PREDATOR: A Cache Side-Channel Attack Detector Based on Precise Event Monitoring

Minjun Wu
University of Minnesota
Minneapolis, MN, USA
wuxx1354@umn.edu

Stephen McCamant
University of Minnesota
Minneapolis, MN, USA
mccamant@cs.umn.edu

Pen-Chung Yew
University of Minnesota
Minneapolis, MN, USA
yew@umn.edu

Antonia Zhai
University of Minnesota
Minneapolis, MN, USA
zhai@umn.edu

***Abstract*—Recent work has demonstrated the security risk associated with micro-architecture side-channels. The cache timing side-channel is a particularly popular target due to its availability and high leakage bandwidth. Existing proposals for defending cache side-channel attacks either degrade cache performance and/or limit cache sharing, hence, should only be invoked when the system is under attack. A light-weight monitoring mechanism that detects malicious micro-architecture manipulation in realistic environments is essential for the judicious deployment of these defense mechanisms.**

In this paper, we propose PREDATOR, a cache side-channel attack detector that identifies cache events caused by an attacker. To detect side-channel attacks in noisy environments, we take advantage of the observation that, unlike non-specific noises, an active attacker alters victim's micro-architectural states on security critical accesses and thus causes the victim extra cache events on those accesses. PREDATOR uses precise performance counters to collect detailed victim's access information and analyzes location-based deviations. PREDATOR is capable of detecting five different attacks with high accuracy and limited performance overhead in complex noisy execution environments. PREDATOR remains effective even when the attacker slows the attack rate by 256 times. Furthermore, PREDATOR is able to accurately report details about the attack such as the instruction that accesses the attacked data. In the case of GnuPG RSA [20], PREDATOR can pinpoint the square/multiply operations in the Modulo-Reduce algorithm; and in the case of OpenSSL AES [45], it can identify the accesses to the T_e-Table.

I. Introduction

Recently, numerous work has demonstrated the severity of micro-architectural side channels that widely exist but have been hidden in plain sight for decades. Among them, various cache timing side-channels can leak information from commonly deployed security libraries [35], [69] on different cache-related micro-architectural components [10], [21], [47], [51], [64], [66], [68]. These work shows that such cache timing side-channels are particularly important due to their availability, high leakage bandwidth and feasibility. During a cache side-channel attack, a victim's cache access patterns are leaked through the hardware cache micro-architectural states. Various optimization features in cache hierarchy that are integrated on modern micro-processors have created more vulnerabilities for hackers to exploit, and that made them difficult to mitigate completely.

Researchers have proposed various countermeasures, such as randomized caches [6], [33], [34], [50], [52], [58], hardware/OS resource partitioning [32], [57], [60], [72] and execution obfuscation [5], [7], [46] to defend against cache side-channel attacks. However, enabling these mechanisms unconditionally can either degrade cache performance and/or limit necessary data sharing. Therefore, dynamic approaches such as run-time attack detectors [2], [9], [13], [25], [28], [30], [41]–[43], [48], [70], [71] that monitor cache micro-architectural behavior to identify suspicious manipulations have been proposed to effectively address the security concerns with minimal overheads.

When attackers attempt to exploit micro-architecture side-channels, they must manipulate and monitor certain shared micro-architecture structures, and thus potentially leave behind some clues. In addition, when the attacker manipulates shared micro-architecture resources, it can also affect the victim's micro-architecture states in those shared resources, such as cache-line existence. They provide opportunities for run-time detectors to catch active attacks. For example, in the Prime+Probe attack [35], it evicts victim's cache lines by accessing a pre-determined eviction set. When the victim accesses the evicted cache-line, it observes a cache miss that otherwise will not occur. When malicious events are initiated by the attacker (i.e. attacker events), such as eviction set accesses, it can potentially create a side effect on the victim (i.e. victim events), such as unexpected cache misses. These events can be captured by hardware performance counters when they are deployed to monitor run-time micro-architecture events.

Existing work attempts to detect either attacker events [41] and/or victim events [43]. However, those approaches are not very effective in realistic execution environments. In realistic environments, precisely identifying attacker/victim events requires techniques that can effectively filter out noises. On modern computing platforms, multiple applications are often running on the same processor, and potentially can share the same library. The cache contention among these applications can generate noises (i.e.noise events) such as massive cache misses. Noise events can mask victim/attacker events and make them difficult to identify. However, existing work has not provided a comprehensive analysis and methodology for handling noises in detectors. To the best of our knowledge, this is the first paper to thoroughly address the side-channel attack-detection problem in noisy environments.

Compared to environmental noises, both attacker initiated events and their corresponding victim events are highly correlated with victim's security-critical information. In particular, for a run-time detector that are monitoring a victim's execution, noise events are often non-specific, but suspicious victim events are usually concentrated on a few locations that reflect the victim's security-critical information. For example, to attack the Modulo-Reduce operation in a vulnerable RSA implementation, the attacker must monitors instruction accesses after the secret-related branch in the code, and the victim will experience suspicious instruction-cache misses only on certain functions. But cache misses caused by environmental noises are non-location specific and can appear quite random. While identifying those victim events is key to distinguish the interferences caused by an attack from those by the environmental noises, most existing detectors are built on statistical performance counters that can only provide numerical profiling and lack the precise location information to identify victim events.

In this work, we propose a new cache side-channel detector, named PREDATOR, that uses precise performance counters rather than statistical counters to detect victim events. The precise performance

counters sample the execution and returns detailed information such as the target addresses. With the help of precise counters, PREDATOR is able to monitor the user's run-time micro-architecture events and identify suspicious victim events caused by an attack.

PREDATOR's detection algorithm is based a signal-noise analysis that differentiates the location-based deviation from application's noise events. In a normal environment, each cache-line has certain chance to be evicted by other applications. When side-channel attacks occur, security-critical accesses will be affected by the attacker and show abnormal cache events that deviate from normal noise events. To measure this deviation, PREDATOR first builds the application's clean-room cache profile for necessary cache events. Then PREDATOR measures extra cache events under different noise environments and records the access candidates that are sensitive to noise. At run-time, PREDATOR estimates cache events expectation under the current noise level and compares with its observation. If certain accesses significantly exceed the expectation, PREDATOR will raise an alarm and report these suspicious cache events.

PREDATOR is implemented concisely in C with careful optimization to reduce its overhead. We demonstrate that PREDATOR can detect several attacker/victim scenarios in complex execution environments: 1) multiple applications with large memory footprints, and 2) multiple processes sharing the same security critical information. We show that PREDATOR can achieve high accuracy in detecting various attacks, including hard-to-detect Reload+Refresh [10] and the directory-based attacks [66]. We also demonstrate that PREDATOR can detect slow-rate attacks that reduce the attack bandwidth by 256 times in a noisy environment with limited performance overhead.

We would like to mention that PREDATOR is based on existing performance counters, and we demonstrate the advantage of using precise counters instead of statistical counters in realistic noisy environments. We also discuss the limitation of sampling-based performance monitoring and advocate that additional hardware can be used to encapsulate and improve precise counters for various micro-architecture events. We also suggest that the side-channel attack detector should be implemented with software to handle various configuration information and updated for new attacks.

In summary, we have proposed PREDATOR, a practical side-channel detector that is built on existing hardware performance counters to monitor the onset of an attack in noisy environments. In the context of using side-channel detectors, this paper has made the following contributions:

1) It demonstrates the feasibility of building practical side-channel detectors using precise micro-architecture event counters. It shows that, comparing with the non-specific environmental noises, victim events are highly correlated to where the vulnerabilities are, thus it can effectively deal with the environmental noises.
2) It proposes a signal-noise analysis for cache events to identify the location-based deviation (victim events) from noise events.
3) It provides an evaluation of the proposed detection schemes in a wide variety of execution environments, with detailed vulnerability results and analysis.

II. Background and Related Work

In this section, we first review existing work on different types of cache side-channel attacks, and then review the mechanisms that detect and defend against those attacks. We focus on run-time detection techniques that attempt to capture the onset of an attack. Finally, we provide a brief introduction on the precise event sampling mechanisms that are available on most modern microprocessors.

A. Cache Timing Side-Channels

Numerous cache side-channel attacks that exploit various aspect of micro-architecture states have been proposed [4], [10], [21], [23], [24], [35], [51], [64], [69]. The Flush+Reload [69] attack creates a high-bandwidth cache-side channel but relies on the ability to flush the memory with the `clflush` instruction and page sharing between the attacker and the victim. The Prime+Probe [35] attack eliminates these constraints by creating an eviction set. Several proposals, such as Flush+Flush [23] and Evict+Reload [24], extend the concept of the eviction-set to build successful attacks.

More recently, researchers have demonstrated how cache side-channel attacks can be launched without directly monitoring the cache presence states. Rather, these attacks monitor secondary micro-architecture effects of cache accesses. TLBleed [21] creates attacks on the private Translate Look-aside Buffer (TLB) when the victim and the attacker are co-located on the same core. Yan et al. [66] proposes an attack targeting cache coherence directory, instead of caches in server microprocessors that use non-inclusive caches. Reload+Refresh [10] attacks the last-level cache replacement state rather than data presence, consequently, the victim only suffers cache misses in the private cache rather than the last-level cache. Prime+Scope [51] not only primes the cache sets, but also the replacement states. It creates a scope cache line that has high priority in the higher-level caches and low priority in the lower-level caches. Once these cache lines are replaced in the lower-level caches, the attacker will observe a cache miss in the higher-level cache. This allows the side-channel to be constantly open and increases the attack efficiency.

Similar to side-channel attacks, recent work on covert-channel attacks [12], [29], [31] has also been extended to exploit micro-architecture states that are not directly related to the presence of data in the cache. Previous work has shown that LRU states [64], cache coherence states [68], and network-on-chip contention [47] can be exploited to construct covert-channels.

B. Defend Against Cache Side-Channel Attacks

Defending against cache side-channel attacks can be achieved through hardware isolation, operating system resource management, as well as code hardening. Secure cache micro-architectures [27], [33], [34], [52], [56], [58], [61], [65] can close the cache side-channel via mechanisms such as random indexing, robust replacement policies, and partitioning. However, advances in hardware support are often challenged by emerging attacks. For example, some secure randomization cache designs are still vulnerable to schemes that use a probability analysis [6], [50]. When new classes of attacks emerge, existing hardware solutions often lack the flexibility to adapt.

System-level solutions for defending cache side-channel attacks [17], [22], [32], [36], [39], [44], [57], [60], [72] close the side-channel by creating isolation between the victim and the attacker through intelligent resource management, such as cache partitioning [32], [60], memory mapping [57], [72] and process scheduling [36]. While partitioning offers strong isolation and achieves relatively low performance overhead, one challenge is the scalability when the number of cores/processes/security-regions increases. Cache side-channel can also be closed by code obfuscation, either through the source code re-implementation [8], compile-time randomization transformation [7] and constant-timing transformation [5], [46]. However, these software-based approaches often incur significant performance overhead when applied indiscriminately.

Another approach is to detect potential cache side-channels by either identifying vulnerabilities on the victim or by detecting the

presence of attackers. For the former, the detector analyzes the data-flow characteristics and demonstrates potential leakage on given cache models [11], [54], [55], [59]. Brotzman et al. [11] explored run-time execution flow through symbolic execution. Wang et al. [59] targeted compiled cryptography libraries and recovered data-flow through reverse-engineering. Weiser et al. [63] improved detection accuracy using a differential address-trace analysis.

C. Run-time Detection

In cache side-channel attacks, the attackers perform a sequence of well-designed memory operations to interfere with the victim's cache states and infer cache access patterns of the victim. However, such attacker's memory operations in active side-channels can also be detected using Performance Monitoring Unit (PMU) [14]. We can infer an attacker's presence and behavior through the changes in victim's cache states.

Various cache side-channel detectors that identify suspicious attackers or attack behavior have been proposed [2], [9], [13], [25], [28], [30], [41]–[43], [48], [70], [71], and they can be categorized in several ways. First, detectors can choose to detect an attacker's events such as eviction-set accesses or the cache calibration process [41], victim's events such as unexpected cache misses during the execution of security critical regions [43], or both [25]. Since a detector should not raise the alarm to the attacker or any process other than the victim's, detectors that identify attacker events should best report their results to some special hardware. Therefore, most detectors collect cache-related statistics and report to the victim using existing Hardware Performance Counters (HPCs) such as Whisper [43], or use special hardware to analyze and detect side-channel leakage [25], [41]. Second, based on detector's classification algorithm, existing detectors can be either signature-based [9], [13], [43], [48], [70] that use benign/attack scenarios to train the classifier, or anomaly-based [2], [13], [28], [30], [41], [70], [71] that raise alarms to all abnormal behavior based on the execution profile. Lastly, for detection algorithms, most existing approaches use machine-learning based techniques such as Support Vector Machine (SVM) [2], [13], [43], [48], decision tree [43], [71], random-forest [2], [43] and Perceptron/Neuron Network [2], [41], [71].

The effectiveness of a detection scheme can be evaluated using different metrics, which include precision (false positive/negative, precision/recall and F1 score), timeliness, runtime overhead, noise tolerance, as well as the classes of attacks that can be detected [1]. It is very challenging for a detection scheme to do well on all these metrics. Also, most existing detection schemes lack the analysis and discussions on complex issues in execution environments such as noises in the detection. We will present how PREDATOR handles the noise issues in Section III-C.

D. Precise Event Sampling

Existing Intel and AMD PMU support can be configured in a *counting mode* or a *sampling mode*. The counting (or statistical) mode reports the total number of occurrences on some specific events. The sampling mode reports detailed information of one sample after a pre-specified number of such events. Precise sampling, such as Intel Processor Event-Based Sampling (PEBS) or AMD Instruction-Based Sampling (IBS), is an extension of the sampling mode in which the processor reports additional information along with the collected sample.

The Intel precise event sampling feature was first introduced in the Intel NetBurst micro-architecture (2000) [49] and is referred to as Data Event Address Registers (DEAR) in the Itanium processor [37] prior to the Intel Core series. The precise event sampling feature is widely used in performance profiling tools such as Intel VTune [53], PAPI [62], and Linux perf [15]. Previously, precise sampling has been used to detect false sharing in Intel's Haswell architecture [38]. Ferracci [19] suggests using Intel PEBS to detect cache side-channel attacks, but provides no details on how it can be achieved. To the best of our knowledge, our work is the first attempt to use precise event sampling to detect cache side-channel attacks.

As an advanced performance monitoring mechanism, precise sampling only captures a portion of the hardware events, but with much more detailed information on the collected samples. Therefore, it provides a new perspective to address the environmental noise issue in detecting cache side-channel attacks. In the following sections, we discuss how the precise sampling mechanism can be used to detect cache side-channel attacks.

III. PREDATOR: Technique Overview

In a cache side-channel attack, the attacker performs a sequence of well-designed cache manipulations to probe the victim's cache states and infer its cache access patterns. The attacker's probing affects the victim's cache states, and thus can be detected using micro-architecture performance counters that monitor and report cache events to determine an active side-channel attack. However, in a realistic execution environment, attacker-issued probings can hide among cache accesses generated by other co-located benign applications, which we refer to as *environmental noises*. Unlike environmental noises that tends to be more random, attacker cache probings often target specific security-critical data or code regions. Therefore, given the detailed information provided by precise performance counters, such as the addresses of cache misses, we can identify victim events caused by the attacker even in a noisy execution environment. This section introduces PREDATOR's techniques that use existing precise counter mechanisms to detect run-time cache side-channel attacks.

A. Threat Model

For the purpose of our discussion, we classify side-channel attacks along two axes. An attack is *active* if the attacker affects and monitors the victim's micro-architecture states. On the other hand, if the attacker only monitors (but is not affecting) the micro-architecture states, the attack is considered *passive*. An attack can also be classified based on the micro-architecture states it affects and monitors. Most existing side-channel attacks are active attacks on cache memories [10], [23], [35], [51], [66], [69], while some target the port contention [3]. There are relatively few examples of passive side-channel attacks as they are more challenging to construct [47]. This paper focuses on active attacks that affect and monitor cache-line states. Although it is possible to extend the proposed mechanisms to active/passive attacks on other micro-architectural components, the required precise counter information, such as access latencies, is not widely available on current processors.

In the threat model that we address, the attacker and the victim share one or more levels of the cache memory. This most often occurs when the attacker and the victim reside on the same CPU processor but running on different cores. Due to the availability of limited counters, the existing precise event sampling schemes only support limited event types per configuration. In this work, we only attempt to detect cache side channels at the shared last-level cache. With an enhanced counter configuration, it is feasible to detect attacks at the first-level cache LRU state [64] or the Translation Look-aside Buffer (TLB) [21]. If future microprocessors can relax the constraints on existing precise event counters, it is even feasible

to simultaneously detect attacks at multiple levels of the cache memory. The applicability of our technique is thus quite general, but is currently limited by the capability of existing available hardware.

PREDATOR relies on trusted performance counter information. In existing systems, this implies that the operating system and the virtualization layers must be trusted not to tamper with performance counter information. Performance counters are often considered as performance enhancing tools, and thus their security implication is not thoroughly evaluated. In response to the increasing concern over micro-architecture side-channel attacks, we believe that hardware performance monitors can play a significant role. Without additional support, we must include the OS that manages the hardware performance counter in the trusted computing base. However, we believe it is possible to relax this constraint in the future if the hardware manufacturers can include performance counters in the same security domain as the victim's process. A discussion on how to support performance counters inside a security domain is beyond the scope of this paper.

B. Interference in the Last-Level Cache

In this section, we discuss how the presence of an attacker can affect the victim's behavior. When the victim and the attacker are located on different cores within the same processor, the closest shared resource is the last-level cache. Hence, we focus our effort on collecting precise last-level cache events and argue that it is sufficient to detect all existing eviction-based cross-core side-channel attacks.

For a cross-core side-channel attack, there is usually a bi-directional contention between the attacker and the victim. This contention is built on a shared micro-architectural resource with limited capacity, such as the last level cache and coherence directory. Both the attacker and the victim can infer the other's behaviors by monitoring the shared state. The victim's behavior breaks the attacker's manipulation, thus the attacker could probe this change to infer the victim's action. Correspondingly, the attacker's manipulation and/or probing will cause victim a different micro-architectural event, thus it could be captured by a detector.

While existing attacks implement different manipulation techniques, most of them generate the bi-directional last-level cache-related contention between the victim and the attacker. For example, Flush+Reload [69] and Prime+Probe [35] monitor the target cache line and evict it with `clflush` instruction or the eviction set. While Flush+Flush [23] hides the attacker's cache behavior, it also evicts the victim's target cache line with the `clflush` instruction. Similar to Prime+Probe, coherence directory attack [66] applies the same eviction-set idea on the coherence extended directory for non-inclusive caches and leads to victim's private cache misses due to directory contention. Reload+Refresh [10] manipulates the last-level cache set with specific replacement states. To ensure that the victim accesses then changes those replacement states, Reload+Refresh evicts the victim's target cache line from the private cache and keeps it in the last-level cache, which results in victim's last-level cache hit events.

Some researchers propose covert-channels targeting micro-architectures such as the coherence states [68] and the Network-on-Chip [47] before a request arriving the last-level cache. However, monitoring a cross-core victim without interfering the last-level cache is challenging, the attacker still needs to prevent the victim from accessing cache lines in its private cache if these attacks are encapsulated for side-channel purposes.

On the victim side, absence of the attacker, there are three cases of security-critical accesses: 1) private cache hits, 2) last-level cache hits, and 3) last-level cache misses. For the most common case such as a private-cache hit, if the attacker is present, the targeted cache line will be evicted and accessed from the last-level cache or the main memory. Therefore, once it is under a cross-core attack, the victim will observe a distinguishable longer access latencies comparing to a no-attack scenario. We call these unexpected cache events *victim events*, and PREDATOR will try to detect such victim events, and raise an alarm for a potential attack.

If a victim's normal execution shows a pattern of heavy last-level cache accesses, which means the victim has a memory access pattern of excessive private-cache misses, known as *self conflicts* [67], we argue that this access pattern will be hard for an attacker to launch a cross-core side-channel attack because such heavy cache contention will also hinder the attacker's last-level cache manipulation. In this case, it is hard to differentiate the target cache-line accesses from victim accesses for both the detector and the attacker.

From the perspective of designing an effective detector, detecting attacker events is challenging because of various attacker techniques such as the eviction set, the `clflush` instruction, and the manipulation of replacement states or the coherence directory. It is possible to detect them with new improved hardware (e.g. PerSpectron [41]), but they often require a large number of performance counters to have a comprehensive defense. We observe that, given the constraints of launching an effective cross-core attack, manipulating and monitoring micro-architectural states in the shared last-level cache is unavoidable because it is where the attacker and the victim have a commonly-shared system resource. If that is the case, the victim will be able to observe unexpected last-level cache events caused by the attacker. Therefore, monitoring a victim's last-level cache events such as cache hits/misses is effective and necessary to identify any potential cross-core cache side-channel attack.

C. Precise Counters in Realistic Environments

Noises appear widely in a computer system. For a cache side-channel attack, micro-architectural cache events generated by other actively running programs affect both the attacker and the victim. For a security-critical code region such as a cryptographic library, the environmental noises come mainly from two sources: programs running by other applications, and other programs accessing the same library. The first source of noises may create heavy cache contention, evicting the victim's cache lines inadvertently and triggering the attack detector with a false alarm. However, such heavy contention may also disturb an attacker's micro-architectural manipulation and result in a failure of the attack.

With the second source of noises, an attacker's evicted cache lines may be loaded back to the cache hierarchy, and thus mislead the victim's detector with a false-negative error (see Section V-C). However, the attacker will also be affected by such noises, and will observe a confusing cache behavior that comes from a mix of several programs. The attacker thus cannot identify the expected target behavior (Section VI-B), which can result in a failed attack, i.e. the failed detection by the detector is actually not a false-negative error in the sense that it is a failed attack by the attacker.

In this section, we will discuss the first kind of noises in detail, and assume that the victim only accesses the security-critical regions on its own. Existing detectors evaluate noise conservatively. Many detectors did not evaluate co-running applications such as PerSpectron [41]. Some detectors only evaluate a limited number ($\leq$6) of co-running SPEC applications and no co-running application sharing the same library [25], [43]. Furthermore, statistical performance counters that are widely used in existing detectors [2], [9], [13], [30], [41]–

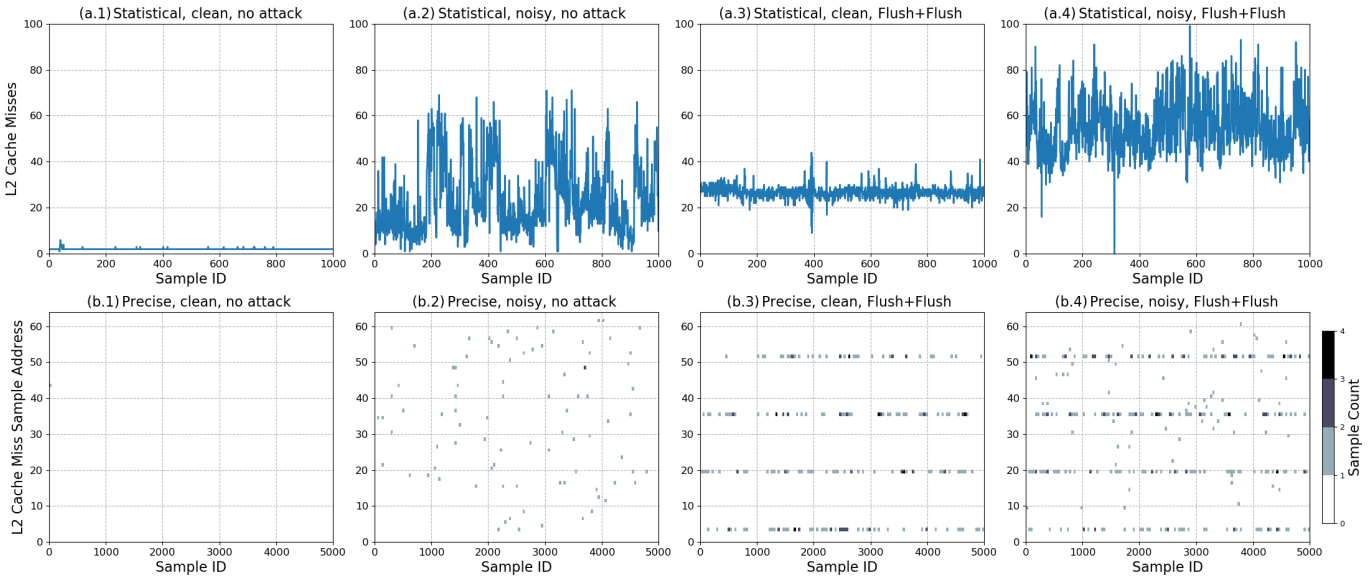

Fig. 1. Detect the cache side-channel attack with different hardware performance counters under different environments. Both statistical and precise performance counters measure the L2 cache miss event. The statistical counter records the event occurrence and the precise counter records the precise access location information. The experiments are evaluated on the Xeon machine in Table II and the noisy environment is simulated with 10 co-running SPEC applications. The attacker attacks 4 T_e-Table accesses in a vulnerable OpenSSL AES implementation [45].

[43], [70], [71], such as those counting the number of cache misses, are subject to error depending on the execution environment.

There are two problems with statistical performance counters that are used to detect a cache side-channel attack in a complex environment. First, many performance counters cannot differentiate between benign cache contention and victim events caused by the attacker. Figure 1 shows some examples of using statistical vs. precise performance counters in different environments. While the cache side-channel attack incurs distinguishable cache misses (a.1 and a.3), the noisy environment can also exhibit high cache misses and lead to potential false positives (a.2 and a.3). Second, statistical performance counters lack methods to measure the environmental noises. A detector based on statistical counters cannot dynamically adjust its threshold and categorize different scenarios, such as grouping a.1/3 and a.2/4 in Fig 1a.

We observe that the attacker's manipulation is strongly correlated to victim's security-critical accesses, but environmental noises are not. In (b.1) through (b.4), if we can obtain precise information about those cache-miss events, such as their addresses, we can effectively differentiate the victim events caused by the attacker (b.3) from the non-specific noises (b.2), even in a noisy environment (b.4). This shows the advantage of using precise performance counters to detect the cache side-channel in a noisy environment over existing statistical counter-based detectors. Furthermore, a precise counter-based detector can calculate the on-going noise level in collected noise samples, and tune the detector's threshold for more precise victim-event recognition.

While an attacker must monitor victim's security-critical accesses to infer the secret, a more advanced attacker has several known techniques to hide its behavior and confuse the detector, e.g. polymorphic attacks, noise injection and slow-rate attacks. A polymorphic attack uses different but equivalent attack operations to vary the attack patterns, such as using the instruction `clflush` instead of eviction-set accesses [16], [41]. However, polymorphic attacks incur the same micro-architectural side effects on the victim, therefore, they are still detectable by the victim-event detector. Noise injection can mislead the detector for the current noise level or leaving a false victim pattern. However, the precise counter-based detector has the capability to handle the noises, and identify both true victim events for security-critical accesses and fake victim events. Later, the detector can record the false victim patterns that are irrelevant to the security critical accesses and prevent denial-of-service attacks. Lastly, the attacker can slow down the attack and reduce the attack bandwidth. It is possible that the sparse victim events may hide the attack within the environmental noises, but an accumulated sample record can eventually distinguish victim events from noises (Section V-D).

D. Processing Sampled Precise Information

A precise event-sampling record contains a rich source of information that may include the instruction pointer (IP), target data address for load/store, access latency, and the event/source type such as hit/miss/pre-fetching in any of L1/L2/LLC/DRAM. From (b1) through (b4) in Fig 1, identifying victim events requires the event type and precise address information. However, processing such precise information in PREDATOR has two challenges.

First, comparing with the statistical sampling techniques, PREDATOR needs to process multi-dimensional data such as sample ID, instruction pointer and access address. Instead of using existing machine learning based detection algorithms such as the decision tree, a new analysis method is required to handle cache-event samples and their spatial information. Second, PREDATOR should provide a way to describe a "normal" execution profile so as to identify an "anomaly" when under an attack, especially, in a complex execution environment. Therefore, the detection algorithm should calibrate the normal execution profile and dynamically measure the environment to adjust run-time estimation. Existing detectors based on statistical counters do not provide such a mechanism.

IV. PREDATOR Design and Implementation

As an anomaly-oriented cache side-channel attack detector, PREDATOR performs run-time signal-noise analysis on data col-

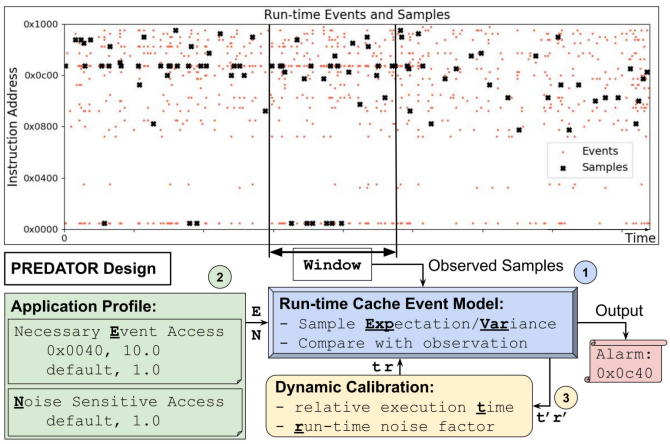

Fig. 2. PREDATOR design overview. PREDATOR consists of three components: 1) a run-time cache-event model, 2) static application profiles, and 3) a dynamic calibration process. In this figure, PREDATOR reads the application profile that indicates necessary cache events on address 0x0040 (approximately 10x more frequent compared to the default). During the detection window, besides of address 0x0040, PREDATOR also observes frequent accesses to address 0x0c40 and reports it as a suspicious signal.

lected by the performance counters to identify irregular cache behavior. As shown in Figure 2, PREDATOR is built based on a run-time cache-event model that calculates the expected cache events, as well as an acceptable range of variances in the presence of environmental noises. PREDATOR compares the observed events and the events predicted by the model to detect anomalies. To compute the expected cache events, the model requires two inputs: static profiles and run-time factors. The static profile describes the application's own cache access characteristics. To effectively handle various execution environments, PREDATOR keeps run-time factors and processes a dynamic calibration that adjusts the expected events based on the observation.

A. Cache-Event Model

The cache-event model in PREDATOR focuses on predicting last-level cache events and comparing with collected samples (Fig 2①). If a particular address location shows an abnormally high cache sampled occurrences, that could be due to 1) victim's self cache contention, 2) noise-sensitive accesses or 3) a sign of the potential attack behavior.

To model total last-level cache events (*Events*), we introduce four variables: normalized noise-affected cache events (N), the run-time noise factor (r), required cache events in the normalized execution (E), and the relative execution time (t). These variables describe the victim's own necessary and noise-affected cache events. Then, we consider the sampling process, which records precise event samples with a fixed event interval (details in Appendix A). In Equation 1, we describe the collected cache event samples (S) in the sampling period (T) and cache events from all accesses (a).

$$S = \frac{Events}{T} = \frac{\sum_a (E_a + N_a r)t}{T} \tag{1}$$

Among these samples, the probability of seeing a target access sample (tar) is shown in Equation 2, and then we can compute the expectation and variance of the target access samples. If any access shows an abnormally large number of samples, it indicates suspicious victim events and a potential run-time attack.

$$p = \frac{(E_{tar} + N_{tar} r)t}{ST} \tag{2}$$

To make Equation 2 valid, we apply two conservative assumptions, and these assumptions infer a binomial distribution $\mathscr{B}(S, p)$. First, we assume that the victim's execution is steady, such as in a repeated program structure (e.g. a loop/recursion), to avoid glitch events. For example, the GnuPG RSA uses a loop/recursion structure to access the secret, which creates an iterative and steady execution. We argue that if the victim's behavior is not repeatable, the attacker will have a difficulty in synchronizing with the victim. In addition, PREDATOR implements a sliding window to filter out occasionally-appeared glitch samples. Second, for a large number of independent processes running on different cores, we assume the overall cache contention is statistically random. Modeling other applications' effect on the cache is difficult. For example, the last-level cache noise model should consider several factors that include execution patterns, private cache locality, and the mix of applications' execution traces. Despite our best effort, we cannot find an exact model in the prior work for our purpose.

To compute the expectation and the variance, PREDATOR needs to know all the variables. The sampling period (T) and the collected samples (S) are known in the precise counter configuration and at run-time. There are four remaining factors: required cache events in normalized execution (E), normalized noise-affected events that are related to noise-sensitive accesses (N), the noise factor (r) and the relative execution time (t). They are determined by a static profiling and run-time calibration presented in the following sections.

B. Static Profiling

An application profile contains information about the victim's own necessary cache events (E) and noise-affected events (N) (Fig 2②). To collect this information, we set PREDATOR in a profiling mode and measure the victim execution's behavior in the absence of any attack. Necessary cache events are thus collected in a "clean room" environment. For noise-affected events, different victim accesses show different sensitivities to the cache contention. For example, if several victim's accesses coincidentally form an eviction set, these accesses are more sensitive to the environmental cache contention. To measure this sensitivity, we run the application in a noisy environment that is created by randomly running 5 SPEC applications with large memory footprints.

Although the victim application appears to have different cache access patterns on different inputs, in the case of the applications we use, which include OpenSSL AES and GnuPG RSA, random inputs are sufficient to collect cache-event profiles. We repeat the measurements 1000 times, and summarize the collected events with a histogram. To save the profiled information storage, we only record significant events by median-absolute-deviation, and the average of the remaining as default values. It turns out the PREDATOR's calibration file is relatively small: it takes 500 bytes for the OpenSSL AES library and 300 bytes for the GnuPG RSA library.

Lastly, the confidentiality of the profiling file is not an issue because attackers can perform a similar measurement to get this information. The profiling file contains the application's normal cache-event information under clean and noisy environments, and thus not a secret. The integrity of the profile file, in which the attacker modifies the content of the file, is beyond the scope of this paper.

C. Dynamic Calibration

The purpose of the dynamic calibration is to measure the relative execution time (t) and the noise factor (r). They are then used to adjust the PREDATOR's expectation about observed cache-event

Noise Level	$r_{curr} * t_{curr}$	t_{curr}	r_{curr}
0	0.000483	0.553181	0.000873
1	0.000322	0.168115	0.001915
2	0.000226	0.052765	0.004283
3	0.000156	0.033269	0.004689
4	0.000151	0.028548	0.005289
5	0.000155	0.025884	0.005988

Table I. Run-time factors $r_{curr} * t_{curr}$ and t_{curr} in OpenSSL AES. Noise level represents the number of co-running SPEC 2017 505.mcf applications. The factor r_{curr} is computed from $r_{curr} * t_{curr}$ and t_{curr}. Each number is experimented with 1,000 repeats and averages the last dynamic calibration result.

samples (Fig 2③). PREDATOR maintains a window that keeps several precise samples. Therefore, instead of starting from the beginning of the execution, PREDATOR's dynamic calibration estimates the relative execution time (t_{curr}) and noise factor (r_{curr}) in the current window.

To compute these factors, as shown in Equation 1, the idea is that the expectation from all accesses should match the observed event samples. For example, if PREDATOR observes fewer samples than expected, it might indicate a shorter execution time or a clean environment. In case the attacker is causing abnormal victim events, we can detect them based on noises and victim's necessary cache events.

We use the accesses in profile files (a) to calculate these two factors because they represent the necessary and cache contention-related events. We name the profiled necessary accesses as Set_E, and the cache contention-sensitive accesses as Set_N. For the noise factor (r_{curr}), we calculate its related factor ($r_{curr} * t_{curr}$) with Set_N, then the relative execution time (t_{curr}) can be obtained by Set_E as shown in Equation 3.

$$\begin{aligned} r_{curr} * t_{curr} &= \frac{\sum_{a \in Set_N}(S_a T)}{\sum_{a \in Set_N}(N_a)} \\ t_{curr} &= \frac{\sum_{a \in Set_E}(S_a T - N_a(r_{curr} * t_{curr}))}{\sum_{a \in Set_E}(E_a)} \end{aligned} \tag{3}$$

We verify our model with the OpenSSL AES application running under different noise environments and present the results in Table I. It shows that with more noises, on average, the application makes relatively slower progress, and the noise factor goes higher.

D. PREDATOR Implementation

PREDATOR is implemented based on the Intel Precise Event-Based Sampling (PEBS) technique. In the set of counters, PREDATOR chooses `frontend_retired.l2_miss` and `mem_inst_retired.all_loads/all_stores` for instruction fetching and data accessing (load/store) events. While both counters cover last-level cache events, they are not perfect: 1) current instruction-fetching counter cannot differentiate a hit from a miss in the last-level cache (only the L2 cache miss), and 2) current data-accesses counter collects all cache hierarchy events from L1 cache hits to the last-level cache misses, which incurs extra sampling overhead and requires a run-time filtering process.

PREDATOR utilizes a helper thread to read and process PMU collected precise samples by periodically waking up the helper thread using the alarm signal. The helper thread maintains a hash table indexed by the event address to record events from the precise counters. It computes the expected occurrence for new recorded sample addresses as described in Section IV-A. PREDATOR keeps a window to avoid glitch spikes and reports suspicious cache events if they have been captured several times. PREDATOR also accumulates the hash-table content as a stateful record to defend against slow-rate attacks (Section V-D).

For the user interface, PREDATOR is implemented as a library that provides APIs for the victim. The API includes detector initialization, start/stop, and final report functions. PREDATOR is initialized with the PMU configuration, the helper thread, and the detection data structure. Start and stop APIs assign the alarm signal, and thus these two functions should wrap the execution of the security-critical region. Lastly, PREDATOR reports the detection result such as suspicious accesses, timeliness and overhead.

	Machine 1	Machine 2
Processor	Intel i7-6700HQ	Intel Xeon Silver 4310
Micro-architecture	Skylake	Icelake
Core Number	8	24
Last-level Cache	6MB inclusive	18MB non-inclusive
OS	Ubuntu 18.04	Ubuntu 20.04

Table II. Experiment Set-up

V. Evaluation

In this section, we evaluate PREDATOR in terms of performance overheads, detection accuracy, detection timeliness and noise tolerance. The experiment set-up is summarized in Table II. Unless otherwise specified, all results are measured and averaged over 1,000 experiments.

1) Attack Techniques: PREDATOR is evaluated with five different types of side-channel attacks on two micro-architectures (Section V-A), and some with a slow-rate attack variation (Section V-D). Xeon 4310 processor provides the non-inclusive last-level cache, and is used to evaluate PREDATOR for the directory-based side-channel attacks [66]. We evaluate PREDATOR's detection capabilities on eviction-set-based side-channel attacks that include Prime+Probe [35] and Reload+Refresh [10] on Intel i7-6700HQ (Skylake) processor because of the known last-level cache hashing functions [40]. Due to the cache's non-inclusiveness, Prime+Probe/Reload+Refresh cannot be implemented on the Xeon machine, and the directory-based attacks are also a challenge on the Skylake machine. Lastly, Flush+Flush [23] and Flush+Reload [69] attacks are evaluated on both machines.

2) Side-Channel Attack Victims: We choose a vulnerable OpenSSL AES implementation [45] (version 1.0.1f) and a vulnerable GnuPG RSA implementation [20] (version 1.4.13) to evaluate PREDATOR's effectiveness. For the AES encryption, the attacker monitors 4 T_e-Tables and attacks 40 thousand rounds of AES encryption to collect statistical information for recovering the key [26]. For the GnuPG RSA, the attacker monitors Square-and-Multiply functions to determine the control flow and to infer individual key bits [69]. We focus on cryptographic applications that have shown to be vulnerable to side-channel attacks for evaluating PREDATOR. Collecting cache side-channel attacks on non-cryptographic applications and evaluating the effectiveness of PREDATOR on these application will be explored in our future work.

3) Execution Environments: To emulate environmental noises, we co-locate multiple memory intensive SPEC 2017 INT benchmarks on the same processor with the attacker and the victim both running. In each experiment, we randomly select N ($0 \leq N \leq 20$) benchmarks to meet the required noise level, and randomly assign these benchmarks to different cores. This process ensures that the environmental noise is different during the profiling and detection runs. With a 24-core Xeon processor, we are able to simulate a wide-range of noise levels. The following benchmarks are used for noise generation because of

Machine 1: Skylake, n=5, slow=1		
	OpenSSL AES	GnuPG RSA
Flush+Flush [23]	99.8%/45%	100%/20%
Flush+Reload [69]	99.9%/44%	100%/21%
Prime+Probe [35]	99.4%/53%	99.2%/38%
Reload+Refresh [10]	97.1%/65%	98.6%/58%
Machine 2: Xeon, n=20, slow=32		
	OpenSSL AES	GnuPG RSA
Flush+Flush [23]	99.7%/31%	98.8%/8%
Flush+Reload [69]	99.9%/34%	98.8%/8%
Directory Attack [66]	95.2%/64%	100%/22%

Table III. PREDATOR's accuracy (F1 score) and timeliness in detecting various attacks under noisy environments. The timeliness is measured by the ratio between reported time and execution completion. All other cores are scheduled with SPEC applications (n=5,20) to generate noise, and PREDATOR is configured with around 10% overhead. On the Xeon machine, the attacker slows down 32 times (slow=1,32). Since it takes a long time constructing the directory eviction-set, the non-inclusive cache directory attack [66] detection only performs 40 experiments.

their large memory footprints: 502.GCC, 505.MCF, 520.OMNETPP, 523.XALANCBMK and 531.DEEPSJENG.

4) Deployment: PREDATOR is implemented based on Intel Performance Monitoring Unit (PMU) and Precise Event-Based Sampling (PEBS). Our experiments are run on two different generations of the Intel processors that use the same PMU configuration and application profiles because the two performance monitoring interfaces are sufficiently consistent. Performing experiments on microprocessors with significant different performance counter interfaces will necessitate re-configuration and additional profile collection.

A. PREDATOR Detection Overview

There are four aspects to evaluate a cache side-channel attack detector. First, it should detect various types of cache side-channel attacks. Second, it should detect the slow-rate attacks with little cache interference (Section V-D). Third, it should detect attacks in a noisy environment (Section V-C). Lastly, it should keep a relatively low performance overhead (Section V-B). PREDATOR achieves all these aspects together with a high detection accuracy.

Table III shows the results of PREDATOR detecting five different kinds of cache side-channel attacks in noisy environments as described in Section V-3. PREDATOR is configured with reduced sampling rates to lower the performance overhead to around 10%. Furthermore, results on the Xeon machine are evaluated with slow-rate attacks that reduces its attack bandwidth by 32 times. PREDATOR achieves low false positives and false negatives on all attacks, and in most scenarios, it is able to detect the attack before the completion of half of the execution. Along with the high accuracy, PREDATOR can also report detailed information about the attack, such as the instructions being attacked.

B. Performance/Timeliness Trade-offs

PREDATOR enables trade-offs between performance and timeliness by tuning the sampling periods of various performance counters. A sampling period for a precise event counter is the event interval between two recorded cache events (see Appendix A). The timeliness is measured by the ratio between the attack report time and the execution completion time when we evaluate an active attack. While late reporting implies more security critical information is being leaked, PREDATOR always detects the attack before the execution of the security region has been completed. For PREDATOR, increasing the sampling period reduces the performance overhead at the cost of the detection timeliness; while decreasing the sampling period leads to a more sensitive detector but incurs a higher overhead.

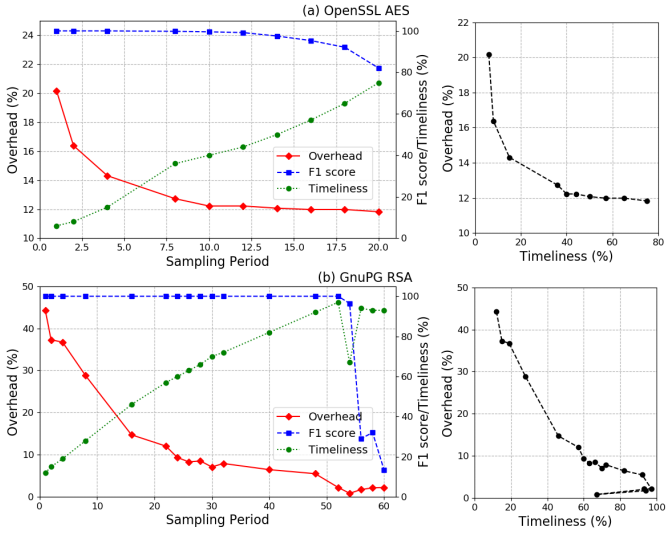

Fig. 3. PREDATOR's performance overhead of OpenSSL AES and GnuPG RSA application on machine 2 (Xeon processor). Results are evaluated in a clean environment. The sampling period shows the slowdown in PMU's shortest performance event sampling period. The overhead is measured as the slowdown when PREDATOR is enabled. For each application, the right side figure shows the overhead-timeliness trade-offs on different sampling periods.

Thus, a trade-off curve between the timeliness and the overhead will determine the optimal sampling period.

Fig 3 shows similar performance/timeliness trade-off characteristics for the two cache side-channel attack victims. Performance results are compared with a baseline in which the application is run with neither PREDATOR nor the attacker. PREDATOR is evaluated for the false positive/negative rates as well as the performance overheads over the increasing PEBS sampling periods until the prediction accuracy dramatically decreases. The false positive rate is evaluated with PREDATOR activated without the attacker, and the false negative rate is evaluated with both PREDATOR and the attacker enabled.

The `F1` scores are calculated using both false-positive and false-negative rates [18]. In Fig 3, both OpenSSL AES and GnuPG RSA show high `F1` scores until the sampling period is too large to collect sufficient events. PREDATOR protects GnuPG RSA with a minimal overhead of around 2.2% and a sampling period of 52 events (100% F1 score). But, a slow sampling rate leads to late detection with a timeliness of 97% (just before one encryption). Since the experiments of the OpenSSL AES are implemented with frequent process synchronization, it incurs extra performance overhead on sampling the blocked processes.

Fig 3 also shows trade-off curves between timeliness and performance overhead. To target the performance overhead of around 10%, we choose a sampling period of 1200 events per sample for OpenSSL AES, and 2300 events per sample for GnuPG RSA. The sampling periods are 12 and 23 times longer than the shortest available sampling period, respectively (see Appendix A). Thus, they will have a corresponding detection overhead of 12.2% and 10.7% with a timeliness value of 44% and 57%, respectively. In GnuPG RSA, that timeliness means that PREDATOR will report anomaly at around half of the encryption process. Although the user can choose a different trade-off, unless otherwise specified, all the results in the evaluation sections are conducted with these two sampling periods.

Noise level, #SPEC apps	0	4	8	12	16	20
OpenSSL AES	99.4%/44%	99.0%/43%	99.1%/41%	99.7%/39%	99.8%/37%	99.9%/35%
GnuPG RSA	100%/59%	100%/59%	100%/59%	99.9%/60%	100%/63%	99.9%/61%

Table IV. PREDATOR's accuracy (F1 score) and timeliness detecting the Flush+Flush attack under different noise levels (number of SPEC applications) on machine 2 (Xeon processor).

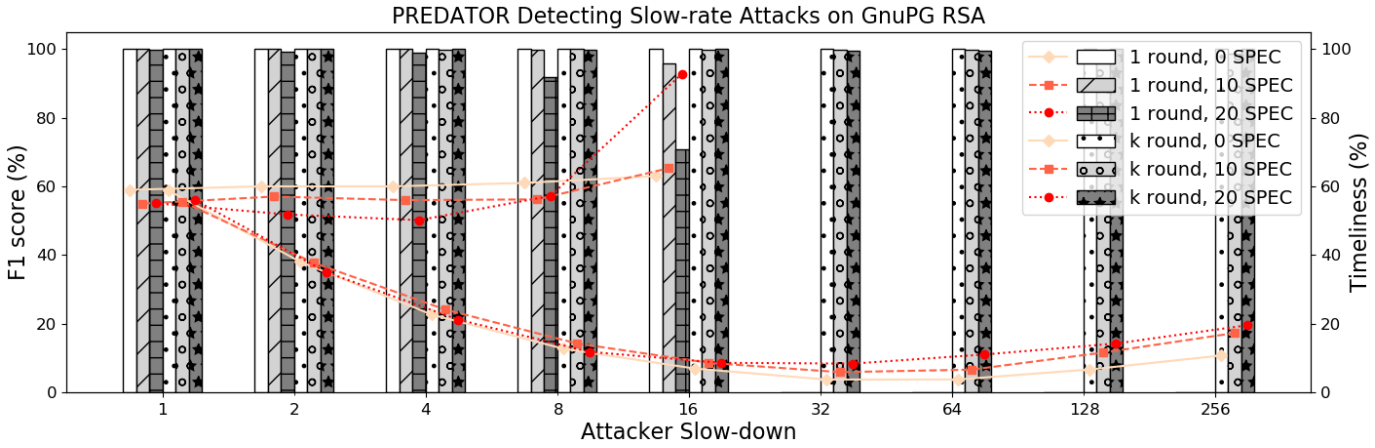

Fig. 4. PREDATOR detecting slow-rate Flush+Flush on GnuPG RSA encryption on machine 2 (Xeon processor). Bar graphs show the accuracy F1 score and plot lines show the timeliness. For detecting within 1 round encryption, PREDATOR fails if the attacker slows down 16 times. But PREDATOR keeps the effectiveness with a stateful record crossing multiple encryption rounds. Round number k in this figure equals to the attack bandwidth slow-down. Evaluation stops at 256 times slow-down due to a long execution time for each experiment.

#shared lib apps	0	3	6
OpenSSL AES	99.8%/37%	91.9%/73%	88.8%/76%
GnuPG RSA	100%/61%	100%/60%	100%/60%

Table V. PREDATOR's accuracy (F1 score) and timeliness detecting the Flush+Flush attack with co-running applications sharing the same library. There are another 15 SPEC applications that run simultaneously to create environmental noises. The experiment is conducted on machine 2 (Xeon processor).

C. Noise Tolerance

In a realistic setting, there are two scenarios that other applications can generate noise events: 1) they are running other programs at the same time that create cache contention, and 2) they are accessing the same security critical information such as security critical libraries. Noise events challenge both the attacker who is observing the side-channel and the defender who is detecting an active attack.

For the noises from cache contention, Table IV shows PREDATOR's F1 scores and timeliness at different noise levels. PREDATOR maintains high accuracy with a similar timeliness in all environments. This confirms our key observation that the environmental noises are non-specific, thus the precise information collected at runtime can effectively differentiate the victim events from the noises. For applications sharing the same security critical information, they may bring the attacker's evicted cache-lines back to the cache hierarchy and affect both the attacker and the victim. While PREDATOR's accuracy decreases as a result as shown in Table V, but attackers are also affected significantly under this scenario. We evaluate and discuss the noise effect on attackers in Section VI-B.

D. Slow-Rate Attacks

The slow-rate attack is a challenging variation for detection because the attacker leaves only a small trace of micro-architecture interference. As mentioned in Section IV-A, if the attacker reduces its attack bandwidth, PREDATOR will have less expectation on the number of abnormal cache event samples, and thus the samples could be hidden in the environmental noises. As shown in the `1 round` results of Fig 4, we can see that PREDATOR has a dramatic decrease in detection accuracy when the attacker is 16 times slower than the baseline attack resolution, especially in a noisy environment.

While a slow-rate attack can hide its trace with less micro-architecture interference, it also pays the price of less information collected in each attack. Therefore, instead of launching a successful attack within one round, the attacker will need multiple rounds of attacks to gain more complete information. PREDATOR can thus detect such multiple attacks with a more stateful detection record. Fig 4 shows that PREDATOR can successfully and accurately detects all slow-rate attacks with a slow-down rate from 1 to 256x within `k rounds`, which means that if the attacker slows down the attack by k times, PREDATOR is able to detect it within k rounds of attacks. This is because accumulated victim events will slowly become distinguishable from the noises. As the victim events are caused by the attacker's probing, such interference monitored by PREDATOR will match the information leakage sought by the attacker. Similar results also observed in the OpenSSL AES scenario, as shown in Table III.

VI. Discussion

Precise performance monitoring shows a great potential in detecting abnormal cache events in complex execution environments. In this section, we discuss several related issues about PREDATOR's performance, the attacks in realistic environments, and possible hardware improvement for precise monitoring.

A. Comparison With Existing Detectors

There are several prior work using statistical PMU counters to detect cache side-channel attacks. Whisper [43] detects three attacks (Flush+Reload [69], Flush+Flush [23] and Prime+Probe [35]) under different attack rates and environments. PREDATOR and Whisper both achieve high accuracy in attack detection in the OpenSSL AES application, and PREDATOR is comparable to Whisper in both detection timeliness and overhead. For example, Whisper detects Flush-based attacks [23], [69] on AES with less than 40% timeliness

Attacks	Noise (#SPEC)	#apps sharing the lib		
		0	1	2
Flush+Reload	0	92.1%	71.8%	70.8%
	3	79.2%	64.5%	63.9%
Prime+Probe	0	69.7%	59.5%	59.5%
	3	63.5%	59.0%	58.5%
Reload+Refresh	0	70.5%	70.2%	69.6%
	3	64.5%	63.4%	63.4%

Table VI. Noise effect on several attack techniques. We measure success rates for the attacker inferring victim one bit correctly under different environments. Note that the 50% success rate means random guess. This experiemnt is conducted on machine 1 (Skylake processor).

(40% and 25% for various attack implementations) and incurs less than 8% overhead.

However, PREDATOR outperforms Whisper in four aspects. First, PREDATOR provides much more detailed information than Whisper. In addition to raising an alarm, PREDATOR can also report attacked victim accesses, which can be efficiently used in defense mechanisms. Second, Whisper is a signature-based detector. It trains a detection model using known attacks, and thus may fail to detect new and unknown side-channels. In addition, Whisper requires training for different attacks and noise environments, which can lead to different configurations. PREDATOR only requires the victims' original execution profiles. Third, Whisper does not demonstrate its effectiveness on Reload+Refresh [10] and directory-based attacks [66], which are challenging for detectors. Lastly, Whisper did not evaluate its ability to detect slow-rate attacks, especially in more complex execution environments. Using statistical counters to detect slow-rate attacks and noises is quite challenging as they show much less interference and hide under noises.

Hardware-based detectors, such as PerSpectron [41] and Cyclone [25] achieve high accuracy and low overhead with special-purpose hardware support. However, PerSpectron only demonstrates its effectiveness in a clean environment with a single core simulation. It will be much more complex to detect various attacks in more realistic environments. Cyclone detects the cycle interference between the victim and the attacker. It does not provide results detecting slow-rate attacks, and it will be challenging for it to detect the multi-party cycles in which multiple attackers are attacking one victim.

PREDATOR detects anomaly by identifying unexpected spikes in cache events that may indicate the onset of a run-time side-channel attack. However, applications may generate spikes in different cache events with different inputs and/or in different phases of a program execution. To handle input-dependent and phase-varying scenarios, PREDATOR collects application profiles that are averaged over multiple runs with different inputs. In the case of OpenSSL AES and GnuPG RSA, random inputs are used. This approach is resilient to input variations and phase changes. However, it may reduce sensitivity to attacks. More future work is needed to allow more effective detection of phase changes and input variations and apply corresponding profiles.

B. Noise Effect on Attackers

In a realistic execution environment, cache behavior caused by other applications might disturb an attacker's cache manipulation. Table VI shows some experimental results of the attacker's success rate in probing the victim in noisy environments, and there are two observations. First, Flush-based attacks are less sensitive to noises compared with techniques that require eviction-sets because noises are more likely to invalidate the eviction sets manipulated by the attacker. Second, when the environment has other running applications that are also accessing the same security critical information, they can create interference that invalidates the attacker's probing. The attacker will observe one of them accessing the target cache-line, but cannot determine whether it comes from the victim or not. In this case, the side-channel still exists, but the mixed behavior of multiple applications can obfuscate the information leakage sought out by the attacker.

C. Security in Performance Monitoring

Existing hardware performance counters are limited in their quantity and thus are shared among physical cores. While this creates a security concern, the sharing problem can be eliminated if per-core performance monitoring hardware is available. For example, to monitor cache accesses through the memory hierarchy, the hardware can implement an extra performance monitoring bit in the Network-on-Chip packet that can trigger the hardware to record additional information. Later, a per-core performance information handler can collect this extra information and process them.

A more challenging issue is related to the untrusted-OS/hypervisor because the software requires OS support to access the PMU. One way to mitigate this issue is to allow direct control of the PMU inside security enclaves. Thus, the previously-mentioned per-core PMU can be configured for an enclave, and the PMU data can be directly writen into the enclave memory for secure recording. In a cloud environment, a virtualized PMU is also possible with the hardware per-core PMU and appropriate context-switching support.

D. Generic Counters for Security

While Cyclone [25] and PerSpectron [41] use hardware detectors to capture micro-architecture attacks, they propose to use specific hardware to detect specific known attacks. To defend against unknown future attacks and to provide flexibility, we advocate using generic counters and applying software-assisted defense mechanisms. By updating the software, information collected by the generic performance counters can be analyzed for defending against new emerging attacks.

VII. Conclusions

In this paper, we propose to build detectors that can capture micro-architecture side-channel attacks using precise event sampling mechanisms. We demonstrate the feasibility of such detectors by building PREDATOR, an effective and efficient cache side-channel attack detector. PREDATOR achieves high accuracy in detecting different types of side-channel attacks, and remains effective even when the attacker attempts to reduce interference by drastically slowing-down the attack bandwidth. PREDATOR also maintains its effectiveness with limited overhead in complex environments with significant noise interference. We believe that processor vendors can enable software-based solutions for countering micro-architecture side-channel attacks by providing precise counters on shared micro-architecture resources. Furthermore, migrating such event counting mechanisms inside the enclave can enable software developers to counter side-channel attacks without relying on the trusted OS and/or virtualization layer.

Acknowledgement

We are very grateful to Professor Huiyang Zhou and the anonymous reviewers for their valuable suggestions and comments. This research was supported in part by NSF under Grants CNS-1514444 and CNS-2106771.

REFERENCES

[1] A. Akram, M. Mushtaq, M. K. Bhatti, V. Lapotre, and G. Gogniat, "Meet the Sherlock Holmes' of Side Channel leakage: a survey of cache SCA detection techniques," *IEEE Access*, vol. 8, pp. 70 836–70 860, 2020.

[2] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance Counters to Rescue: A Machine Learning based safeguard against Micro-architectural Side-Channel-Attacks," *IACR Cryptol. ePrint Arch.*, vol. 2017, p. 564, 2017.

[3] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 870–887.

[4] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[5] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization," 2021.

[6] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1110–1123.

[7] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "DR.SGX: automated and adjustable side-channel protection for SGX using data location randomization," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 788–800.

[8] E. Brickell, G. Graunke, and J. Seifert, "Mitigating cache/timing attacks in AES and RSA software implementations," in *RSA Conference 2006, San Jose, session DEV*, vol. 203, 2006.

[9] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth, "Cacheshield: Detecting cache attacks through self-observation," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 224–235.

[10] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "Reload+Refresh: Abusing cache replacement policies to perform stealthy cache attacks," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1967–1984.

[11] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 505–521.

[12] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 249–266.

[13] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.

[14] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "SoK: The challenges, pitfalls, and perils of using hardware performance counters for security," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 20–38.

[15] A. C. De Melo, "The new linux 'perf' tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[16] S. Deng, W. Xiong, and J. Szefer, "A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 683–697.

[17] X. Dong, Z. Shen, J. Criswell, A. L. Cox, and S. Dwarkadas, "Shielding software from privileged side-channel attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1441–1458.

[18] "F1 score," https://en.wikipedia.org/wiki/F-score.

[19] S. Ferracci, "Detecting Cache-based Side Channel Attacks using Hardware Performance Counters," Master's thesis, Sapienza, University of Rome, 2019.

[20] "GnuPG 1.4.13," https://gnupg.org/ftp/gcrypt/gnupg/, 2012.

[21] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 955–972.

[22] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 217–233.

[23] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.

[24] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015, pp. 897–912.

[25] A. Harris, S. Wei, P. Sahu, P. Kumar, T. Austin, and M. Tiwari, "Cyclone: Detecting contention-based cache information leaks through cyclic interference," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 57–72.

[26] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, Cross-VM attack on AES," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 299–319.

[27] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel, "RIC: relaxed inclusion caches for mitigating LLC side-channel attacks," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*. IEEE, 2017, pp. 1–6.

[28] S. Khan, G. Mruru, and S. Pande, "A Compiler Assisted Scheduler for Detecting and Mitigating Cache-Based Side Channel Attacks," *arXiv preprint arXiv:2003.03850*, 2020.

[29] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[30] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas, "SpyDetector: An approach for detecting side-channel attacks at runtime," *International Journal of Information Security*, vol. 18, no. 4, pp. 393–422, 2019.

[31] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[32] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 406–418.

[33] F. Liu and R. B. Lee, "Random fill cache architecture," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014, pp. 203–215.

[34] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, 2016.

[35] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[36] F. Liu, L. Ren, and H. Bai, "Mitigating Cross-VM Side Channel Attack on Multiple Tenants Cloud Platform," *JCP*, vol. 9, no. 4, pp. 1005–1013, 2014.

[37] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, "The performance of runtime data cache prefetching in a dynamic optimization system," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 180–190.

[38] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti, "Laser: Light, accurate sharing detection and repair," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 261–273.

[39] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 118–129.

[40] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering Intel last-level cache complex addressing using performance counters," in *International Symposium on Recent Advances in Intrusion Detection*. Springer, 2015, pp. 48–65.

[41] S. Mirbagher-Ajorpaz, G. Pokam, E. Mohammadian-Koruyeh, E. Garza, N. Abu-Ghazaleh, and D. A. Jiménez, "PerSpectron: Detecting invariant footprints of microarchitectural attacks with perceptron," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1124–1137.

[42] M. Mushtaq, P. Benoit, and U. Farooq, "Challenges of Using Performance Counters in Security Against Side-Channel Leakage," in *5th International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2020)*, 2020.

[43] M. Mushtaq, J. Bricq, M. K. Bhatti, A. Akram, V. Lapotre, G. Gogniat, and P. Benoit, "Whisper: A tool for run-time detection of side-channel attacks," *IEEE Access*, vol. 8, pp. 83 871–83 900, 2020.
[44] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 227–240.
[45] "OpenSSL 1.0.1f," https://www.openssl.org/source/old/1.0.1/, 2014.
[46] M. Orenbach, Y. Michalevsky, C. Fetzer, and M. Silberstein, "CoSMIX: a compiler-based system for secure memory instrumentation and execution in enclaves," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 555–570.
[47] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 645–662.
[48] A. W. Paundu, D. Fall, D. Miyamoto, and Y. Kadobayashi, "Leveraging KVM events to detect cache-based side channel attacks in a virtualization environment," *Security and Communication Networks*, vol. 2018, 2018.
[49] "Intel® 64 and IA-32 Architectures Software Developer Manuals, Volume 3B," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.
[50] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *42th IEEE Symposium on Security and Privacy*, vol. 5, 2021.
[51] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2906–2920.
[52] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 775–787.
[53] J. Reinders, "Vtune performance analyzer essentials," *Intel Press*, 2005.
[54] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 110–120.
[55] M. Sabbagh, Y. Fei, T. Wahl, and A. A. Ding, "SCADET: a side-channel attack detection tool for tracking Prime+Probe," in *Proceedings of the International Conference on Computer-Aided Design*, 2018, pp. 1–8.
[56] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling ways and associativity," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 187–198.
[57] J. Seo, B. Lee, S. M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs," in *NDSS*, 2017.
[58] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized last-level caches are still vulnerable to cache side-channel attacks! but we can fix it," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 955–969.
[59] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, "Identifying cache-based side channels through secret-augmented abstract interpretation," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 657–674.
[60] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: secure dynamic cache partitioning for efficient timing channel protection," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.
[61] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 494–505.
[62] V. M. Weaver, "Advanced hardware profiling and sampling (PEBS, IBS, etc.): creating a new PAPI sampling interface," Technical Report UMAINE-VMWTR-PEBS-IBS-SAMPLING-2016-08. University of Maine, Tech. Rep., 2016.
[63] S. Weiser, A. Zankl, R. Spreitzer, K. Miller, S. Mangard, and G. Sigl, "DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 603–620.
[64] W. Xiong and J. Szefer, "Leaking information through cache LRU states," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 139–152.

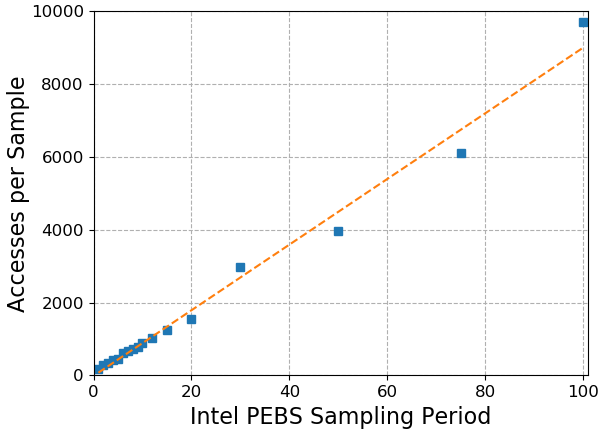

Fig. 5. Relationship between sampling periods and event numbers on Intel PEBS. Longer sampling period will cause larger micro-archiecture events interval between two samples. The experiment is conducted on machine 1 (Skylake processor).

[65] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Atacks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 347–360.
[66] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 888–904.
[67] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: a secure directory to defeat directory side-channel attacks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 332–345.
[68] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 168–179.
[69] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.
[70] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 118–140.
[71] B. Zheng, J. Gu, and C. Weng, "CBA-Detector: An Accurate Detector Against Cache-Based Attacks Using HPCs and Pintools," in *International Symposium on Advanced Parallel Processing Technologies*. Springer, 2019, pp. 109–122.
[72] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 871–882.

APPENDIX A
INTEL PEBS SAMPLING PERIOD

For a sampling based performance monitoring, PMU records one event information on certain amount of micro-architecture events (sampling). The sampling period is the number of events between two recorded micro-architecture events. The sampling period can be programmed and we perform an experiment measuring the relationship between sampling periods and event numbers on Intel PEBS, as shown in Fig 5. For each sampling period, the data point shows the ratio between collected samples and created cache events. If we set the sampling period with 1, Intel PEBS will roughly collect one sample on every 100 micro-architecture events. Adding sampling period by 1 will increase the cache event interval by 100 between two samples (slope=90.14 without considering other accesses and noise).