cache, divided into 14 slices (one per core), and 1MB L2 cache per core.[5]

Real Cloud. We also evaluate PDM using a Kubernetes cluster deployed on Amazon EKS based on Amazon Fargate[6] where PDM is integrated into a container with one vCPU and 1GB of memory. PDM's deployment in AWS turns out to be straightforward. The only required Linux capability, `CAP_SYS_PTRACE`, is already enabled on Fargate containers [7]. This confirms the practicality of deploying PDM as a tenant-based solution (in contrast, HPCs are inaccessible in AWS instances on a shared host [28]).

Attacks. We implemented both cache side-channel attacks (e.g., FLUSH+RELOAD [86], FLUSH+FLUSH [30], and PRIME+PROBE [42, 58, 92]), and transient execution attacks (e.g., Spectre [48] variants PHT, BTB, STL, and RSB). We leveraged open-source implementations [1, 36, 37] for the former, and proof-of-concept codes [14, 38] and Google SafeSide [26] for the latter.

Victim Applications. Following [12, 18, 32, 53, 75, 80], our victim applications include the vulnerable AES T-table implementation in *OpenSSL* [30, 42, 44] and RSA in *Libgcrypt* [86] for cache side-channel attacks, and *Nginx*, *Redis*, and *OpenSSH* for transient execution attacks.

Datasets. We train PDM's detection classifier only once for each environment. The training lasts two hours during which the aforementioned attacks are executed in a random order, and for a random number of iterations, while we vary the attack speed (to cover slowed evasive attacks), the victim's workload, and the co-resident noises. The trained models are then evaluated *online* under different scenarios in each experiment, without any re-training or parameter re-tuning.

[5] We have also obtained similar results on several other Intel CPUs, including Skylake, Haswell, Alder Lake, and Kaby Lake.

[6] Fargate is the serverless platform of AWS, where tenants can run their applications in Kubernetes Pods without the need for deploying VMs [7].

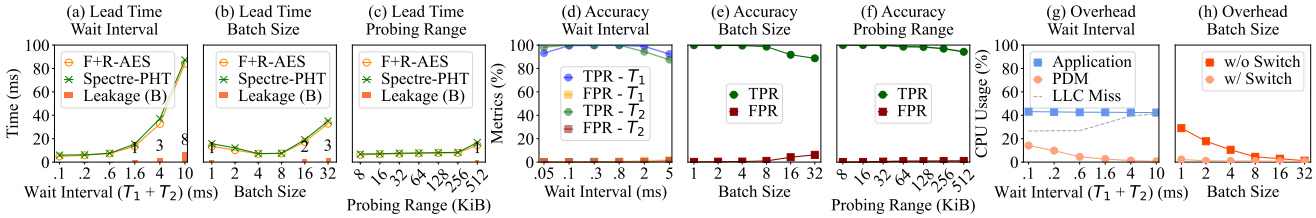


Figure 6: Impact of PDM's Parameters on Lead Time (a–c), Accuracy (d–f), and Overhead (g–h)

4.2 Parameters Evaluation

We study how PDM's parameters affect its lead time, accuracy, and overhead.

Lead Time. Fig. 6a shows that the lead time for detection to trigger mitigation increases with longer wait intervals, as a longer delay between two probing operations means each probing iteration also takes longer. Fig. 6b shows that the lead time slightly decreases for batch sizes up to 8 (due to less frequent triggering of ML inference by the inference switch), and then increases for larger batch sizes (due to larger numbers of probes). Fig. 6c shows that increasing the probing range has a negligible effect on the lead time until almost 512KiB. This is achieved with multiplexing (Scheme 3+), which can fit almost 8,192 probes (i.e., 512KiB) inside one wait interval, without much impact on the lead time. All three figures also demonstrate that a lead time below 17ms is sufficient for completely eliminating leakage (i.e., mitigation is triggered before any bytes can be leaked).

Accuracy. Fig. 6d shows that PDM has a true positive rate (TPR) of almost 100% and a false positive rate (FPR) of almost 0% with wait intervals between 0.1ms and 0.8ms (smaller or larger intervals either render the intervals too small for attacks to fall inside, or reduce the probing resolution [71, 86]). Fig. 6e shows that PDM can achieve nearly 100% TPR and 0% FPR with batch sizes smaller than 8 (larger batches cause increased imbalance between normal and attack data points). Fig. 6f shows PDM achieves nearly 100% TPR and 0% FPR for probing ranges less than 256KiB (PDM becomes more susceptible to noises for larger ranges).

Overhead. Fig. 6g shows PDM's CPU usage decreases with larger wait intervals, whereas the effect on the application itself remains negligible. Larger intervals also cause a slightly increasing LLC miss rate (54.36% without PDM), since less frequent probing means less aggressive prefetching. Fig. 6h shows the CPU overhead with the inference switch applied remains negligible, while constant ML inference has more overhead (though decreasing in batch sizes).

Based on those results, for the remaining experiments, we choose (i) an average wait interval of 0.3ms, (ii) a batch size of 8, and (iii) a probing range of 8KiB (which is sufficient for the secrets of real-world applications used in our experiments, and can be expanded for special cases up to 512KiB).

4.3 Detection Evaluation

We answer RQ1 by evaluating PDM's detection accuracy and overhead, and compare it to existing works.

4.3.1 Accuracy

Testbed. Following the literature [12, 18, 30, 32, 53, 75, 80], we evaluate PDM by performing various cache side-channel and transient execution attacks (detailed in Section 4.1) while running 20 instances of the SPEC CPU 2017 benchmark [33] as co-resident noises. As Table 2a shows, PDM's fast detection classifier (Section 3.2.2) can detect all the attacks with a $\geq$99.72% TPR and a $\leq$0.13% FPR. PDM's slow detection classifier maintains an FPR of almost 0% for all attacks, showing its ability to verify the fast classifier's results.

Real Cloud. We evaluate PDM in AWS Fargate against the FLUSH+RELOAD and FLUSH+FLUSH attacks as well as the four Spectre attack variants.[7] As shown in Table 2b, PDM can effectively detect all those attacks, with comparable results to our in-house experiments, despite the potentially noisier environment of AWS Fargate (reflected in slightly higher FPRs/lower TPRs).

Table 2: Detection Accuracy

(a) Testbed

Attack		Fast TPR (%)	Fast FPR (%)	Slow TPR (%)	Slow FPR (%)
AES	F+R/O.SSL	99.94	0.05	99.83	0
	F+F/O.SSL	99.99	0.09	99.81	0
	P+P/O.SSL	99.99	0.04	99.73	0
RSA	F+R/LibgC	99.72	0.04	99.91	0
	P+P/LibgC	99.9	0.06	99.79	0
Spectre	PHT	100	0.05	99.87	.003
	BTB	99.99	0.13	99.77	0
	RSB	99.99	0.03	100	0
	STL	99.99	0.01	100	0

(b) AWS Fargate

Attack		Fast TPR (%)	Fast FPR (%)	Slow TPR (%)	Slow FPR (%)
AES	F+R/O.SSL	99.59	0.57	99.78	0
	F+F/O.SSL	100	0.19	99.77	0
RSA	F+R/LibgC	99.9	0.25	99.52	0
	F+F/LibgC	99.78	0.23	99.8	0
	F+R/LibGDK	98.94	0.83	98.93	0.09
Spectre	PHT	100	0.2	99.79	0
	BTB	98.63	0.34	99.92	0
	RSB	99.13	0.26	100	0
	STL	99.64	0.54	99.71	0

False Positives. Assuming there is no microarchitectural attack by other tenants targeting our container during the testing, we observe an FPR of 0.02% over 12 hours of an attack-free dataset. The fast detection classifier performs around 100 inferences per second, with an FPR of 0.02%, which means one false positive can be expected *at only one of the 8KiB addresses* every 50 seconds. Moreover, as explained in Section 3.2.2 and Section 3.3, such false positives will be handled by our two-stage detection approach: (i) the

[7] PRIME+PROBE is excluded (explained in Ethical Considerations).

fast classifier only triggers non-disruptive mitigation (i.e., the only impact to a falsely detected application is a small overhead applicable only to a few addresses), and (ii) as soon as a false positive is confirmed by the slow classifier, such mitigation will be deactivated.

4.3.2 Overhead

We evaluate PDM's detection overhead in terms of both application response time and resource consumption. Probing threads must be pinned to the victim's hardware thread (vCPU) to capture its cache occupancy state, while ML inference thread does not require pinning. Since PDM is designed to operate without consuming an additional hardware thread, we pin all PDM's threads (probing, ML, victim and co-resident noise) to the same hardware thread as the victim (except in the two vCPUs setup used for performance-cost tradeoff analysis). We disable CPU frequency scaling to avoid a false sense of low overhead.

Table 3: PDM's Overhead on Response Time

	App.	One vCPU (ms)			Two vCPUs (ms)		
		Baseline	With PDM		Baseline	With PDM	
			w/o switch	w/ switch		w/o switch	w/ switch
Real-world Applications	OpenSSL AES	0.006	0.006	0.006	0.006	0.006	0.006
	OpenSSL RSA	1.012	1.015	1.01	1.012	1.013	1.012
	Libgcrypt RSA	0.146	0.147	0.147	0.14	0.143	0.143
	Libgcrypt ElGamal	19.41	19.63	19.37	19.41	19.47	19.36
	Libgcrypt EdDSA	2.98	3.1	2.9	2.98	2.99	2.98
	Nginx	2.12	2.21	2.15	2.13	2.19	2.12
	Redis	0.06	0.065	0.061	0.061	0.064	0.061
	OpenSSH	78.5	80.7	79.1	78.4	78.9	78.5
	Tomcat	0.388	0.394	0.389	0.381	0.388	0.386
	SQLite	72.62	75.02	72.89	72.63	72.98	72.65
SPEC CPU 2017 Benchmarks	600.perlbench	526e3	576e3	539e3	524e3	531e3	526e3
	602.gcc	775e3	834e3	790e3	776e3	798e3	780e3
	605.mcf	1.14e6	1.20e6	1.14e6	1.14e6	1.15e6	1.14e6
	620.omnetpp	554e3	540e3	552e3	553e3	555e3	555e3
	623.xalancbmk	409e3	435e3	414e3	409e3	415e3	408e3
	625.x264	464e3	496e3	465e3	464e3	470e3	464e3
	631.deepsjeng	635e3	681e3	636e3	635e3	642e3	636e3
	641.leela	883e3	955e3	858e3	883e3	912e3	884e3
	648.exchange2	605e3	653e3	606e3	606e3	629e3	600e3
	657.xz	3.33e6	3.54e6	3.33e6	3.33e6	3.21e6	3.33e6

Overhead on Response Time. We deploy PDM on both common cryptographic primitives and real-world applications running on one vCPU and two vCPUs. The upper half of Table 3 compares user-experienced response time (without considering other delays such as network latency) averaged over 1,000 iterations, with and without PDM. The results show that PDM introduces negligible overhead over the baseline in all cases. As the applications and the *intspeed* benchmark in SPEC CPU 2017 are all single-threaded, the second vCPU in the two vCPUs setup remains mostly idle. We therefore offload ML inference to it, to evaluate the performance-cost tradeoff of adding an extra vCPU (e.g., 3.3% overhead of SQLite reduced to 0.48%). In contrast, PDM's inference switch offers similar advantages (e.g., 3.3% overhead of SQLite reduced to 0.37%) by preventing unnecessary ML inference and without additional expense. Thus, by avoiding using an extra vCPU, PDM keeps the cost negligible, as AWS Fargate charges per-vCPU-hour.

Table 4: PDM's CPU/Memory Consumption

Lib	App.	Key (B)	Taint Size (B)	CPU (%)		Mem. (MiB)
				w/o switch	w/ switch	
OpenSSL	AES	16	3600	8.5	0.3	38.24
	AES	32	3684	7.1	0.3	38.51
	RSA	64	3470	7.3	0.3	35.52
	RSA	256	3718	9.1	0.4	38.43
	ECDH	32	7663	7.7	0.3	42.34
	RSA + AES	288	15215	11.8	3.1	36.67
O.SSL	ECDSA	32	16891	10.7	3.3	43.21
	ECDSA + ECDH	64	21318	12.1	3.7	42.89
Libgcrypt	ElGamal	128	4820	7.8	0.3	40.38
	AES	16	4848	9.6	0.3	39.33
	RSA	256	5492	7.6	0.4	40.37
	EdDSA	32	10244	11.6	2.9	41.22

Table 5: PDM's Per-component CPU/Memory Consumption

Lib	App.	CPU (%)					Memory (MiB)				
		Prob.	Noise		ML		Prob.	Noise		ML	
			Vict.	Co-res.	w/o switch	w/ switch		Vict.	Co-res.	w/o switch	w/ switch
OpenSSL	AES-16	< 0.1	< 0.1	< 0.1	8.2	0.2	0.4	0.2	1.35	36.29	2.31
	AES-32	< 0.1	< 0.1	< 0.1	7	0.1	0.4	0.2	1.35	36.56	2.28
	RSA-64	< 0.1	< 0.1	< 0.1	7.1	0.1	0.4	0.2	1.35	33.57	2.25
	RSA-256	< 0.1	< 0.1	< 0.1	8.8	0.3	0.4	0.2	1.35	36.48	2.31
	ECDH	< 0.1	< 0.1	< 0.1	7.4	0.2	0.4	0.2	1.35	40.39	2.39
	RSA + AES	2.9	< 0.1	< 0.1	8.6	0.2	0.4	0.2	1.35	34.72	2.3
	ECDSA	3.3	< 0.1	< 0.1	7.1	0.1	0.4	0.2	1.35	41.26	2.4
	ECDSA + ECDH	3.5	< 0.1	< 0.1	8.3	0.1	0.4	0.2	1.35	40.94	2.34
Libgcrypt	ElGamal	< 0.1	< 0.1	< 0.1	7.7	0.2	0.4	0.2	1.35	38.43	2.27
	AES-16	< 0.1	< 0.1	< 0.1	9.3	0.2	0.4	0.2	1.35	37.38	2.26
	RSA-256	< 0.1	< 0.1	< 0.1	7.5	0.2	0.4	0.2	1.35	38.42	2.27
	EdDSA	2.7	< 0.1	< 0.1	8.8	0.2	0.4	0.2	1.35	39.27	2.29

SPEC CPU 2017 Benchmark. We further evaluate PDM's overhead using the SPEC CPU 2017 benchmark [33] (with `LD_PRELOAD`). As the lower half of Table 3 shows, PDM's overhead remains negligible, especially with the inference switch enabled (e.g., 9.5% overhead of 600.perlbench reduced to 2.47%).

Overhead on CPU/Memory Consumption. We evaluate PDM's CPU and memory overhead by using the */proc* Linux pseudo-filesystem for measurements. As Table 4 shows, PDM's CPU and memory overheads are both reasonable, and the inference switch can significantly reduce the CPU overhead. In particular, for all the applications with tainted bytes less than 8KiB, the overhead stays below 0.4% CPU and 43.21MiB memory.

We further evaluate PDM's per component CPU and memory overhead. As Table 5 shows, PDM's probing and noise-handling threads CPU consumption stays below 0.1% for all the applications with tainted bytes less than 8KiB. This is due to the use of `nanosleep` for wait intervals, which constantly yields to the victim application. The slightly higher overhead in other cases is mainly due to multiplexing more probes within wait intervals. The ML inference accounts for more than 95% of PDM's CPU and memory consumption which motivates the use of the inference switch in PDM.

4.3.3 Comparison with Existing Works

We compare PDM to two state-of-the-art HPC-based detection solutions (despite its disadvantage of having no access to HPCs), namely, CacheShield [12] and Cho et al. [18]. We select those works as they encompass the HPCs commonly used in other studies [12, 18, 30, 32, 53, 75, 80], and represent two common approaches to HPC-based monitoring (i.e., monitoring either the victim, or all processes). We re-implemented

Table 6: Comparison of PDM and CacheShield [12]

(a) w.r.t. Victim Workload

Workload	PDM		CacheShield	
	TPR (%)	FPR (%)	TPR (%)	FPR (%)
Idle	100	0.12	15.64	0.16
Normal	99.82	0.32	99.13	0.28
High	98.64	0.9	99.86	0.04

(b) w.r.t. Co-resident Noise

# of Stress Workers	PDM		CacheShield	
	TPR (%)	FPR (%)	TPR (%)	FPR (%)
1	100	0.09	99.82	0.27
3	99.88	0.07	99.87	0.48
5	99.71	0.23	98.24	0.79

Table 7: Comparison of PDM and Cho et al. [18]

(a) w.r.t. Different Attacks

Attack	PDM		Cho et al.	
	TPR (%)	FPR (%)	TPR (%)	FPR (%)
F+R	100	0.1	100	0
F+F	99.94	0	46.3	0
P+P	99.78	0.27	99.87	0
Spec.PHT	99.93	0.14	99.8	0

(b) w.r.t. Co-resident Noise

Benign App.	PDM		Cho et al.	
	TPR (%)	FPR (%)	TPR (%)	FPR (%)
O.SSL Speed	99.91	0.12	100	0
Redis	99.94	0.01	100	0
602.gcc	99.84	0.71	100	68.2
600.perlb.	99.87	0.24	100	48.7

their methodology as these are not open-sourced.

Comparison to CacheShield [12]. As Table 6a shows, CacheShield has similar results to PDM when the victim has a *normal* workload, but worse results when the victim is *idle* (no cache misses to monitor or match to attack signatures). CacheShield shows slightly better results under *high* workload, while PDM remains reasonably accurate. Table 6b compares them under different levels of co-resident noises (generated by memory stress workers `stress-ng -m` on the same core), and high workload (which gives CacheShield advantage). Despite its lack of direct access to HPCs, PDM shows similar results to CacheShield in all cases. Finally (not shown in the table), PDM's CPU usage ($\leq$3.7%) is slightly lower than CacheShield's ($\leq$5%), and both solutions achieve a comparable detection lead time of ($\leq$7ms).

Comparison to Cho et al. [18]. As Table 7a shows, Cho et al. [18] and PDM (with no other applications running) are equally effective in detecting all the attacks with high accuracy, except that the former is ineffective against the FLUSH+FLUSH attack (as it only flushes target addresses and generates no cache misses [30]). Table 7b compares them with benign applications running during the attack. Cho et al. [18] misclassifies the SPEC CPU 2017 *gcc* and *perlbench* as attacks. Our analysis shows that those applications generate far more anomalous cache hits/misses than attacks, e.g., *gcc*'s cache misses are 25 times more than FLUSH+RELOAD (Appendix B). In contrast, PDM can successfully handle such cache hits/misses as co-resident noises since their impact on the victim is different from that of attacks.

4.4 Mitigation Evaluation

We answer RQ2 by evaluating PDM's mitigation effectiveness and overhead, and compare it to an existing work.

4.4.1 Cache Side-Channel Attacks

We evaluate PDM's mitigation on FLUSH+RELOAD [86] and PRIME+PROBE [42,58,69,92]) performed on the T-table implementation of *OpenSSL AES* [36,37]. As Table 8a shows,

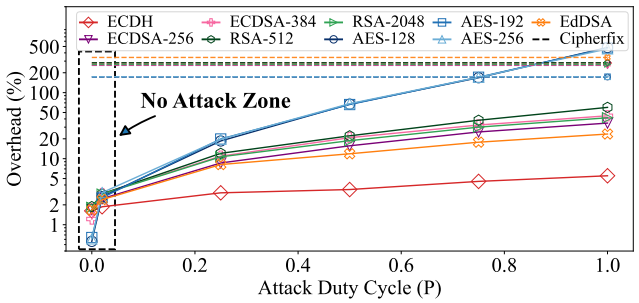


Figure 7: PDM's In-memory Encryption Overhead vs. Cipherfix [83] Overhead under Different Attack Duty Cycles (P)

without PDM (*baseline*), the attacker can successfully retrieve the AES key with 0% bit error rate. With PDM's first stage of mitigation (probing only), the bit error rate increases by up to 37% for all attacks (due to deceiving access patterns generated by probing, as explained in Section 3.3). With the second stage of mitigation (intense obfuscation), PDM can basically eliminate the leakage as the bit error rate increases up to 51%.

Table 8: Effectiveness of Mitigation on Detected Attacks

(a) Cache Side-Channel Attacks

Attack	Bit Error Rate (%)		
	Base.	Stage 1	Stage 2
F+R/O.SSL AES	0	35	51
P+P/O.SSL AES	0	32	46
F+R/Libgc RSA	0	37	47
P+P/Libgc RSA	0	35	46

(b) Transient Execution Attacks

Attack	Leakage Rate (B/s)	Leaked Bytes
Spectre-PHT [48]	4420.46	0
Spectre-BTB [26]	269.53	0
Spectre-STL [26]	272.46	0
Spectre-RSB [26]	116.70	0

4.4.2 Transient Execution Attacks

Next, we evaluate PDM's in-memory encryption mitigation method (detailed in Section 3.3) against transient execution attacks. Table 8b shows the leakage rate of various Spectre variants, and the number of bytes that can be leaked before PDM activates its mitigation (7ms). As the results show, despite the different leakage rates of those Spectre variants, PDM's detection can effectively trigger its in-memory encryption mitigation just in time before any leakage occurs.

4.4.3 Comparison with Existing Work

We compare PDM with Cipherfix [83] in terms of mitigation overhead for different cryptographic applications. Cipherfix represents the closest state-of-the-art approach, as it also proposes a binary-level in-memory encryption solution. The attack pattern used for evaluation has its binary state transition following a continuous-time Markov chain (CTMC), where the *attack duty cycle* (i.e., the steady-state probability of being *On*) is denoted by $P = \frac{\mu}{\lambda+\mu}$, with μ for the rate of state changes from *Off* to *On*, and λ for the opposite.

In Fig. 7, the logarithmic scale on the y-axis highlights that PDM's in-memory encryption generally incurs less overhead (in terms of average application execution time observed over one hour) than Cipherfix, except for AES when the attack duty cycle is $P = 1$. In particular, the left end of the curves

Table 9: PDM's In-memory Encryption Overhead under Different Attack Duty Cycles (P), Workload (λ), and Ratio of Instrumented/Total Memory Accesses, in Comparison to Cipherfix [83] overhead (crypto applications are chosen to allow comparison)

	App.	Base	P=0						P=0.25		P=0.5		P=0.75		P=1				# of Instrum. Mem. Access	Total # of Mem. Access	Instrum. /Total Ratio (%)	Cipher Fix Over. (%)
			λ = 0.4		λ = 10		λ = 100								w/ detect.		w/o detect.					
		Time (μs)	Time (μs)	Over. (%)	Time (μs)	Over. (%)	Time (μs)	Over. (%)	Time (μs)	Over. (%)	Time (μs)	Over. (%)	Time (μs)	Over. (%)	Time (μs)	Over. (%)	Time (μs)	Over. (%)				
OpenSSL	ECDH	556.93	565.94	1.61	566.04	1.63	567.34	1.86	573.84	3.03	575.94	3.41	582.16	4.53	587.61	5.50	580.87	4.29	940K	2.9B	0.03	281
	ECDSA-384	3843.9	3891.2	1.22	3909.1	1.69	3957.5	2.95	4263.7	10.92	4642.9	20.78	5087.5	32.35	5578.5	45.12	5535.3	44.00	62.7M	36.6B	0.17	262
	RSA-512	153.86	156.76	1.88	156.9	1.97	158.37	2.93	172.4	12.04	188	22.18	212.61	38.18	245.85	59.78	243.28	58.11	2.9M	2.32B	0.12	284
	RSA-2048	1848.9	1882.7	1.83	1884.4	1.92	1902.7	2.91	2046.1	10.66	2195.5	18.74	2404	30.02	2618.3	41.61	2601.1	40.68	56.1M	79.8B	0.07	268
WolfSSL	AES-128	1.51	1.52	0.66	1.52	0.66	1.55	2.64	1.79	18.54	2.52	66.88	4.05	168.2	8.79	482.1	8.78	481.4	347K	2.0M	17.18	172
	AES-192	1.56	1.57	0.64	1.57	0.64	1.6	2.56	1.87	19.87	2.59	66.03	4.23	171.1	8.96	474.3	8.95	473.7	363K	2.1M	17.12	174
	AES-256	1.63	1.64	0.61	1.64	0.61	1.68	3.06	1.95	19.63	2.74	68.09	4.41	170.5	9.13	460.1	9.12	459.5	379K	2.2M	17.08	174
	EdDSA	83.27	84.67	1.68	84.78	1.81	85.27	2.4	90.06	8.15	93.14	11.86	98.06	17.76	102.93	23.6	101.89	22.36	20.7K	102M	0.02	341

(*No Attack Zone*) shows what users would experience most of the time, i.e., with no attack present. PDM's negligible (< 4%) overhead in this case highlights the major benefit of our detect-and-mitigate approach in contrast to Cipherfix's constant overhead (even when no attack is present).

Table 9 shows more detailed numerical results. Increasing victim workload (λ) in the absence of an attack (*P*=0) leads to slightly higher overhead due to the rise in false positives (reported in Table 10). Nonetheless, the overhead remains below 3.06% across all cases. Among all the applications, AES variants show the highest overhead, which is explained by the highest ratio of instrumented-to-total memory accesses. The standalone mitigation overhead (i.e., that of the signal handler alone without detection) is reported under *P=1, w/o detect.*

4.5 Robustness

We answer RQ3 by evaluating PDM against (i) victim noise, (ii) co-resident noise, and (iii) evasive attack.

Table 10: Impact of Victim Application Noise

App.	Idle						Normal						High					
	w/o		w/		slow		w/o		w/		slow		w/o		w/		slow	
	TP (%)	FP (%)	TP (%)	FP (%)	TP (%)	FP (%)	TP (%)	FP (%)	TP (%)	FP (%)	TP (%)	FP (%)	TP (%)	FP (%)	TP (%)	FP (%)	TP (%)	FP (%)
O.SSL AES	100	0.1	100	0.1	99.8	0	99.5	0.9	99.8	0	99.3	0	90	2.7	96.4	1.8	97.1	0.2
Libg. RSA	100	0	100	0	100	0	99.8	0.1	99.7	0	99.2	0	98.1	1.5	99.2	0.7	98.9	0
Nginx	99.9	0.3	99.9	0.1	99.9	0	99.7	0.8	99.7	0.2	99.5	0	96.8	4.6	98.5	0.3	98.6	0
Redis	99.9	0.4	99.8	0.2	99.6	0	99.7	0.1	99.5	0.1	99.8	0	95.7	2.5	98.6	1.4	99.3	.08
OpenSSH	100	0.6	100	0.6	99.8	0	99.8	0.6	99.8	0.2	99.8	0	98.7	1.8	99.1	0.4	98.3	0

4.5.1 Impact of Victim Noise

We evaluate PDM on several real-world applications as listed in Table 10, with varying workloads and with 100 client processes that: (i) send no request (*idle*), (ii) send requests following a Poisson distribution with λ = 0.4 [59] to simulate real-world traffic (*normal*), or (iii) send requests at CPU clock rate (λ = 100) (*high*). As Table 10 shows, without workload analysis (*w/o*), detection remains accurate under the *idle* and *normal* noise levels, but degrades under *high* (e.g., AES TPR=90%, Nginx FPR=4.6%). Enabling workload analysis (*w/*) can improve the results under *high* (e.g., AES TPR=96.4% and Nginx FPR=0.3%). Finally, the slow classifier can further reduce false positives in all cases.

4.5.2 Impact of Co-resident Noise

Fig. 8 shows the impact of co-resident noise across three settings: (i) host-level (no isolation), (ii) container-level (Kubernetes `cgroups` isolation), and (iii) VM-level (KVM). Since cloud tenants are typically isolated [4], host-level noise represents the worst case and is intended as a stress-test.

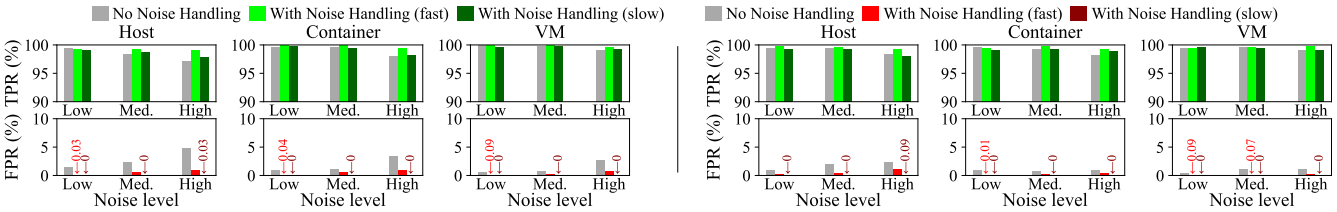
Applications on Same Core. Fig. 8a shows the case where co-resident applications share the victim's core. Without noise handling, detection is unaffected under *low* (five apps) and *medium* (10 apps), but degrades under *high* (15 apps). Enabling noise handling improves the results, e.g., FPR reduced from 4.78% to 0.89% in the worst case (host-level, 15 apps). The slow classifier maintains an almost zero FPR.

Applications on All Cores. Fig. 8b shows similar results but with the applications running on 14 (*low*), 28 (*medium*), or all 56 (*high*) cores other than the victim's. The results follow a similar trend as in Fig. 8a, but with a lower impact (since contention is limited to shared L3 and only indirectly affects dedicated caches). Noise handling proves effective, reducing FPR from 2.36% to 1.11% in the worst case (host-level, 15 apps). The slow classifier also achieves a near-zero FPR.

Intensive Noise using Stressors. To further challenge PDM, we use *stress-ng* to generate synthetic workloads targeting cache, CPU, I/O, and memory. Fig. 9a shows results with 5, 10, and 15 workers (on the victim's core). Memory stressor has the highest impact as it evicts all cache levels, while cache stressor has less impact as it is limited to L1/L2. Fig. 9b shows the impact becomes lower with workloads on all cores, as contention is limited to L3. In all cases, noise-handling remains effective (e.g., FPR reduced from 2.1% to 1.98% under memory stressor, high, all cores), and the slow classifier maintains a near-zero FPR.

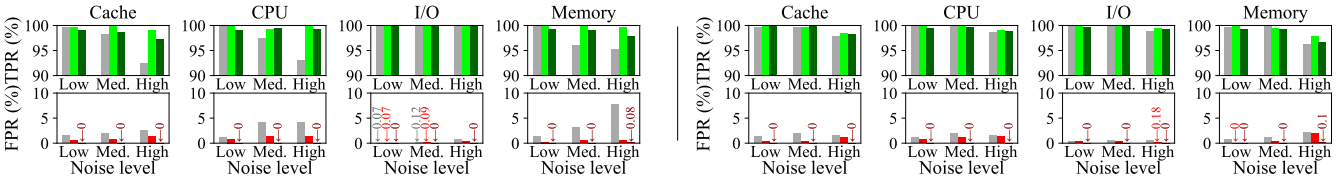
4.5.3 Impact of Evasive Attacks

We compare PDM with CacheShield [12] and Cho et al. [18] against evasive attacks that try to evade detection by slowing down [45, 50, 54, 75, 85]. While this tactic may achieve stealthiness, it further prolongs attack duration (already long under real-world cloud settings [42, 44, 48, 58, 73, 92]), and reduces the attack success rate, (e.g., 7× slowdown in Spectre drops success to 85% [54]). Thus, the 32×-1024× slowdowns are well outside practical attack ranges, and primarily serve



(a) Noise on the Victim's Core (b) Noise on All Cores

Figure 8: Impact of Co-resident Application Noise, with and without Noise Handling, at Host-level (No Isolation), Container-level (`cgroups` Isolation), VM-level (Hypervisor Isolation)



(a) Stress on the Victim's Core (b) Stress on All Cores

Figure 9: Impact of Host-level Stressors (Cache, CPU, I/O, and Memory) Noise, with and without Noise Handling

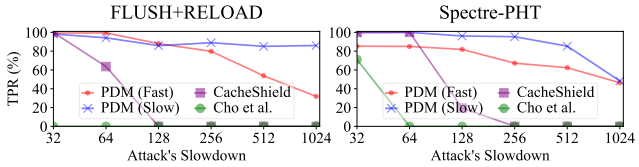


Figure 10: PDM's Effectiveness in Detecting Evasive Attacks in Comparison to CacheShield [12] and Cho et al. [18]

as stress-tests. As shown in Fig. 10 (left), PDM's fast classifier's TPR for FLUSH+RELOAD begins to drop only after slowdown factors $\geq$128, while Spectre-PHT degrades earlier ($\geq$32) and remains lower due to its inherently slower nature (branch predictor mistraining + covert exfiltration). In contrast, the slow classifier is less affected in both cases due to its better hyperparameters and coverage of more data points (FPR stays negligible in all cases and thus is omitted).

CacheShield's TPR for FLUSH+RELOAD drops after slowdown factors $\geq$32, while Cho et al. achieve 0% TPR across all slowdowns. This demonstrates that victim-centric detection is more robust to evasion since slowdowns reduce the attacker's signal faster than it reduces the victim's footprint (also observed by Weber et al. [80]). PDM victim-centric detection outperforms CacheShield due to its *selective* monitoring of secrets, and its *active* probing independent of victim's workload. While neither CacheShield nor Cho et al. consider Spectre in their threat model, both can naturally detect the FLUSH+RELOAD exfiltration phase, whereas PDM detects the speculative access phase itself. As a result, CacheShield cannot detect cross-process Spectre attacks due to the absence of FLUSH+RELOAD patterns. For a fair comparison, we evaluate both solutions against an intra-process Spectre. CacheShield maintains nearly 100% TPR up to a slowdown of 64 before degrading, while Cho et al. achieves about 70% TPR at a slowdown of 32 and 0% TPR thereafter.

5 Related Work

Existing works on detecting microarchitectural attacks mostly rely on hardware performance counters (HPCs) to identify suspicious patterns that are indicative of microarchitectural attacks [12,18,30,32,53,75,80]. Among these, the majority take an *attacker-centric* approach to detect the attacker's process by monitoring the entire system [18,30,32,75], which can be useful for the cloud provider to identify misbehaving tenants or applications. The main limitation of such attacker-centric solutions lies in the high false positive rate during memory-intensive operations, as mentioned in recent works [50,80], and also shown in Section 4.3.3 and Appendix B.

Closer to our work, there exist *victim-centric* HPC-based monitoring solutions [12,53,80] and solutions specifically designed for cloud environments [12,75]. Specifically, we have reported experimental comparison results for CacheShield [12] (Section 4.3.3), which is an HPC-based self-monitoring approach for detecting microarchitectural attacks, and its victim-centric approach is similar to our work. Lantz et al. [53] explore the feasibility of detecting load value injection attacks on Intel SGX enclaves using HPCs (although the authors acknowledge that direct access to HPCs is disabled within SGX, similarly to clouds). IRQGuard [80] monitors specific parts of the victim code to achieve a more focused and synchronous HPC data collection, although it requires annotating critical segments within the source code, not applicable to existing binaries. Schwarzl et al. [75] leverage HPCs to detect Spectre attacks running on JavaScript sandboxes of Cloudflare workers. All those HPC-based solutions can only be deployed by the cloud provider, since tenants generally lack access to low-level features such as HPCs, or a central view of all the processes running on the physical host (e.g., on shared hosts in Amazon AWS [28]).

In contrast, PDM only relies on tenant-accessible resources and can be independently deployed by cloud tenants to work in tandem with those provider-level solutions.

Similar to PDM, the authors in [90] also rely on probing for detection, although they target SGX enclaves (where HPC access is also disabled) instead of clouds. Moreover, their probing is performed outside the enclave by scanning the cache, and it depends on hardware performance counters and monitoring the page-table accessed bit, all of which require provider-level access. WaitGuard [52] detects flush-based attacks by monitoring sensitive cache lines and flagging spurious wake-ups caused by the flush instruction on Intel Sapphire Rapids and Emerald Rapids CPUs. The potential of employing the same side-channel analysis as used in microarchitectural attacks for detecting cache side-channel attacks has been mentioned in earlier works [31, 89], but has not been explored further. In contrast, PDM has fully developed this idea into a concrete solution while addressing various challenges and extending the scope to detect Spectre attacks. There also exist works for detecting the attacker's program [43], detecting microarchitectural vulnerabilities in production software [67, 78, 79, 82, 87], or modifying the program's binary to enable runtime detection [17]. In contrast to those works, PDM has a different focus, i.e., detecting ongoing microarchitectural attacks at the tenant level. Finally, there also exist mitigation approaches for microarchitectural attacks, such as preventing unsafe speculative execution through offline code analysis [15,61,66,78], marking memory locations containing secrets as uncacheable using a kernel module [73], and using execute-only memory [35]. In contrast, PDM only relies on tenant-accessible mitigation solutions including obfuscation, encryption, and migration, and PDM's detect-and-mitigate approach ensures a low overhead in applying them.

6 Limitations and Discussions

Other Microarchitectural Attacks. PDM focuses on microarchitectural attacks that alter the victim's *cache occupancy state* via active memory probing, which enables detection even if the contention is low. For instance, the RELOAD+REFRESH attack [13] avoids generating L3 misses but evicts victim's data from L1/L2 caches while refreshing replacement ages [85]. We tested PDM's classifier (without retraining) on this attack and achieved a TPR of 91% (fast classifier) and 98.3% (slow classifier), with attack patterns visible as increased L3 hits in both Access+Reload and Flush+Reload probing features. PDM's approach can also be extended to other contention channels. For instance, port contention [3] can be detected by deploying a concurrent thread that probes CPU ports, where latency variations both reveal attacks and obfuscate access patterns. More generally, those demonstrate how side-channel techniques used in attacks can be repurposed as defensive mechanisms, especially in constrained cloud settings.

Evasive Slow Attacks. As shown in Section 4.5.3, PDM can only detect attacks slowed down to a certain level. Jiang et al. [45] propose reducing data leakage to a single bit per run by assuming a powerful attacker controlling the OS and victim's execution. Although such an assumption may not be realistic in cloud environments, PDM can nonetheless be extended to detect such attacks by spawning a dedicated thread to work in a *slow* mode, which relaxes the constraint on detection lead time for a larger time window covering more historical data.

Evasive Spectre. PDM detects Spectre by identifying cache hits caused by unexpected memory accesses. Note that even when alternative covert channels replace cache-based data exfiltration [11, 20], the secret-dependent speculative access remains, so PDM will be effective. To evade detection, an attacker must eliminate these unexpected accesses, e.g., the attacker synchronizes the attack's speculative memory loads with benign memory accesses of the victim, which is challenging since the attacker would require knowledge of the victim's binary and how/when secrets are being accessed by the victim.

Mask Protection. A legitimate concern of in-memory encryption is the storage of XOR masks, as these can also become attack targets. In our design, masks are refreshed for every memory write using an AES-based PRNG, reducing the time window during which a leaked mask remains useful. Nonetheless, stored masks can further be protected using additional mechanisms: (i) the masks can additionally be included in the probing range monitored by PDM; (ii) the masks can be derived from an overly long secret in such a way that the entire secret must be leaked before any key can be derived by an attacker.

Secret Annotation. To determine the probing range of PDM, users annotate memory addresses based on information about the binary and its shared libraries (Section 3.2.1). In general, secret annotation is an important requirement for most solutions performing *selective* protection (e.g., data encryption and isolation). Existing works rely on developers to annotate secrets in code [52, 70, 80, 83]. Semi-automated tools can also be applied to ease this process, e.g., by locating high-entropy regions of the memory [55].

7 Conclusion

We proposed PDM to enable cloud tenants to independently detect and mitigate microarchitectural attacks. We provided methodologies for PDM to (i) monitor the victim's memory space through probing, (ii) inject itself as a thread to detect attacks while handling noises, and (iii) mitigate data leakage upon detection. We implemented and evaluated PDM both in our local testbed and on Amazon Fargate, and our experimental results confirmed its effectiveness, efficiency, and robustness. PDM could potentially (i) increase cloud tenants' awareness of microarchitectural attacks, (ii) pressure cloud providers to timely deploy vendor patches, (iii) complement provider-level solutions to provide additional protection.

Ethical Considerations

Stakeholder Analysis. Our work involves several stakeholders: (i) *cloud tenants*, who may benefit from better visibility and self-defense, with safeguards to avoid exposing tenant-specific information; (ii) *cloud providers*, who may gain from identifying gaps in provider-only defenses without disclosure of proprietary details; (iii) *researchers and the security community*, who can reuse our methods and artifacts, which avoid attack-enabling information; (iv) *the general public and software ecosystem*, who indirectly benefit from improved cloud security; and (v) *our industrial partner*, who sponsors the work and owns the IP, without influencing ethical safeguards.

Ethical Principles. Following the Menlo Report, we considered the principles of (i) *Beneficence*: PDM is designed to reduce harm by enabling tenants to detect and mitigate microarchitectural attacks, thereby strengthening (not weakening) cloud security; (ii) *Respect for Persons*: no human subjects were involved. All experiments were performed on controlled testbeds and author-owned cloud instances; (iii) *Justice*: the research is intended to benefit cloud tenants broadly, regardless of provider or resource level, and treats all providers uniformly; (iv) *Respect for Law and Public Interest*: all work complied with platform terms of service, local regulations, and responsible disclosure norms. We avoided unauthorized testing or data collection and followed "Good Faith Security Research" guidelines for AWS. Permission to publish was obtained from the relevant IP owners.

Potential Harms and Mitigations. Potential harms fall into two categories. (i) *Tangible harms*: probing techniques could be misused offensively; therefore, we release only benign detection components and sanitized datasets, omitting attack code or parameters that would enable exploit reproduction. Public-cloud experiments were conducted with low-intensity workloads on isolated, author-owned instances and within provider policies. We excluded PRIME+PROBE due to the ethical risk of unintentionally affecting co-resident tenants on shared LLC resources; (ii) *Intangible harms*: privacy or confidentiality risks are avoided because no third-party data, personal information, or real tenant workloads were used. Dual-use concerns are minimized by restricting evaluation to self-controlled instances and synthetic attacks, and by designing the framework strictly for defensive purposes.

Decision to Proceed and Publish. We determined that the benefits (increased tenant autonomy, awareness of microarchitectural threats, and contribution to defensive research) outweigh the potential risks of misuse. We concluded that the publication aligns with public interest, enabling the community to understand and build upon tenant-side defenses without exposing dangerous attack details. Our decision was supported by three ethical analyses: the expected harm reduction for tenants and the security community justifies dissemination (*Beneficence*), no human subjects were affected (*Respect for Persons*), and all work was compliant with platform and legal standards (*Respect for Law*).

Open Science

We provide our artifacts to enable reproduction of our results, which are available at https://zenodo.org/records/17967756. The repository contains both detection artifacts (probing and runtime inference source code alongside with all the trained models and datasets used in our experiments, which include the normal activities of real-world applications, attack activities, natural noises from SPEC CPU 2017 and artificial noises from memory stressors, and slow evasive attacks), and mitigation artifacts (in-memory encryption source code and pre-compiled binaries of applications with PDM integrated), with usage instructions.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable comments. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Canada, Prompt Quebec, and the Canada Foundation for Innovation (CFI).

References

[1] 0xADE1A1DE. Mastik: A micro-architectural side-channel toolkit. https://github.com/0xADE1A1DE/Mastik, 2016. Accessed: 2025-08-15.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2019.

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE Symposium on Security and Privacy (S&P)*, pages 870–887, 2019.

[4] Amazon Web Services. Security design of the AWS Nitro system. https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf, 2021. Accessed: 2025-08-15.

[5] Amazon Web Services. Shared responsibility model. https://aws.amazon.com/compliance/shared-responsibility-model/, 2023. Accessed: 2025-08-15.

[6] Amazon Web Services. *Amazon EC2 Dedicated Instances*, 2024. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/dedicated-instance.html. Accessed: 2025-08-15.

[7] Amazon Web Services. Fargate security best practices in Amazon ECS. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/security-fargate.html, 2024. Accessed: 2025-08-15.

[8] Ben Amos, Niv Gilboa, and Arbel Levy. Spectre without shared memory. In *ACM/SIGAPP Symposium on Applied Computing*, pages 1944–1951, 2019.

[9] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre v2 attacks. In *USENIX Security Symposium (USENIX Security)*, pages 971–988, 2022.

[10] Jonathan Behrens, Adam Belay, and M Frans Kaashoek. Performance evolution of mitigating transient execution attacks. In *European Conference on Computer Systems (EuroSys)*, pages 251–265, 2022.

[11] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *ACM Conference on Computer and Communications Security (CCS)*, pages 785–800, 2019.

[12] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. CacheShield: Detecting cache attacks through self-observation. In *ACM CODASPY*, pages 224–235, 2018.

[13] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. {RELOAD+ REFRESH}: Abusing cache replacement policies to perform stealthy cache attacks. In *USENIX Security Symposium (USENIX Security)*, pages 1967–1984, 2020.

[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium (USENIX Security)*, pages 249–266, 2019.

[15] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. SoK: Practical foundations for software Spectre defenses. In *IEEE Symposium on Security and Privacy (S&P)*, pages 666–680, 2022.

[16] Marco Cesati, Renato Mancuso, Emiliano Betti, and Marco Caccamo. A memory access detection methodology for accurate workload characterization. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 141–148. IEEE, 2015.

[17] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjá vu. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 7–18, 2017.

[18] Jonghyeon Cho, Taehun Kim, Soojin Kim, Miok Im, Taehyun Kim, and Youngjoo Shin. Real-time detection for cache side channel attack using performance counter monitor. *Applied Sciences*, 10(3):984, 2020.

[19] CRIU Community. CRIU - checkpoint/restore in userspace. https://github.com/checkpoint-restore/criu, 2024. Accessed: 2025-08-15.

[20] Jesse De Meulemeester, Antoon Purnal, Lennert Wouters, Arthur Beckers, and Ingrid Verbauwhede. SpectrEM: Exploiting electromagnetic emanations during transient execution. In *USENIX Security Symposium (USENIX Security)*, pages 6293–6310, 2023.

[21] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 15–27, 2019.

[22] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. PRIME+ABORT: A timer-free high-precision L3 cache attack using Intel TSX. In *USENIX Security Symposium (USENIX Security)*, pages 51–67, 2017.

[23] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 187–190, 2018.

[24] Lukas Giner, Stefan Steinegger, Antoon Purnal, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss. Scatter and split securely: Defeating cache contention and occupancy attacks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2273–2287, 2023.

[25] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the spectre era. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1871–1885, 2020.

[26] Google. SafeSide: Comprehensive demonstrations of hardware side channels and exploits. https://github.com/google/safeside, 2020. Accessed: 2025-08-15.

[27] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, 2018.

[28] Brendan Gregg. The PMCs of EC2: Measuring IPC. https://brendangregg.com/blog/2017-05-04/the-pmcs-of-ec2.html, 2017. Accessed: 2025-08-15.

[29] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: Long live KASLR. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 161–176, 2017.

[30] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. FLUSH+FLUSH: A fast and stealthy cache attack. In *Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 279–299, 2016.

[31] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium (USENIX Security)*, pages 897–912, 2015.

[32] B Gulmezoglu, A Moghimi, T Eisenbarth, and B Sunar. FortuneTeller: Predicting microarchitectural attacks via unsupervised deep learning. *arXiv preprint arXiv:1907.03651*, 2019.

[33] John Henning. *SPEC CPU 2017 Documentation*. Standard Performance Evaluation Corporation (SPEC), 2024. https://www.spec.org/cpu2017/.

[34] Jann Horn. Speculative execution, variant 4: Speculative store bypass, 2018.

[35] Tristan Hornetz, Lukas Gerlach, and Michael Schwarz. Lixom: Protecting encryption keys with execute-only memory. In *Financial Cryptography and Data Security (FC)*, 2025.

[36] Institute of Applied Information Processing and Communications. FLUSH+FLUSH. https://github.com/isec-tugraz/flush_flush, 2016. Accessed: 2025-08-15.

[37] Institute of Applied Information Processing and Communications. Cache template attacks. https://github.com/isec-tugraz/cache_template_attacks, 2018. Accessed: 2025-08-15.

[38] Institute of Applied Information Processing and Communications. TransientFail: Transient execution attacks failures and mitigations. https://github.com/isec-tugraz/transientfail, 2024. Accessed: 2025-08-15.

[39] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2021. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html. Accessed: 2025-08-15.

[40] Intel. PIN: A dynamic binary instrumentation tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, 2024. Accessed: 2025-08-15.

[41] Intel Corporation. Guidelines for mitigating timing side channels against cryptographic implementations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html. Accessed: 2025-08-15.

[42] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *IEEE Symposium on Security and Privacy (S&P)*, pages 591–604, 2015.

[43] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. MASCAT: Preventing microarchitectural attacks before distribution. In *ACM CODASPY*, pages 377–388, 2018.

[44] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! A fast, cross-VM attack on AES. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 299–319. Springer, 2014.

[45] Jianyu Jiang, Claudio Soriente, and Ghassan Karame. On the challenges of detecting side-channel attacks in SGX. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 86–98, 2022.

[46] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, pages 121–132, 2012.

[47] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[48] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[49] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre

Returns! speculation attacks using the return stack buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[50] William Kosasih, Yusi Feng, Chitchanok Chuengsatiansup, Yuval Yarom, and Ziyuan Zhu. SoK: Can we really detect cache side-channel attacks by monitoring performance counters? In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 172–185, 2024.

[51] Kubo. Injector - a tool to inject shared libraries into running processes. https://github.com/kubo/injector, 2021. Accessed: 2025-08-15.

[52] Lukas Lamster, Fabian Rauscher, Martin Unterguggenberger, and Stefan Mangard. Waitwatcher & waitguard: Detecting flush-based cache side-channels through spurious wakeups. In *European Symposium on Research in Computer Security (ESORICS)*, 2025.

[53] David Lantz, Felipe Boeira, and Mikael Asplund. Towards self-monitoring enclaves: Side-channel detection using performance counters. In *Nordic Conference on Secure IT Systems (NordSec)*, pages 120–138, 2022.

[54] Congmiao Li and Jean-Luc Gaudiot. Challenges in detecting an "evasive spectre". *IEEE Computer Architecture Letters*, 19(1):18–21, 2020.

[55] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-hunt: Pinpointing insecure cryptographic keys from execution traces. In *ACM Conference on Computer and Communications Security (CCS)*, pages 412–425, 2018.

[56] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium (USENIX Security)*, pages 973–990, 2018.

[57] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE International Symposium on High Performance Computer Architecture*, pages 406–418, 2016.

[58] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (S&P)*, pages 605–622, 2015.

[59] Zhen Liu, Nicolas Niclausse, and César Jalpa-Villanueva. Traffic model and performance evaluation of web servers. *Performance Evaluation*, 46(2-3):77–100, 2001.

[60] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *ACM Conference on Computer and Communications Security (CCS)*, pages 2109–2122, 2018.

[61] Tiziano Marinaro, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Hamed Nemati. Beyond over-protection: A targeted approach to Spectre mitigation and performance optimization. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 203–216, 2024.

[62] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 48–65, 2015.

[63] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178*, 2019.

[64] M Asim Mukhtar, Maria Mushtaq, M Khurram Bhatti, Vianney Lapotre, and Guy Gogniat. FLUSH+PREFETCH: A countermeasure against access-driven cache-based side-channel attacks. *Journal of Systems Architecture*, 104:101698, 2020.

[65] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1737–1752, 2023.

[66] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You Shall Not Bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506*, 2018.

[67] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *USENIX Security Symposium (USENIX Security)*, pages 1481–1498, 2020.

[68] ONNX Runtime Team. ONNX runtime documentation. https://onnxruntime.ai/docs/, 2024. Accessed: 2025-08-15.

[69] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology–CT-RSA*, pages 1–20, 2006.

[70] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. DynPTA: Combining static and dynamic analysis for practical selective data protection. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1919–1937, 2021.

[71] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. PRIME+SCOPE: Overcoming the observer effect for high-precision cache contention attacks. In *ACM Conference on Computer and Communications Security (CCS)*, pages 2906–2920, 2021.

[72] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A systematic evaluation of novel and existing cache side channels. In *Network and Distributed System Security Symposium (NDSS)*, 2025.

[73] Michael Schwarz, Moritz Lipp, Claudio Alberto Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A generic approach for mitigating Spectre. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[74] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security (ESORICS)*, pages 279–299, 2019.

[75] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Robust and scalable process isolation against Spectre in the cloud. In *European Symposium on Research in Computer Security (ESORICS)*, pages 167–186, 2022.

[76] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium (USENIX Security)*, pages 991–1008, 2018.

[77] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *IEEE Symposium on Security and Privacy (S&P)*, pages 88–105, 2019.

[78] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against Spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 47(11):2504–2519, 2019.

[79] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying cache-based side channels through secret-augmented abstract interpretation. In *USENIX Security Symposium (USENIX Security)*, pages 657–674, 2019.

[80] Daniel Weber, Leonard Niemann, Lukas Gerlach, Jan Reineke, and Michael Schwarz. No leakage without state change: Repurposing configurable cpu exceptions to prevent microarchitectural attacks. In *Annual Computer Security Applications Conference (ACSAC)*, pages 366–379, 2024.

[81] Zane Weissman, Thore Tiemann, Thomas Eisenbarth, and Berk Sunar. Microarchitectural security of firecracker vmm for serverless cloud platforms. In *International Conference on Information Systems Security*, pages 3–24. Springer, 2025.

[82] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. Microwalk: A framework for finding side channels in binaries. In *Annual Computer Security Applications Conference (ACSAC)*, pages 161–173, 2018.

[83] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. Cipherfix: Mitigating ciphertext side-channel attacks in software. In *USENIX Security Symposium (USENIX Security)*, pages 6789–6806, 2023.

[84] Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. InSpectre Gadget: Inspecting the residual attack surface of cross-privilege Spectre v2. In *USENIX Security Symposium (USENIX Security)*, 2024.

[85] Minjun Wu, Stephen McCamant, Pen-Chung Yew, and Antonia Zhai. PREDATOR: A cache side-channel attack detector based on precise event monitoring. In *IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 25–36, 2022.

[86] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache Side-Channel attack. In *USENIX Security Symposium (USENIX Security)*, pages 719–732, 2014.

[87] Yuanyuan Yuan, Zhibo Liu, and Shuai Wang. CacheQL: Quantifying and localizing cache side-channel vulnerabilities in production software. In *USENIX Security Symposium (USENIX Security)*, pages 2009–2026, 2023.

[88] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. DoS sttacks on your memory in cloud. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 253–265, 2017.

[89] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *ACM Conference on Computer and Communications Security (CCS)*, pages 990–1003, 2014.

[90] Zeyu Zhang, Xiaoli Zhang, Qi Li, Kun Sun, Yinqian Zhang, Songsong Liu, Yukun Liu, and Xiaoning Li. See through walls: Detecting malware in SGX enclaves

with SGX-bouncer. In *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, pages 931–943, 2021.

[91] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Everywhere all at once: Co-location attacks on public cloud FaaS. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–149, 2024.

[92] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Last-level cache side-channel attacks are feasible in the modern public cloud. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASLPLOS)*, pages 582–600, 2024.

A Implementation of In-memory Encryption

Coverage. We implemented our in-memory encryption approach in approximately 5,000 lines of C code. It supports x86_64 loads, stores, and read–modify–write operations, and can handle both scalar and vector forms across all general-purpose registers (including the high 8-bit registers) and SIMD registers (XMM/YMM). It supports all standard addressing modes (base-only, RIP-relative, and `base+index*scale`) by reconstructing effective addresses exactly as the hardware does. In practice, this covers common instruction families used over sensitive data including scalar moves and zero/sign-extends (`MOV`/`MOVZX`/`MOVSX`/`MOVSXD`), comparisons and tests with memory (`CMP`/`TEST`), arithmetic/logic with a memory operand or immediate (`ADD`/`XOR`/`OR`), vector moves (`(V)MOVDQA`/`(V)MOVDQU`, `MOVAPS`/`MOVAPD`/`MOVUPS`/`MOVUPD`), and the string primitives `REP MOVS*`/`REP STOS*` (DF=0).

Security Analysis. PDM will only build trampolines after an architecturally delivered fault is caused by the program's legitimate instructions for accessing sensitive data. Spectre adversaries can only exploit public non-sensitive instructions and transiently redirect them toward secrets. They can neither invoke the signal handler nor cause trampoline generation during speculation. Moreover, sensitive data are *always encrypted at rest* and are decrypted only in CPU registers (except for mask storage), and the trampolines do not emit any plaintext-dependent memory access. Consequently, even if speculation later fetches an already-built trampoline, the values read from memory remain ciphertext, and any transient computation over them will not create a plaintext-dependent cache footprint [70]. Finally, our approach protects against any out-of-bound read attacks (e.g., Heartbleed) since secret data remains encrypted at rest.

Data-centric Protection Without Instruction Tracking. A natural alternative for a tenant-level mitigation is selective fencing, where memory barriers are inserted before instructions that may transiently expose secrets [15, 61, 66, 78]. However, the main challenge lies in identifying all such instructions, especially in binaries where secret-dependent flows may be non-trivial. Instead, in-memory encryption shifts focus to the data itself. However, such *data-centric* approaches still require identifying all instructions accessing secret data through static and dynamic data flow tracking [70, 83]. Instead, PDM shifts the burden from code discovery to the data itself by making the program *self-declare* any instructions accessing secret data.

Emitting Stolen Instructions. A stolen instruction may itself access a secret which requires an identical code generation as the original faulting instruction explained above. To reduce the number of segmentation faults, we keep checking the next instructions and steal them if they access secrets. Instruction stealing is stopped if the number of stolen bytes is above 5 (or 13), and there are no subsequent secret accesses. Moreover, the stolen instruction may leave the trampoline upon execution (e.g., jmp, ret, call). For direct control-flow transfers, the immediate target address is first extracted and checked to see if the target is inside the trampoline. If yes, an equivalent branch is rebuilt locally with a fresh relative displacement such that the control remains correct. If the branch would leave the trampoline region, the instruction is re-generated to perform the transfer out. On the other hand, indirect control transfers (e.g., ret, leave, indirect jmp and call) which cannot be rewritten ahead-of-time with fixed displacements because their destination is resolved at runtime, are re-emitted verbatim.

B HPC-based Detection Example

Table 11 shows the average HPCs values of different benign and attacker processes. As the results show, cache intensive benign processes often generate higher cache miss rates compared to microarchitectural attacks. Therefore, relying solely on HPC-based monitoring to detect the attacker's process may lead to more false positives.

Table 11: HPC Values of Benign and Attacker Applications

	Application	L1 Miss	L2 Miss	L3 Miss	Ret. Branch	IPC
Benign	600.perlbench	20,742,527	5,606,709	22,957	1,094,853,476	2.73
	602.gcc	134,244,625	30,012,712	12,513,524	115,036,974	0.59
	605.mcf	80,300,752	18,732,390	602,160	463,826,880	0.96
	641.leela	23,496,725	178,280	1,894	447,545,458	1.21
	ML Training	40,107,135	18,397,335	1,506,938	202,214,731	0.94
	`stress-ng -m`	265,352,019	12,641,688	4,233,726	1,143,621,349	1.74
Attack	FLUSH+FLUSH	2,588,355	219	19	11,213,088	0.04
	FLUSH+RELOAD	5,195,358	1,030,469	1,023,499	13,720,576	0.81
	PRIME+PROBE	7,658,456	20,152	15,364	10,478,129	0.26
	Spectre-PHT	31,726,386	26,078,666	4,939,985	12,944,076	0.05