SoK: Can We Really Detect Cache Side-Channel Attacks by Monitoring Performance Counters?

William Kosasih
The University of Adelaide
Adelaide, Australia

Yusi Feng
Institute of Information Engineering,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China

Chitchanok Chuengsatiansup
The University of Melbourne
Melbourne, Australia

Yuval Yarom
Ruhr University Bochum
Bochum, Germany

Ziyuan Zhu
Institute of Information Engineering,
Chinese Academy of Sciences
Beijing, China

ABSTRACT

Sharing microarchitectural components between co-resident programs leads to potential information leaks, with devastating implications on security. Over the last decade, multiple proposals suggested monitoring hardware performance counters as a method for detecting such attacks.

In this work we investigate these proposals and find that the promising results presented in most are unlikely to carry over to realistic use scenarios. We identify four main shortcomings affecting many of the proposals: implications of detection accuracy, unaccounted performance overheads, undocumented or slow detection speed and a weak threat model. We further find that research artifacts for the vast majority of proposals are not available, significantly hampering the reproducibility and scientific validation of the results.

To overcome the reproducibility issue, we implement a detection scheme similar to those proposed in literature, achieving results similar to those in the literature. We then focus on the last shortcoming—the weak threat model. We observe that the threat model in existing proposals assumes that the attacker uses some variants of published proof-of-concept attacks, without trying to hide the attack. Instead, we propose an attack that modifies a benign program. We demonstrate that such attacks remain feasible, yet display no statistically significant variations in performance counter values. Hence, such attacks cannot be detected by monitoring performance counters. We therefore conclude that despite the large number of proposals, side-channel attack detection with hardware performance counters is not yet ready for real-world deployment.

KEYWORDS

HPC-based detection, cache side-channel attacks, security metrics

1 INTRODUCTION

Microarchitectural attacks have been in the spotlight in recent years given their ability to compromise information confidentiality [1, 9, 10, 30, 34, 36, 44, 49, 52, 62, 63, 65, 69, 70, 74, 96, 97, 99]. Unlike traditional malware that leaves obvious traces of their activities, microarchitectural attacks leave only microarchitectural traces, making them more difficult to detect and mitigate. The effects of these attacks can be severe, ranging from leaking cryptographic keys [1, 9, 10, 30, 34, 39, 52, 63, 65, 70, 96, 97, 99] and other sensitive information [33, 53, 93], to bypassing software [2, 43, 44] and hardware [49, 80, 81] security boundaries.

Recognizing the risks of such attacks, the research community proposed a wide array of modifications to software [13, 17, 50, 73, 100] and hardware [28, 41, 51, 54, 68, 89, 90, 94] with the aim of protecting against such attacks. However, the implementation of these mitigation techniques can be complex. Hardware-based defenses cannot be applied to existing hardware, whereas software countermeasures may have prohibitive performance impacts.

Rather than continuously paying the cost of protection, even in the absence of attacks, an alternative approach aims to detect ongoing attacks and apply countermeasures only when attacks are detected [3–5, 12, 31, 72]. A common approach in that space is based on using Hardware Performance Counters (HPCs), a set of machine-specific registers that monitor microarchitectural events, looking for statistical variations that distinguish adversarial from benign software. The statistical tests used vary from simple threshold techniques [16, 64] to sophisticated machine learning [25, 56], with all proposals boasting high detection accuracy at a minimal performance overhead.

Given the large number of proposals, and the excellent reported results, in this work we ask the following question:

Are published HPC-based detection methods properly evaluated, such that their quality can be ensured for real-world deployment against cache side-channel attacks?

Our Contribution

In this paper, we find that the answer is, unfortunately, negative. We review the correctness of the evaluation of 50 relevant works from the side-channel detection literature, and find four commonly occurring problems in their settings and assumptions, including accuracy, overhead, detection speed, and threat modeling. We further demonstrate how these improper evaluation settings, especially weak threat models assumed by publicly available detection methods, leave them vulnerable to sophisticated attack.

To illustrate this weakness, we develop new camouflaged attacks that mask their malicious execution patterns behind benign program execution, allowing them to evade detection effectively. We show the success of these attacks in stealing sensitive data while being undetected by two publicly available detection methods, despite their high detection success rate for proof-of-concept attacks. To further support our case, we construct our own implementation of a detection method, named *HPCache*, that is capable of detecting proof-of-concept cache attacks with perfect precision, and yet we show that even with such perfect accuracy, it is still gullible to our camouflaged attacks. This emphasizes the importance of assessing detection methods against more advanced attack models.

Through the review and via our own experimentation on the vulnerability of HPC-based cache side-channel attack detection methods, we highlight two key results. Firstly, we demonstrate that the current state of detection methods does not fully support realistic scenarios as in typical practical settings. Secondly, we show that current detection method designs can still be circumvented by smarter and evasive attacks.

In summary, our work makes the following contributions:

- We review HPC-based cache side-channel attack detection methods, proposing four evaluation criteria for these methods and evaluate 50 papers based on their published results.
- We propose a new detection method *HPCache*, with performance comparable to recent studies.
- We develop a variant of the Flush+Reload attack that imitates the actions of a harmless program to mask its malicious intent during execution by camouflaging behind the execution of a harmless program. The attack can effectively steal the complete ElGamal key.
- Using *HPCache* and two other open-source methods [25, 64] as templates, we show how to evaluate detection methods using the four evaluation criteria proposed in this paper.

Since no work fully considers the four crucial evaluation criteria we emphasize, we conclude that without addressing the aforementioned evaluation shortcomings, it is uncertain whether real-time cache side-channel attack detection systems can truly be deemed effective for practical use in real-world scenarios.

2 BACKGROUND

2.1 Caches

In an attempt to boost computing performance, CPU manufacturers implement numerous optimizations. One such optimization is the inclusion of caches, which are a form of intermediary memory that functions as a bridge between the CPU and the random access memory (RAM). Caches store copies of frequently-used data from the RAM and provide data to the CPU with fast access speed while providing low latency.

When a processor tries to access memory, it first checks whether the requested data is stored in the cache. In the event that the data is found in the cache, in other words, a cache hit, the processor can promptly access the data from the cache. On the other hand, if the data is not found in the cache, also referred to as a cache miss, the processor must obtain the data from the main memory (RAM), resulting in a slower access time. Afterwards, the processor stores the retrieved data in the cache, potentially improving future memory access times.

Due to the limited size of caches, when new data needs to be added to the cache, another data that is already in the cached may have to be removed to make room for that new data. Hence, the state of the cache typically reflects recent program execution. Since the cache is shared between multiple programs, the state of the cache may be affected by one of those programs, affecting execution speed of other programs.

Modern cache systems are typically organized into multiple sets, each consisting of several ways, in a set-associative manner. Each memory line is mapped to a specific cache set and can only be stored in one of the ways within that set. Memory addresses that map to the same cache set are called congruent addresses. A set of congruent memory addresses is referred to as an eviction set.

2.2 Cache Side-Channel Attacks

While data eviction due to program activity is harmless in the context of a single process, cross-eviction between different application contexts could potentially be exploited to break information isolation guarantees between applications. By observing timing differences of load operations, an attacker can infer whether or not their data is present in the cache, and therefore learn about the execution of victim program.

This effect has been exploited to create covert channels where two applications assume the roles of Trojan and spy, with the former acting as a transmitter, encoding secret data by bringing a memory address into the cache, and the latter acting as a receiver, timing the read of that memory address. In this case, both the Trojan and the spy are controlled by the attacker, and are essentially reading their own memories. Thus, as far as the system is concerned, there is no explicit communication pathway that occurs between these applications. This essentially breaks any isolation rules that may be put in place within a system.

In the context of side-channel analysis, the victim serves as the transmitter, while the attacker works as the receiver, listening on and inferring secret information. This has been exploited by many to break cryptographic keys [1, 9, 10, 30, 34, 52, 63, 65, 70, 96, 97, 99] as well as stealing keystrokes [33].

2.2.1 *Prime+Probe.*

Prime+Probe [52, 63, 65] is a type of cache attack that takes advantage of the design of set-associative caches to gather information. The attack involves three steps. In the first step, called the prime step, the attacker fills one or more cache sets with their data by repeatedly accessing the addresses of an eviction set. This places the cache in a certain condition premeditated by the attacker to allow for an effective attack. In the second step, the attacker waits for a certain period of time to allow the victim to execute its program, which may include memory accesses. This, in turn, may evict some of the attacker's data out of the cache. That is, in case the victim accesses memory that maps to a cache set previously filled with the attacker's data, the attacker's data will be removed from the cache. In the final step, called the probe step, the attacker measures the time it takes to access the data used in the prime step. If the access time is short, it implies that the data is still in the cache and is not evicted by the victim. If the access time is long, it implies that the victim accessed memory that mapped to that cache set, indicating that the victim accessed data previously stored in that cache set. The attacker can use the mapping between cache sets and address bits to determine the memory address accessed by the victim.

2.2.2 Flush+Reload. The Flush+Reload [33, 34, 96] takes advantage of shared memory in operating systems. To prevent data redundancy, frequently used objects such as libraries are shared among multiple processes by mapping different virtual addresses in different application contexts to the same physical memory. However, a vulnerability arises when sensitive libraries such as those containing cryptographic code are shared between multiple applications. This is because a malicious process gain an ability to influence the time of cryptographic code execution and use it to infer secret keys.

The Flush+Reload attack involves three steps. In the first flush step, the attacker flushes the victim's line of interest from the cache. The attacker is able to perform this action because of memory sharing, which allows the attacker to refer to the same physical memory location as the victim. In the second step, the attacker waits for the victim to execute, which may perform memory accesses. If the victim accesses the memory that the attacker flushed, the memory will be brought back into the cache. If the victim does not access the flushed memory, it will remain out of the cache. In the final step, the attacker measures the time it takes to access the data that it previously flushed. If the access time is short, it means that the victim accessed that data. If the access time is long, it means that the victim did not access that data. Yarom and Falkner [96] demonstrate this attack to steal the private key of RSA by flushing and monitoring the memory containing the square and multiply routine.

2.3 Transient Execution Attack

Transient-execution attacks take advantage of the microarchitectural side effects of instructions that are executed but whose results are never committed to the system's architectural state. Unlike microarchitectural side-channel attacks that can only leak metadata about program execution such as executed instructions or data accesses, transient-execution attacks can directly extract sensitive data from the system. The first two classes of transient execution attacks are Spectre [44] and Meltdown [49].

2.3.1 Spectre. Modern processors are capable of out-of-order processing, which is a technique where the processor schedules instructions that have their data dependencies resolved, and are ready to be executed, regardless of their order in the program code. This means that as soon as their operands are ready, instructions can be processed even before their predecessors complete their execution. To further improve performance by reducing bottlenecks caused by uncertainties in determining which execution path to take, they are also equipped with a branch prediction unit (BPU). When the processor encounters a branching instruction whose operand is not available for use (data not cached, or pending computation), under normal, serial execution, the processor stalls until the data is at hand. This behavior is costly, especially given the relatively long memory fetch time and calculations that may constitute up to hundreds of cycles.

To alleviate this situation, modern processors use a predictive tactic, in which they determine the branch direction that they predict is more likely to be taken once the branch operand is resolved. In case the prediction taken by the processor is wrong, it rolls its execution back to the point where the branch happened and continues along the correct path. This implies that the penalty of such incorrect prediction is no worse than simply idling. However, if the prediction is correct, the processor continues its execution along the correctly predicted branch path as if no delay in memory fetch or computation had occurred. By using this strategy the processor gains performance improvement given a sufficient prediction accuracy.

Branch prediction attacks exploit the BPU by purposefully mistraining it to predict execution paths that are not intended for normal execution such as reading out-of-bounds data or bypassing policy checks [44]. The attack then exploits the out-of-order execution feature to forward this illicit data to a read operation on some buffer. Despite the fact that this ill-informed execution path eventually end up getting rolled-back, the microarchitectural state altered by this stays and can be exfiltrated, leading to data recovery.

Spectre-type attacks are a type of branch prediction attack that takes advantage of the way processors execute instructions after a control or data-flow mis-prediction. The attacker manipulates the branch-prediction unit, causing the processor to speculatively execute instructions that do not appear in the actual instruction stream. This is made possible by multiple prediction units that work together to determine the outcome and target of a branch. By poisoning one or more of these prediction units, Spectre-type attacks direct the processor's execution to "gadgets" which are code snippets that enable the attacker to uncover sensitive data by exploiting microarchitectural state changes.

- **Spectre-PHT** takes advantage of a component in modern processors called the Pattern History Table (PHT), which is part of the branch predictor. The attacker repeatedly trains the PHT to take a certain branch using valid inputs, then executes the targeted code with an out-of-range input that would not take the branch according to the architectural specification, i.e., strictly following the program's logic results in the branch not being taken. This causes the PHT to mis-predict the branch direction, and the processor speculatively executes the instructions with the out-of-range input. The attacker can use this technique to perform out-of-bound reads, and forward the illicit data into a temporary buffer such as the cache for the purpose of retrieval into the architectural state later.
- **Spectre-BTB** is a type of Spectre attack where the attacker leverages the Branch Target Buffer (BTB) to mis-predict the target address of a branch. The attacker repeatedly executes a code snippet with a target address that is not normally taken, which trains the BTB to predict that the target address should be taken. Once the BTB is trained, the attacker then executes the same code snippet with an out-of-bounds target address, causing the processor to speculatively execute code at the out-of-bounds address. This behavior can then be used to perform a return-oriented programming (ROP) attack, where the processor is made to execute a sequence of instructions at unintended locations to perform malicious actions.

2.4 Hardware Performance Counters.

Modern CPUs include hardware performance counters (HPCs), initially introduced for the purpose of debugging [23]. This is done through recording CPU events such as number of cycles, and branch misses. In Intel processors, this functionality is implemented under the name *Performance Monitoring Units* (PMUs). They consist of individual counters called *Performance Monitoring Counters* (PMCs). HPCs can be programmed by setting specific *Machine Specific Registers* (MSRs) in the processor. Despite the vast selection of events that can be chosen from, there are only a small number of PMCs that can be active at any instance. Consequently, attempts to collect more than the number of available counters must resolve to multiplexing, which lowers sampling accuracy.

2.5 HPC-Based Cache Side-Channel Attack Detection Methods

HPCs have been used to facilitate debugging and dynamic profiling [23]. However, recent research has uncovered another area where these counters may be advantageous, namely, in the realm of cache side-channel attack detection [5, 57]. In this section, we examine challenges of using HPCs for security-related purposes. We then explore the classification of HPC-based detection methods for cache side-channel attacks.

2.5.1 Challenges of Using Performance Counter for Security Practices. Das et al. [23] have brought to light issues of non-determinism and contamination when utilizing hardware performance counters for security purposes. This is due to the fact that the usage of non-architectural events, which are specific to the microarchitecture of a processor (e.g., cache accesses, branch prediction, and TLB accesses) for security applications of hardware performance counters can be problematic. These events differ across processor architectures and may also change with processor enhancements. The second issue, contamination, arises because the performance monitoring unit operates at the hardware level and is application agnostic. Therefore, when an interrupt is configured to notify of performance monitoring events, the PMU can generate interrupts for all processes running on a given processor core. To obtain an accurate profile of an application, it is essential to filter the performance counter data relevant only to the process of interest, as performance data can be contaminated by the events of other processes. The authors note that while these issues may not have significant consequences for certain applications, they can have a significant impact on approaches whose security depends on having accurate and consistent hardware performance counter measurements. For instance, malware exploit defences are vulnerable to non-deterministic effects and contamination of events, as security applications rely on small variations in performance counter data to distinguish between suspicious and benign behaviors. Even minor variations of 1-5% in counter values can cause these models to perform poorly. Therefore, it is especially important to address these challenges in security settings.

In line with what was suggested by Das et al. [23], Zhou et al. [102] conducted an experimental study on traditional malware that demonstrates how the use of microarchitectural level information obtained from hardware performance counters (HPCs) cannot differentiate between benignware and malware. Previous HPC-based malware detectors rely on the assumption that malicious behavior affects measured HPC values differently than benign behavior. However, it is debatable and counter-intuitive as to why the semantically high-level distinction between benign and malicious behavior would manifest itself in the microarchitectural events that are measured by HPCs. The authors do not believe that there is a causal relationship between low-level microarchitectural events and high-level software behavior. They argue that the positive results in previous research are due to a series of optimistic assumptions and unrealistic experimental setups.

Furthermore, Jiang et al. [38] evaluate existing detection tools for cache attacks on Secure Guard Extension(SGX), Intel's implementation of a trusted execution environment in x86_64 processors. They identify how these tools fail to accommodate various subtleties in the use of HPCs in the case of exploit prevention and malware detection. They also demonstrate how an adversary can manipulate HPCs to bypass certain security defenses, making detection tools less effective in detecting side-channel attacks on SGX enclaves. Existing detection mechanisms are geared towards an adversary that interferes with the victim's execution to extract the most secret bits, causing significant performance degradation that can signal an attack. However, they show that an adversary leaking smaller portions of secret, as small as a single bit at each execution of the victim, can remain undetected. They specifically demonstrate that an adversary can profile a victim enclave to identify the precise moment during execution when a specific part of the secret can be leaked via a side-channel attack. By running the victim multiple times and leaking a different part of the secret each time, their technique can recover the whole secret while remaining undetected. They adapt known attacks that leverage page tables, L3 cache, or a combination of the two and evaluate their performance on routines on libgcrypt, used by cryptographic algorithms like ElGamal, RSA, and EdDSA. They show that an adversary using their attack technique cannot be detected by existing detection tools unless they tolerate a large number of false positives. They also provide evidence that any detection tool that monitors the performance of the victim is equally likely to fail.

2.5.2 Classification of Cache Attack Detection Methods Using HPCs. Despite the difficulties and literature advising against the use of HPCs in a security context, a considerable number of papers support the application of HPCs to detect cache attacks. We gathered 50 papers related to cache side-channel attack detection methods that use HPCs. To gain a better understanding of these detection techniques, we first present the categories into which they can be grouped. For this purpose, we refer to the study by Akram et al. [5], which classifies academic papers on cache attack detection methods promoting HPC use into two primary groups based on their detection pattern: *signature-based* and *anomaly-based*. Moreover, these detection approaches can be sorted by their classifier type, either as *machine learning-based* or *threshold-based*.

Signature-based detection methods analyze the status of microarchitectural components to identify any patterns that may be indicative of an attack. This technique looks for similarities to known attack patterns. If the HPC readings of an application reach a certain similarity threshold with a known attack pattern, the detection mechanism is triggerred.

Anomaly-based detection techniques continuously scan microarchitectural patterns to search for similarities with a benign application. These methods identify potential attacks by comparing the behavior of monitored applications by reading their HPC values and comparing them with the expected values of a benign application. When an application's readings deviate from what is expected of a harmless application, the detection method flags the application as a possible threat. This is based on the premise that benign applications usually generate a modest number of microarchitectural HPC readings, and any numbers that exceed a certain threshold are considered anomalous and may indicate the presence of cache attacks.

Signature-based detection is more accurate in detecting known attacks, but is more prone to false negatives when faced with new attacks, whereas anomaly-based can possibly detect new attacks but at the same time may experience false positives when execution of benign program unexpectedly changes [7].

The detection techniques for cache side-channel attacks can be further classified based on their classifiers, which are the methods used to determine the likelihood of an attack based on the collected data. There are two main methods of classification:

Threshold-based detection methods uses a simple limit-based classification method, this is done in such a way that when HPC values are above certain threshold, then detection method associates that trace with an attack.

Machine learning-based detection methods use a more advanced way of classifying data, i.e., with the help of machine learning classifiers. The idea behind this approach is to let machine learning algorithms learn features in the collected data. This is done with hopes that the classifier can generalize better over the data, to improve detection accuracy and detect new attacks better.

Mushtaq et al. [61] advocate for the use of HPCs in detecting cache side-channel attacks. They note that in controlled settings with minimal background noise, threshold-based techniques may be adequate for identifying attacks. However, in more realistic and noisy environments, these methods have difficulty distinguishing between benign features and attacks. Therefore, Mushtaq et al. [61] suggest that machine learning-based approaches are better suited for use in these types of settings.

2.5.3 Related work. There are also other detection methods that utilize HPC, but HPC is not their focus. SCAGuard [87] can detect and accurately classify cache side-channel attacks. However, it is based on the attack model derived from extracting attack semantics from the collected data (e.g., HPC, accessed memory) rather than relying on the raw data. SCAGuard uses attack behavior modeling and similarity comparison. First, it introduces the concept of cache state transition enhanced basic block sequences to model attack behaviors. Then, determine whether the program is similar to the original POC through similarity comparison. It only uses a few HPC events as an aid in modeling attack behavior.

3 REVIEW OF HPC-BASED CACHE SIDE-CHANNEL ATTACK DETECTION METHOD EVALUATION

In Section 2.5, we briefly discuss a previous survey by Akram et al. [5] that classified various works on cache side-channel attack detection methods, revealing that 20 of them rely on hardware performance counters (out of 23 in total). Given the growing popularity of this approach, we set out to investigate whether the use of performance counters is a viable option for detecting cache attacks and whether it is an appropriate and effective approach.

As part of our research, we searched scholarly works related to cache side-channel detection using HPCs and identified 50 articles on the subject matter. After conducting an analysis, we identified several recurring issues across these papers that we consolidated into four categories of methodological shortcomings, including improper measurement of accuracy, overhead, and detection speed, as well as a weak threat model used for assessing the effectiveness of detection methods.

In this section, to confirm whether the HPC-based detection methods are effective and feasible, we propose specific evaluation criteria for the above four issues and evaluate whether the evaluations of these four criteria in existing detection method papers satisfy our evaluation criteria. Our findings are summarized in Table 1, where each circle represents the conformance of the proposed method to the criteria described above.

3.1 Accuracy

Undoubtedly, the accuracy of detection is one of the most important indicators for judging detection methods. Assessing the accuracy of cache attack detection methods entails examining both false positive and false negative rates. Keeping a low false negative rate is essential, as detection techniques strive to detect potential attacks on the systems they safeguard, thus reducing the likelihood of overlooking malicious efforts. While having a false negative rate of 0% is the ideal situation, it may be challenging due to the continuous emergence of new threats and the intrinsic disadvantage defenders experience in a "cat and mouse" context with malicious actors perpetually avoiding detection. As such, we do not mandate a specific false negative rate for detection methods to reach. Nevertheless, we expect that research papers evaluate the accuracy of their detection method's online phase, which involves real-time detection within a system, instead of merely gauging the accuracy of their classifier based on stored traces from past program executions.

Out of all the papers we examined, the majority (40 out of 50) evaluate the detection accuracy of their methods. Ten exceptions [7, 21, 24, 26, 29, 46, 64, 67, 76, 82] do not include any online-phase accuracy assessment. We denote these papers with empty circles in Table 1.

Prada et al. [67] does not mention detection accuracy. It shows the PMC values with and without attacks, claiming that its detection method can detect cache attacks against AES based on these values.

False positive rate is another important aspect to consider. A detection method with a high false positive rate can be detrimental to the protected system, as legitimate programs may be incorrectly flagged and terminated. For instance, during a performance test of various detection methods using the SPEC CPU 2017 benchmark, Depoix and Altmeyer [25] consistently misclassified the benchmark program *gcc_r*, which had a high cache miss rate. While this detection method did not actually terminate the program, if it were to do so, as shown by Payer [64], it could disrupt user activity and cause critical applications to be unexpectedly terminated.

Table 1: Review Results. ACC represents accuracy, OV represents overhead, DS represents detection speed, TM represents evaluation against stronger threat models. An empty circle ○ signifies that the criterion has not been assessed, nor mentioned, while a semi-filled circle ◐ implies that the literature mentions the specific criterion but lacks certain attributes required for accurate assessment. A fully-filled circle ● demonstrates that the method conducts its assessment of the criterion appropriately.

Literature	ACC	OV	DS	TM
Ahmad [3]	●	◐	◐	◐
Ahmad [4]	●	◐	◐	◐
Alam et al. [7]	○	○	○	◐
Alam et al. [6]	◐	◐	◐	◐
Albalawi et al. [8]	◐	◐	○	○
Allaf et al. [11]	◐	◐	○	○
Bazm et al. [14]	◐	◐	○	○
Briongos et al. [15]	◐	○	●	◐
Carnà et al. [16]	●	●	●	○
Chiappetta et al. [18]	●	◐	○	●
Cho et al. [19]	◐	◐	●	○
Choudhari et al. [20]	●	○	●	○
Chouhan and Halabi [21]	○	○	●	○
Demme et al. [24]	○	○	○	○
Depoix and Altmeyer [25]	◐	◐	◐	○
Dutta and Sinha [26]	○	○	○	○
Ferracci [29]	○	◐	◐	○
Gregory and Harini [32]	◐	○	○	●
Gülmezoglu et al. [35]	◐	◐	●	●
Hamza et al. [37]	◐	◐	●	○
Kim et al. [42]	◐	◐	◐	○
Kulah et al. [45]	◐	●	○	●
Lantz [46]	○	◐	●	○
Li and Gaudiot [47]	◐	○	◐	○
Li and Gaudiot [48]	◐	○	◐	◐
Mushtaq et al. [56]	◐	○	○	○
Mushtaq et al. [55]	◐	◐	●	○
Mushtaq et al. [58]	◐	◐	●	○
Mushtaq et al. [59]	◐	◐	●	◐
Mushtaq et al. [60]	◐	◐	●	○
Mushtaq et al. [61]	◐	◐	●	○
Payer [64]	○	◐	◐	◐
Polychronou et al. [66]	●	◐	◐	●
Prada et al. [67]	○	○	◐	○
Sabbagh et al. [71]	◐	○	○	○
Schwarzl et al. [72]	●	◐	○	●
Singh and Rebeiro [75]	●	●	○	◐
Tao et al. [76]	○	◐	○	○
Tong et al. [78]	◐	○	◐	○
Tong et al. [79]	◐	○	◐	◐
Vanathi and Chokkalingam [82]	○	○	○	○
Wang et al. [84]	◐	◐	○	◐
Wang et al. [85]	●	○	○	◐
Wang et al. [83]	◐	○	◐	○
Wang et al. [88]	◐	◐	○	●
Wang et al. [86]	◐	◐	○	●
Wu et al. [91]	◐	◐	●	●
Yan and Cui [92]	●	○	◐	○
Zhang et al. [98]	◐	◐	●	◐
Zheng et al. [101]	●	◐	●	◐

To prevent the potential harm caused by mistakenly terminating legitimate applications, Ahmad [3, 4] suggested pausing the identified malicious process and leaving the decision on subsequent actions to the user's discretion. While this approach is suitable for scenarios with a low rate of false positives, it becomes impractical when the rate exceeds a certain threshold as the user would receive an overwhelming number of prompts. Furthermore, relying on user input contradicts the notion of an automated detector.

Schwarzl et al. [72] implemented a Spectre-PHT attack on edge computing service software called Cloudflare Workers. Process isolation can prevent these attacks but incurs high overhead. To reduce the overhead, they propose dynamic process isolation, which isolates only suspicious workers flagged by the HPC-based detection into separate processes.

Consider that detection tools are long-running, that a large number of applications may be running on the protected system, and that detection is performed at a very high rate (multiple tests per second), even a minimal false positive rate can lead to the accumulation of a large number of mislabeled applications. Directly killing these benign applications will affect the normal operation of the protected system, which is unacceptable. Even flagging suspected applications for human assessment may create an unacceptable load, which will lead to ignoring such alerts. Consequently, there is a need to either have an extremely low false-positive rate or have a non-destructive method of handling those.

If a paper reports zero false positive rates, or discusses the potential outcomes of high false positives and gives corresponding mechanisms for dealing with processes deemed malicious, rather than simply killing them, we consider it to have accounted for false positives and give them fully-filled circles. Seven papers [16, 18, 20, 66, 85, 92, 101] meet our criteria of absolute zero false positive rates,[1] and four [3, 4, 72, 75] analyze the potential outcomes of high false positive rates and give corresponding mechanisms, we

[1]We note that a false-positive rate of zero may be the consequence of insufficient evaluation, rather than indicative of a low enough error rate. Our classification gives the papers the benefit of the doubts, assuming that even if the real false positive is above zero, it is low enough to be acceptable.

argue that these eleven papers appropriately evaluate the accuracy criterion.

To minimize the impact of false positives, Singh and Rebeiro [75] modified Linux's scheduler and provided potentially malicious behavior threads with insufficient time and resources to carry attacks. They claimed that their implementation incurred less than 1% overhead on average. Briongos et al. [15] proposed several ways to deal with detected potentially malicious applications, such as adding cache noise and performing dummy operations on meaningless secrets. However, no implementation of such mechanisms is proposed, and they are not evaluated.

For the remaining 29 papers that evaluated the real-time detection accuracy of their methods but had false positive rates above 0% and did not discuss the consequences of high false positive rates, we considered them to partially meet the evaluation criteria for accuracy and gave them half circles.

3.2 Overhead

The term "overhead" refers to the decrease in system speed that occurs when attack detection methods are executed. Performance is one of the most important indicators of a computer. Detection methods must demonstrate their impact on computer performance overhead in order to be selected and deployed.

In order to ensure a fair assessment of the overhead, we recommend that applications under benchmark are either pinned to the same core as the detection method or run on all cores. This guarantees that the detection method influences the benchmark by ensuring that the benchmark is scheduled on the same core as the detection method. This approach prevents the misconception of the detection method being "overhead-free" due to them being scheduled on different cores. Only three papers [16, 45, 75] appropriately examine this criterion. Their detection techniques operates during every context switch (which occurs on each core of the system), signifying that their benchmark application and the detection method run on the same core, thus allowing for a fair overhead assessment.

Out of the 50 papers reviewed, 32 assess their detection overhead, while 18 do not perform any overhead evaluation. For papers that do not perform any overhead evaluation, we assign an empty circle in Table 1. For papers that evaluate their detection method's overhead but do not meet the aforementioned criterion, we assign a semi-filled circle. For the three papers that meet the criterion, we assign a fully-filled circle.

Out of the 18 papers that do not evaluate overhead, five [21, 32, 47, 48, 78] claim that the detection overhead associated with collecting HPC data is low, without evaluating their overhead. We believe that this claim is inadequate to demonstrate the low overhead of their detection system. This is because evaluation of system overhead should include not only the cost of reading HPC data but also the overhead associated with performing attack classification, and potentially scanning the processes running on the system.

Despite measuring their overhead with the benchmark application and data collection process pinned to the same core, we consider the overhead evaluation of Chiappetta et al. [18] to partially cover the true overhead of their detection method. This is because their evaluation only encompasses the overhead of their HPC collection module, ignoring the more resource intensive classification step. Therefore, we assign a semi-filled circle.

Another example on the importance of fair overhead evaluation is demonstrated by Hamza et al. [37] who proposed a mitigation technique for cross-hyperthread *TSX Asynchronious Aborts* (TAA). This method continuously scans for TAA by running a TSX block to check for activities in the sibling thread that causes any aborts. While this method may work, it occupies sibling thread at all time, effectively enabling only one thread to be used for other purposes than the detection method. This is counterproductive to their aim of minimizing cost of introducing a detector for the purpose of preventing the cost of disabling hyperthreading.

3.3 Detection Speed

Assessing the speed of detecting potential attacks is crucial in establishing a safe threshold that users are willing to accept with regard to the possibility of information leakage. Even if a method possesses high detection accuracy, it would not be very useful if it fails to recognize malicious activities in a timely manner, as attackers can steal data before the system detects the attack.

We evaluate research papers based on whether they examine the detection speed of their detection techniques. A thorough evaluation of detection speed involves conducting such an assessment and presenting the results in terms of either the time required to identify attacks or the percentage of attack completion when the attack is detected. We assign fully-filled circles to papers that meet these criteria. Out of 50 papers analyzed, six [19–21, 35, 37, 98] present their detection speed using the former metric, while ten [15, 16, 46, 55, 58–61, 91, 101] use the latter. We consider the latter metric to be more informative because it signifies the maximum amount of key leakage from a cryptographic algorithm. Nevertheless, both approaches are arguably valid, and therefore we assign a fully-filled circle in Table 1 for these 16 papers.

Yan and Cui [92] claim that in a real-world setting, because attackers cannot synchronize with the victim, their execution is much longer than a proof-of-concept attack where synchronization is assumed. They did not test their method's detection speed.

Ferracci [29] evaluate that high sampling frequency results in low accuracy. Too high frequency results in too few information, and therefore it is not possible to distinguish a malicious application from a non-malicious one. When the sampling frequency is too low, detection can be bypassed by inserting code before and after the attack to normalize the HPC readings.

Li and Gaudiot [47], Polychronou et al. [66], Singh and Rebeiro [75], Tong et al. [78], Wang et al. [83] state their data collection interval. However, none of them detail the time for the whole detection process.

Among the remaining 34 papers, 16 [3, 4, 6, 24, 25, 29, 42, 47, 48, 64, 66, 67, 78, 79, 83, 92] provide the detection method's sampling interval but do not evaluate the system's detection as a whole, for which we assigned a half-filled circle. The other 18 [7, 8, 11, 14, 18, 26, 32, 45, 56, 71, 72, 75, 76, 82, 84–86, 88] do not address this criterion at all, to which we assigned an empty circle.

3.4 Threat Model

A significant problem with HPC-based detection methods is that many of them are developed with the assumption that attackers will only use naive, proof-of-concept implementations of attacks. However, this assumption is inaccurate because in reality, attackers are more likely to use advanced techniques to evade detection. Therefore, it is critical to determine whether current detection systems can effectively detect these evasive attacks.

Several papers discuss the feasibility of evasive attacks. Li and Gaudiot [48] suggested that attackers may attempt to understand how the detection method works and mimic the behavior of normal applications to prevent detection. On the other hand, Chiappetta et al. [18] proposed a "smarter" spy process that accesses a random number of addresses generating a random number of cache hits or misses to evade detection. Additionally, Wu et al. [91] discussed the possibility of having an attack that leaks information at a very small rate at a time. The attack is then done over a large number of iterations, recovering more secret as it does so. They argued that this approach may be enough to circumvent detection from detection methods that sample data over a short interval. It is critical that detection systems are proficient in defending against more advanced types of attacks as adversary are likely to use smarter techniques that improve their chances of success.

Next, we present papers that evaluate or improve detection methods against arbitrary evasion attacks. Wang et al. [88] addressed this issue by varying the parameters of attacks such as the number of attack attempts, the time interval between attack attempts, and the number of training in each training phase and the interval between two consecutive training attempts.

Gregory and Harini [32] acknowledged that prior work to detect cache side-channel attacks can be easily bypassed by simply slowing down the attack and interleaving benign code execution between exploit attempts. They claimed that their approach of using undocumented HPCs enabled them to create a detector invulnerable to modifications that break traditional detection methods.

Kulah et al. [45] evaluated their detection method against stealthy spy processes that try to keep their cache activity minimum. They showed that it is still possible to detect these attacks as they still cause abnormal level of cache contentions to achieve their goal.

Gülmezoglu et al. [35] put different amount of sleep between attack steps to avoid detection. They concluded this technique is not adequate to fool their detection method.

Wu et al. [91] demonstrated the possibility of an intelligent adversary that executes attack at a slow rate. While this technique is appropriate for evading detection, especially those that have small window of observation, aggregated HPC data of small microarchitecture traces at a time will eventually accumulate such that distinction from benign execution becomes apparent. They demonstrated their detection system's ability in detecting such attacks. While this argument is certainly valid, it raises the question of how to keep track of the aggregated readings when both the victims and the attacker run in multiple execution context rather than just one.

Polychronou et al. [66] tested their detection method robustness against cache side channel, Spectre, and other microarchitecture attacks with varying eviction interval, insertion of nops, and random sleep functions during the attack. Their insertion of nop instructions and sleep functions is done in a way that does not interfere with the attack. Their detection method was capable of detecting these evasive attacks without false negatives.

Schwarzl et al. [72] discussed two types of camouflaged attacks. Their detection is based on thresholds. First, when the attacker slows down the attack and gets below the threshold, although it results in a false negative, the attacker requires more requests to complete the attack, which can be mitigated by limiting the number of subrequests. Second, an attacker can also attempt to get below the threshold by adding additional code pages. However, a larger code size will cause V8 to abort the attack.

Overall, 23 out of 50 papers considered the possibility of evasive attacks [3, 4, 6, 7, 15, 18, 32, 35, 45, 48, 59, 64, 66, 72, 75, 79, 84–86, 88, 91, 98, 101]. However, only nine [18, 32, 35, 45, 66, 72, 86, 88, 91] evaluated their detection method against any sort of attack modification efforts. For these nine papers, we assigned a fully-filled circle, whereas for the remaining 14 papers, we used a half-filled circle. For the 27 papers that did not acknowledge this issue, we assumed that these works were developed under the assumption of a naive proof-of-concept attacks and give them empty circles.

4 ASSESSING THE QUALITY OF ATTACK DETECTION METHODS

In Section 3, we highlight four performance evaluation criteria for HPC-based cache side-channel attack detection methods. In this section, we apply these four criteria to evaluate publicly accessible detection methods, and since only two methods are reproducible, we also construct and evaluate our own method.

4.1 Experiment Environment

The experiments in this section are conducted on an Intel NUC 9 Extreme Kit that comes with an Intel Core i7-9750H CPU. The system runs on Ubuntu 22.04.

4.2 Our Method

We made efforts to acquire the implementation code from the authors, but we were only able to obtain a limited number of solutions. Out of the total number of papers (50), we found two available online and contacted the authors of the remaining papers via email. We received responses from 21 of them, out of which 13 provided us with the code. However, only the two that were previously open sourced online worked, meaning they were able to compile and perform the detection as expected.

Due to the unavailability of reliable implementations, we were unable to verify their quality. We supplement our experiments with our own cache attack detection solution that uses comparable technique to other proposed methods. We call our method **HPCache**.

It is important to note that HPCache does not aim to offer flawless detection accuracy, minimum performance overhead, nor the ability to detect advanced threat models. Instead, it serves as an illustration of how to apply the evaluation criteria outlined in Section 3.

It consists of three modules: the Process Checker, the Data Collector, and the Classifier. The Process Checker module scans the system for running processes, tracks started and killed processes, and sends process information to the Data Collector module. The

Data Collector module uses the process information from the Process Checker to collect HPC data from each running process in the system every 100 milliseconds. The collected data is associated with the process from which it was sampled and then passed on to the Classifier module. The Classifier module processes the HPC data using a classifier algorithm chosen by the user (in the experiments in this paper, a neural network classifier is used) to determine whether the HPC data is indicative of cache attacks.

To gather performance counter data for our detection, we utilize the PAPI library [77], which offers a consistent interface and approach for accessing the performance counter hardware present in most major microprocessors. The HPC events we use as a data source for our detection method are listed in Table 2. This table includes the names of PAPI events, along with their corresponding descriptions, Intel performance counter mnemonics, and an indication of whether or not the events are derived. If an event is derived, this indicates that it is computed from a combination of multiple underlying performance counter events.

4.3 Accuracy

First, to assess the accuracy criterion, we test the accuracy of these detection methods in detecting standard proof-of-concept attacks. To this end, we have selected an implementation of Spectre-PHT [22], and an implementation of the Flush+Reload attack on GnuPG from the Mastik library [95]. These attacks are chosen due to their significant security implications [44]. In particular, the risk posed by Flush+Reload is noteworthy as it is capable of facilitating the theft of cryptographic keys [52, 63].

We assess the accuracy of our detection technique, along with two other detection methods [25, 64] across an 8-hour time frame, during which we execute multiple benign programs, encompassing the CPU stress-testing application stress-ng, the gcc compiler for compilation, GnuPG for decryption, and the SPEC CPU 2017 gcc_r benchmark. Additionally, we run malicious applications such as Spectre and Flush+Reload on GnuPG. Our detection method is put to the test against 1,000 benign and 1,000 malicious applications.

Table 3 presents our detection method's accuracy compared to that of Payer [64] and Depoix and Altmeyer [25]. To make a fair comparison, we test all detection methods against the same collection of benign and malicious samples. We find that Depoix and Altmeyer [25] only scans for processes running before starting their detection method, therefore we conduct our evaluation by initiating the malicious attack, then launching their detection method, and repeating this sequence for each subsequent experiment. As for Payer [64], we encounter some memory errors after a few minutes of running their detection method and had to modify our testing approach to start a new instance of the detection method for each sample being tested.

Furthermore, we conduct experiments to evaluate the effect of different sampling intervals on the accuracy of our detection method. Table 4 shows that using a sampling interval of 1 or 10 milliseconds leads to a decline in accuracy compared to using a 100 millisecond sampling interval. Under a 1-millisecond sampling interval, both the false negative and false positive rates increase. Similarly, under a 10-millisecond sampling interval, the false negative rate increases, while the false positive rate remains at zero. We conclude that this outcome results from inadequate amount of data being collected within both 1 and 10 millisecond intervals, as both benign and malicious applications have execution periods where the cache miss rate is exceptionally high or low. Collecting HPC data within these intervals fails to capture a comprehensive view of program execution. Therefore, we determine that the sampling interval of 100 milliseconds is optimal for our detection method.

Our tool's detection capabilities and functionality are comparable to others in the field [20, 55–61, 78, 79, 91, 92], as demonstrated by the low false positive and false negative rate of 0%. In conclusion, it is important to conduct accurate evaluations of detection methods' accuracy. A low false negative rate ensures effective protection, and a low false negative rate prevents excessive false positive rate that could render them impractical.

4.4 Overhead

Second, we evaluate the overhead of these detection methods using the technique we recommend for overhead evaluation in Section 3.2. We used the SPECspeed2017_int_base suite, which includes 10 representative benchmarks, such as the GNU C compiler, video compression, and route planning. We initially ran the benchmark application on all cores to establish a baseline CPU speed. Subsequently, we ran the benchmark application on all cores while the detection system was active.

The benchmark results for our method, as well as for the methods proposed by Payer [64] and Depoix and Altmeyer [25], are summarized in Table 5. Since each benchmark took a different length of time, we used geometric means rather than averages to maintain fairness and avoid a large impact on the results from benchmarks that took longer. Using a sampling period of 100 milliseconds, our method resulted in an average slowdown of 1.106. The slowdown of Depoix and Altmeyer [25] was 1.003, which may be due to its implementation flaw of only scanning processes that were running before its detection method started. Payer [64] had minimal impact on CPU performance, which we attribute to its threshold-based approach that does not require complex data classifier algorithms, and its sampling frequency is only 1/10 that of ours.

4.5 Detection Speed

Third, we assess the detection speed of detection methods by measuring the time it takes for them to identify ongoing attacks.

With a sampling interval of 100 milliseconds, HPCache can identify attacks within 300 milliseconds of the execution of a malicious program. By increasing the sampling interval to 10 milliseconds, the tool can detect attacks in just 100 milliseconds. However, we have found that using a sampling interval of 1 millisecond leads to a longer detection time. This is because the amount of information collected during this period is insufficient, as discussed in Section 4.3. Consequently, the detection method can only recognize an attack trace as malicious after it has been running for a longer period, resulting in longer detection times.

For comparison, Payer [64] uses a sampling interval of 1000 milliseconds, and detects attacks within 1100 milliseconds. Regarding Depoix and Altmeyer [25], we were unable to test their detection speed because their method only detects attacks that were executed prior to its start-up, meaning that it cannot detect new attacks.

Table 2: PAPI events used in our detection methods, their description and native performance counter events in x86_64.

PAPI Event Name	Description	Intel Mnemonic	Derived
PAPI_L1_DCM	L1d misses	L1D.REPLACEMENT	N
PAPI_L1_ICM	L1i misses	L2_RQSTS.ALL_CODE_RD	N
PAPI_L1_TCM	L1 misses	L1D.REPLACEMENT, L2_RQSTS.ALL_CODE_RD	Y
PAPI_L2_ICM	L2i misses	L2_RQSTS.CODE_RD_MISS	N
PAPI_L2_TCA	L2 accesses	L2_RQSTS.ALL_CODE_RD, L2_RQSTS.ALL_DEMAND_REFERENCES	Y

Table 3: Accuracy of three detection methods.

Criteria	Payer [64]	Depoix and Altmeyer [25]	HPCache
Number of Datapoints	2000	2000	2000
Number of True Positive	861	843	1000
Number of False Positive	0	500	0
Number of True Negative	1000	500	1000
Number of False Negative	139	157	0
False Negative Rate	13.9%	15.7%	0.0%
False Positive Rate	0.0%	50.0%	0.0%

Table 4: Accuracy of HPCache with 100, 10, and 1 millisecond sampling interval.

Criteria	HPCache 100ms	HPCache 10ms	HPCache 1ms
Number of Datapoints	2000	2000	2000
Number of True Positive	1000	570	577
Number of False Positive	0	0	28
Number of True Negative	1000	1000	972
Number of False Negative	0	430	423
False Negative Rate	0.0%	43.0%	42.3%
False Positive Rate	0.0%	0.0%	2.8%

These findings highlight the discrepancy between the sampling interval of HPC and the detection speed of a detection method. They emphasize the importance of accurately evaluating the detection speed rather than solely stating sampling interval used (as seen in numerous papers in Section 3.3). For example, collecting HPC-data every 100 milliseconds does not guarantee that attacks are detected within such time-frame. Such precise reporting of detection speed enables users to make informed decisions regarding the suitability of detection methods in safeguarding their systems against specific threats.

4.6 Threat Model

Finally, we evaluate the accuracy of the detection methods against stronger attack models. First, we propose camouflaged attacks that hide malicious activities within benign code, and demonstrate the ability of these attacks to recover ElGamal keys. Second, we test the effectiveness of the three aforementioned detection methods in identifying camouflaged attacks. We also compare the accuracy of these detection methods in detecting proof-of-concept attacks to understand the differences between these two threat models, and understand whether weaker threat model assumptions lead to an overestimation of the capabilities of detection methods.

Listing 1: Injection Code

```
void inject_attack() {
  if (rand() < PROB)
    do_fr_gpg();
}

static inline bitmap_element *
bitmap_find_bit (bitmap head, unsigned int bit)
{
  // Inject attack at the beginning of this function.
  inject_attack();

  bitmap_element *element;
  ...
}
```

4.6.1 *Our Camouflaged Attack.*

We injected the Flush+Reload attack and Spectre attack into the SPEC CPU 2017 gcc_r benchmark. The gcc_r benchmark is a C compiler that tests the optimization and code generation capabilities of the CPU. We insert a piece of attack code in the most frequently called function in the benchmark application, and set the probability of executing the attack code.

We choose the *bitmap_find_bit* function which is called the most number of times during the benchmark execution. In Listing 1, we show the insertion of *inject_attack* function at the very beginning of the *bitmap_find_bit* function. In *inject_attack*, we perform the actual Flush+Reload attack sequence against GnuPG with a low probability setting. With these, we essentially interleave the execution of Flush+Reload alongside the actual benchmark. Note that the probability of running the attack code is intentionally very small. Consequently, the execution of the injected program largely resembles that of the actual benchmark. We inject Spectre attack in the same way.

To hide malicious behavior, we schedule the attack infrequently between actual benchmark procedures and ensure that the execution of the attack at each iteration is brief. This results in shorter traces that do not capture the complete key. Furthermore, since we do not assume any synchronization between our attack and the victim's ElGamal encryption algorithm, the traces may begin at any stage of the encryption algorithm. To address these issues, we adopted the approach used by Katzman et al. [40] to recover the complete key from our partial traces.

4.6.2 *The Utility and Cost of our Camouflaged Attacks.*

Our Flush+Reload attack specifically aims to recover the private key from a vulnerable implementation of the ElGamal encryption algorithm [27].

Table 5: Overhead of detection when running all-core SPEC CPU 2017 benchmark.

	Base	Payer [64]		Depoix and Altmeyer [25]		HPCache	
SPEC CPU 2017	Time (s)	Time (s)	Slowdown	Time (s)	Slowdown	Time (s)	Slowdown
600.perlbench_s	263	263	1.000	263	1.000	312	1.186
602.gcc_s	385	384	0.997	384	0.997	431	1.120
605.mcf_s	574	582	1.014	592	1.031	642	1.119
620.omnetpp_s	386	385	0.997	388	1.005	393	1.018
623.xalancbmk_s	257	254	0.988	254	0.988	289	1.125
625.x264_s	162	163	1.006	162	1.000	179	1.105
631.deepsjeng_s	307	309	1.007	309	1.007	342	1.114
641.leela_s	401	402	1.003	401	1.000	440	1.097
648.exchange2_s	289	290	1.004	290	1.004	314	1.087
657.xz_s	552	552	1.000	553	1.002	606	1.098
Geometric Mean	336.0	336.5	1.002	337.1	1.003	371.6	1.106

Table 6: Time required for camouflaged attacks with different probabilities to recover full ElGamal key.

Frequency	Time needed
1/10,000,000	18:13:15
1/1,000,000	1:50:34
1/100,000	00:15:08
1/10,000	00:05:41

The core of this attack lies on the modular exponentiation operation, which involves raising a base b to the power e modulo some modulus m, i.e., calculating $b^e \bmod m$. In the context of ElGamal decryption, the private key serves as the exponent e. Consequently, the attack aims to retrieve the exponent.

In Table 6, we present the results of our camouflaged Flush+Reload attacks, including the frequency of attack injection and the time required for complete key recovery. The frequency column represents the probability of executing the malicious attack injection within the *bitmap_find_bit* function listed in Listing 1.

As shown in the table, when the injected attack is executed with a probability of one in ten million, the time needed to recover the full 459 bits private ElGamal decryption key is approximately 18 hours. However, when the injection is run more aggressively, at the probability of one in ten thousand, the time required to recover the key drops to around six minutes.

4.6.3 Detection Differences Between PoC and Camouflaged Attacks. We conduct experiments to measure the effectiveness of Payer [64], Depoix and Altmeyer [25], and HPCache in detecting attacks, in particular, we test these detection methods capability in detecting both proof-of-concept attacks and camouflaged attacks. We run both Spectre and Flush+Reload attack applications 1000 times and allow the methods a maximum of ten seconds to identify each scenario. If the detection method is able to detect an attack within ten seconds, we consider it a true positive. Otherwise, we consider it a false negative. We also test the detection method against benign SPEC CPU 2017 gcc_r benchmark for 1000 times. If the benchmark is detected as malicious, we consider this a false positive; otherwise, we consider it a true negative.

We summarize the results of our experiments in Table 7, which present the true positive and false positive rates of each detection method in identifying both proof-of-concept and camouflaged Spectre and Flush+Reload attacks. HPCache achieves perfect accuracy in detecting proof-of-concept Spectre and Flush+Reload attacks on GnuPG. However, it fails to detect any of the camouflaged Spectre and Flush+Reload attacks.

Depoix and Altmeyer [25] detect proof-of-concept Spectre and Flush+Reload attacks with 100.0% and 62% accuracy respectively. They also detect 100.0% of the camouflaged as malicious. At the same time, they falsely detect the benchmark without any attack with 100% false positive rate, while HPCache and Payer [64] do not falsely detect the benchmark application.

Payer [64] detect proof-of-concept Spectre and Flush+Reload attacks with 73.1% and 100.0% accuracy respectively, but they fail to detect any of the camouflaged Spectre and Flush+Reload attacks.

The findings show the differences in accuracy when detecting proof-of-concept attacks compared to camouflaged attacks, emphasizing the need to evaluate detection methods against more robust threat models. Clearly, simply stating the defense against a particular attack without providing details of its implementation and threat model can lead to an overestimation of the effectiveness of detection methods.

4.6.4 Re-training Model with Camouflaged Attacks. At first glance, the failure of the detection methods may appear to be caused by the training data not being well-suited to such camouflaged attacks, and a simple re-training of the classifier or adjusting threshold values could solve the problem. As suggested by Depoix and Altmeyer [25], retraining of classifiers is needed when deploying a detector in a new environment or when supporting detection of new attacks.

Our analysis indicates that the root cause of the problem goes beyond inadequate training data. Retraining HPCache to include camouflaged attacks proved ineffective, as it resulted in a sharp increase in false positives, resulting in 100.0% false positive rate. Table 8 shows the result of the detection method trained with camouflaged attack labeled as malicious.

This shows that the detection method in discerning between genuinely benign program execution and the execution of a benign program injected with malicious attacks. Since the detection

Table 7: True/False positive rates of proof-of-concept(PoC) and camouflaged attacks.

		Spectre		Flush+Reload	
Name	Classifier	PoC	Camouflaged	PoC	Camouflaged
Payer [64]	Threshold-Based	73.1%/ 0.0%	0.0%/ 0.0%	100.0%/ 0.0%	0.0%/ 0.0%
Depoix and Altmeyer [25]	Neural Network	100.0%/100.0%	100.0%/100.0%*	62.2%/100.0%	100.0%/100.0%*
HPCache	Neural Network	100.0%/ 0.0%	0.0%/ 0.0%	100.0%/ 0.0%	0.0%/ 0.0%

* Depoix and Altmeyer [25] detected camouflaged attacks with 100% due to false positive in the original SPEC CPU 2017 gcc_r benchmark

Table 8: Accuracy of our detection method when trained with camouflaged attacks

Total number of datapoints	2000
Number of true positives (TP)	1000
Number of false positives (FP)	1000
number of true negatives (TN)	0
number of false negatives (FN)	0
False negative rate	0.0%
False positive rate	100%

method is unable to distinguish between the two, it labels genuinely benign applications as malicious. This is because during the training of the detection method, the training data for the execution of injected attacks is marked as malicious. Consequently, the detection method becomes confused and starts classifying benign applications as malicious.

5 CONCLUSIONS

Prevention and mitigation techniques against cache side-channel attacks have been proposed to counter the ever-increasing threat of these attacks. However, the high cost of hardware solutions has prompted researchers to explore cheaper software-based alternatives, such as HPC-based attack detection methods. In this paper, we reveal that the performance evaluation of current proposed methods are insufficiently conducted to ensure effective protection in practical real-world scenarios. Our analysis of 50 papers reveals that none meets all the necessary criteria of accuracy, overhead, detection speed, and threat model evaluation.

We highlight how the inadequate evaluation of these criteria compromises the protection provided by detection methods. We highlight the importance of conducting accurate evaluation of detection accuracy to ensure effective protection. Additionally, attention should be given to prevent excessive false positives, which can render detection methods impractical and diminish their adoption.

Furthermore, we demonstrate the importance of appropriately evaluating the overhead. An improper setup of benchmark applications can result in unfairly low overhead evaluations. Precise reporting of overhead is crucial since unexpectedly high overhead can impede the adoption and practicality of detection methods.

Additionally, we underscore the importance of properly evaluating detection speed and highlight the difference between the sampling interval of HPC data collection and the detection speed of a detection method. Accurate information about detection speed enables users to make informed decisions about the suitability of detection methods for protecting against specific threats.

Finally, we illustrate how a weak threat model can lead to an overestimation of the effectiveness of detection methods. To illustrate this, we performed an assessment of three cache side-channel attack detection methods, showing them ineffective against our camouflaged attack. Based on these findings, we propose that authors should acknowledge the limitations of their detection methods when they fail to identify attacks under a stronger threat model. Transparent disclosure of such limitations is crucial for users to avoid unexpected compromises due to a lack of information.

In conclusion, we find that HPC-based cache side-channel attack detection methods still have a long way to go before they can be considered practical and widely applicable. We conclude that without addressing the aforementioned evaluation shortcomings, it remains uncertain whether these detection methods can truly be deemed effective for deployment in real-world scenarios.

ACKNOWLEDGMENTS

This research was supported by the Australian Research Council projects DE200101577 and DP210102670; the China Scholarship Council (CSC), grant No. 202104910308; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

Parts of this work were undertaken while Chitchanok Chuengsatiansup and Yuval Yarom were affiliated with the University of Adelaide and while Yusi Feng was in a research visit at the University of Adelaide.

REFERENCES

[1] Onur Aciiçmez and Jean-Pierre Seifert. 2007. Cheap Hardware Parallelism Implies Cheap Security. In *FDTC*. 80–91.
[2] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. 2022. Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution. In *IEEE SP*. 699–715.
[3] Bilal A. Ahmad. 2019. *Detecting Spectre and Meltdown Attacks Using Hardware Performance Counters and Machine Learning*. Ph. D. Dissertation. University of the Punjab.
[4] Bilal A. Ahmad. 2020. Real Time Detection of Spectre and Meltdown Attacks Using Machine Learning. *arXiv preprint arXiv:2006.01442* (2020).
[5] Ayaz Akram, Maria Mushtaq, Muhammad Khurram Bhatti, Vianney Lapotre, and Guy Gogniat. 2020. Meet the Sherlock Holmes' of Side Channel Leakage: A Survey of Cache SCA Detection Techniques. *IEEE Access* 8 (2020), 70836–70860.
[6] Manaar Alam, Sarani Bhattacharya, and Debdeep Mukhopadhyay. 2021. Victims Can Be Saviors: A Machine Learning-Based Detection for Micro-Architectural Side-Channel Attacks. *ACM J. Emerg. Technol. Comput. Syst.* 17, 2 (2021), 14:1–14:31.
[7] Manaar Alam, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Sourangshu Bhattacharya. 2017. Performance Counters to Rescue: A Machine Learning Based Safeguard Against Micro-Architectural Side-Channel-Attacks. *IACR*

Cryptol. ePrint Arch. (2017), 564.

[8] Abdullah Albalawi, Vassilios G. Vassilakis, and Radu Calinescu. 2022. Protecting Shared Virtualized Environments Against Cache Side-channel Attacks.. In *ICISSP*. 507–514.

[9] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *IEEE SP*. 870–887.

[10] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. 2019. Cache-Timing Attacks on RSA Key Generation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019, 4 (2019), 213–242.

[11] Zirak Allaf, Mo Adda, and Alexander E. Gegov. 2019. Malicious Loop Detection Using Support Vector Machine. In *INISTA*. 1–6.

[12] Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Rui Qiao, Reetuparna Das, Matthew Hicks, Yossi Oren, and Todd M. Austin. 2016. ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks. (2016), 743–755.

[13] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *CCS*. 1267–1279.

[14] Mohammad-Mahdi Bazm, Thibaut Sautereau, Marc Lacoste, Mario Südholt, and Jean-Marc Menaud. 2018. Cache-based Side-Channel Attacks Detection Through Intel Cache Monitoring Technology and Hardware Performance Counters. In *FMEC*. 7–12.

[15] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. 2018. CacheShield: Detecting Cache Attacks Through Self-Observation. In *CODASPY*. 224–235.

[16] Stefano Carnà, Serena Ferracci, Francesco Quaglia, and Alessandro Pellegrini. 2022. Fight Hardware with Hardware: System-Wide Detection and Mitigation of Side-Channel Attacks Using Performance Counters. *Digital Threats: Research and Practice* (2022).

[17] Chandler Carruth. 2018. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html.

[18] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. 2016. Real Time Detection of Cache-Based Side-Channel Attacks Using Hardware Performance Counters. *Appl. Soft Comput.* 49 (2016), 1162–1174.

[19] Jonghyeon Cho, Taehun Kim, Soojin Kim, Miok Im, Taehyun Kim, and Youngjoo Shin. 2020. Real-time Detection for Cache Side Channel Attack Using Performance Counter Monitor. *Applied Sciences* 10, 3 (2020), 984.

[20] Amit Choudhari, Sylvain Guilley, and Khaled Karray. 2022. SpecDefender: Transient Execution Attack Defender using Performance Counters. In *ASHES*. 15–24. https://doi.org/10.1145/3560834.3563830

[21] Munish Chouhan and Hasbullah Halabi. 2016. Adaptive Detection Technique for Cache-Based Side Channel Attack using Bloom Filter for Secure Cloud. In *ICCOINS*. 293–297.

[22] Crozone. [n. d.]. Crozone/spectrepoc: Proof of Concept Code for The Spectre CPU Exploit. https://github.com/crozone/SpectrePoC

[23] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In *IEEE (SP)*. 20–38.

[24] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the feasibility of online malware detection with performance counters. *ACM SIGARCH computer architecture news* 41, 3 (2013), 559–570.

[25] Jonas Depoix and Philipp Altmeyer. 2018. Detecting Spectre Attacks by Identifying Cache Side-Channel Attacks Using Machine Learning. *Advanced Microkernel Operating Systems* 75 (2018).

[26] Swastika Dutta and Sayan Sinha. 2019. Performance Statistics and Learning Based Detection of Exploitative Speculative Attacks. In *CF*. 206–210.

[27] Taher ElGamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory* 31, 4 (1985), 469–472.

[28] Yusi Feng, Ziyuan Zhu, Shuan Li, Ben Liu, Huozhu Wang, and Dan Meng. 2021. Constant-Time Loading: Modifying CPU Pipeline to Defeat Cache Side-Channel Attacks. In *TrustCom*. 1132–1140.

[29] Serena Ferracci. 2019. *Detecting Cache-based Side Channel Attacks using Hardware Performance Counters*. Ph. D. Dissertation. Sapienza, University of Rome.

[30] Daniel Genkin, Romain Poussier, Rui Qi Sim, Yuval Yarom, and Yuanjing Zhao. 2020. Cache vs. Key-Dependency: Side Channeling an Implementation of Pilsung. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020, 1 (2020), 231–255.

[31] Jeferson Gonzalez-Gomez, Lars Bauer, and Jörg Henkel. 2023. Cache-based Side-Channel Attack Mitigation for Many-core Distributed Systems via Dynamic Task Migration. *IEEE Transactions on Information Forensics and Security* (2023).

[32] Nick Gregory and Kannan Harini. 2021. Using Undocumented Hardware Performance Counters to Detect Spectre-Style Attacks. (2021).

[33] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*. 897–912.

[34] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In *IEEE SP*. 490–505.

[35] Berk Gülmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2019. FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning. *CoRR* abs/1907.03651 (2019).

[36] Berk Gülmezoglu, Andreas Zankl, M. Caner Tol, Saad Islam, Thomas Eisenbarth, and Berk Sunar. 2019. Undermining User Privacy on Mobile Devices Using AI. In *AsiaCCS*. 214–227.

[37] Ameer Hamza, Maria Mushtaq, Khurram Bhatti, David Novo, Florent Bruguier, and Pascal Benoit. 2021. Diminisher: A Linux Kernel Based Countermeasure for TAA Vulnerability. In *ESORICS*. 477–495.

[38] Jianyu Jiang, Claudio Soriente, and Ghassan Karame. 2022. On the Challenges of Detecting Side-Channel Attacks in SGX. In *RAID*. 86–98.

[39] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. 2023. The Gates of Time: Improving Cache Attacks with Transient Execution. In *USENIX Security*.

[40] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. 2023. The Gates of Time: Improving Cache Attacks with Transient Execution. In *USENIX Security*.

[41] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *DAC*. 60.

[42] Hodong Kim, Changhee Hahn, and Junbeom Hur. 2021. Real-Time Detection of Cache Side-channel Attack Using Non-cache Hardware Events. In *ICOIN*. 28–31.

[43] Ofek Kirzner and Adam Morrison. 2021. An Analysis of Speculative Type Confusion Vulnerabilities in the Wild. In *USENIX Security Symposium*. 2399–2416.

[44] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. (2019), 1–19.

[45] Yusuf Kulah, Berkay Dincer, Cemal Yilmaz, and Erkay Savas. 2019. SpyDetector: An Approach for Detecting Side-Channel Attacks at Runtime. *Int. J. Inf. Sec.* (2019), 393–422.

[46] David Lantz. 2021. Detection of Side-Channel Attacks Targeting Intel SGX.

[47] Congmiao Li and Jean-Luc Gaudiot. 2018. Online Detection of Spectre Attacks Using Microarchitectural Traces from Performance Counters. In *SBAC-PAD*. 25–28.

[48] Congmiao Li and Jean-Luc Gaudiot. 2019. Detecting Malicious Attacks Exploiting Hardware Vulnerabilities Using Performance Counters. In *COMPSAC*. 588–597.

[49] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. (2018), 973–990.

[50] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos V. Rozas, Gernot Heiser, and Ruby B. Lee. 2016. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*. 406–418.

[51] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. 2016. Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks. *IEEE Micro* 36, 5 (2016), 8–16.

[52] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level Cache Side-Channel Attacks are Practical. In *IEEE SP*. 605–622.

[53] Jialin Liu, Ning Miao, Chongzhou Fang, Houman Homayoun, and Han Wang. 2023. Side Channel-Assisted Inference Leakage from Machine Learning-based ECG Classification. arXiv 22304.01990.

[54] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. Dolma: Securing Speculation with the Principle of Transient Non-Observability. In *USENIX Security Symposium*. 1397–1414.

[55] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Vianney Lapotre, and Guy Gogniat. 2018. NIGHTs-WATCH: A Cache-based Side-Channel Intrusion Detector Using Hardware Performance Counters. In *HASP*. 1:1–1:8.

[56] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry, Muhammad Muneeb Yousaf, Umer Farooq, Vianney Lapotre, and Guy Gogniat. 2018. Machine Learning for Security: The Case of Side-Channel Attack Detection at Run-Time. In *ICECS*. 485–488.

[57] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Vianney Lapotre, and Guy Gogniat. 2018. Cache-Based Side-Channel Intrusion Detection using Hardware Performance Counters. In *CryptArchi*.

[58] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Rao Naveed Bin Rais, Vianney Lapotre, and Guy Gogniat. 2018. Run-Time Detection of Prime+Probe Side-Channel Attack on AES Encryption Algorithm. In *GIIS*. 1–5.

[59] Maria Mushtaq, Jeremy Bricq, Muhammad Khurram Bhatti, Ayaz Akram, Vianney Lapotre, Guy Gogniat, and Pascal Benoit. 2020. WHISPER: A Tool for

Run-Time Detection of Side-Channel Attacks. *IEEE Access* 8 (2020), 83871–83900.

[60] Maria Mushtaq, David Novo, Florent Bruguier, Pascal Benoit, and Muhammad Khurram Bhatti. 2021. Transit-Guard: An OS-Based Defense Mechanism Against Transient Execution Attacks. In *ETS*. 1–2.

[61] Maria Mushtaq, Muhammad Muneeb Yousaf, Muhammad Khurram Bhatti, Vianney Lapotre, and Guy Gogniat. 2022. The Kingsguard OS-Level Mitigation Against Cache Side-Channel Attacks Using Runtime Detection. *Ann. des Télécommunications* (2022), 731–747.

[62] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*. 1406–1418.

[63] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*. 1–20.

[64] Mathias Payer. 2016. HexPADS: A Platform to Detect "Stealth" Attacks. In *ESSoS*. 138–154.

[65] Colin Percival. 2005. Cache missing for fun and profit.

[66] Nikolaos Foivos Polychronou, Pierre-Henri Thevenon, Maxime Puys, and Vincent Beroulle. 2021. MaDMAN: Detection of Software Attacks Targeting Hardware Vulnerabilities. In *DSD*. 355–362.

[67] Iván Prada, Francisco D. Igual, and Katzalin Olcoz. 2019. Detecting Time-Fragmented Cache Attacks Against AES Using Performance Monitoring Counters. In *JCC&BD*. 3–15.

[68] Moinuddin K. Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *MICRO*. 775–787.

[69] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*. 199–212.

[70] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. 2019. The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In *IEEE SP*. 435–452.

[71] Majid Sabbagh, Yunsi Fei, Thomas Wahl, and A. Adam Ding. 2018. SCADET: A Side-Channel Attack Detection Tool for Tracking Prime+Probe. In *ICCAD*. 107.

[72] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Daniel Gruss, and Michael Schwarz. 2021. Dynamic Process Isolation. *CoRR* abs/2110.04751 (2021).

[73] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. 2011. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DSN Workshops*. 194–199.

[74] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2020. Robust Website Fingerprinting Through the Cache Occupancy Channel. (2020), 639–656.

[75] Nikhilesh Singh and Chester Rebeiro. 2021. LEASH: Enhancing Micro-Architectural Attack Detection With A Reactive Process Scheduler. *CoRR* abs/2109.03998 (2021).

[76] Xiaojie Tao, Liming Wang, Zhen Xu, and Ru Xie. 2021. SCAMS: A Novel Side-Channel Attack Mitigation System in IaaS Cloud. In *MILCOM*. 329–334.

[77] Daniel Terpstra, Heike Jagode, Haihang You, and Jack J. Dongarra. 2009. Collecting Performance Data with PAPI-C. In *International Workshop on Parallel Tools for High Performance Computing*. Springer, 157–173. https://doi.org/10.1007/978-3-642-11261-4_11

[78] Zhongkai Tong, Ziyuan Zhu, Zhanpeng Wang, Limin Wang, Yusha Zhang, and Yuxin Liu. 2020. Cache Side-channel Attacks Detection Based on Machine Learning. In *TrustCom*. 919–926.

[79] Zhongkai Tong, Ziyuan Zhu, Yusha Zhang, Yuxin Liu, and Dan Meng. 2022. Attack Detection Based on Machine Learning Algorithms for Different Variants of Spectre Attacks and Different Meltdown Attack Implementations. *CoRR* abs/2208.14062 (2022).

[80] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*. 991–1008.

[81] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *IEEE SP*. 88–105.

[82] R. Vanathi and Sp. Chokkalingam. 2018. Cache-Based Side Channel attack Discovery using Intelligent-Detection Algorithm for Securing the Cloud Computing Environment.

[83] Han Wang, Soheil Salehi, Hossein Sayadi, Avesta Sasan, Tinoosh Mohsenin, Sai Manoj P. D., Setareh Rafatirad, and Houman Homayoun. 2021. Evaluation of Machine Learning-Based Detection Against Side-Channel Attacks on Autonomous Vehicle. In *AICAS*. IEEE, 1–4.

[84] Han Wang, Hossein Sayadi, Gaurav Kolhe, Avesta Sasan, Setareh Rafatirad, and Houman Homayoun. 2020. Phased-Guard: Multi-Phase Machine Learning Framework for Detection and Identification of Zero-Day Microarchitectural Side-Channel Attacks. In *ICCD*. 648–655.

[85] Han Wang, Hossein Sayadi, Setareh Rafatirad, Avesta Sasan, and Houman Homayoun. 2020. SCARF: Detecting Side-Channel Attacks at Real-time using Low-level Hardware Features. In *IOLTS*. 1–6.

[86] Limin Wang, Lei Bu, and Fu Song. 2022. Locality Based Cache Side-Channel Attack Detection. *International Workshop* 87 (2022).

[87] Limin Wang, Lei Bu, and Fu Song. 2023. SCAGuard: Detection and Classification of Cache Side-Channel Attacks via Attack Behavior Modeling and Similarity Comparison. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[88] Wubing Wang, Guoxing Chen, Yueqiang Cheng, Yinqian Zhang, and Zhiqiang Lin. 2021. Specularizer: Detecting Speculative Execution Attacks via Performance Tracing. In *DIMVA*. 151–172.

[89] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*. 494–505.

[90] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Sec*. 675–692.

[91] Minjun Wu, Stephen McCamant, Pen-Chung Yew, and Antonia Zhai. 2022. PREDATOR: A Cache Side-Channel Attack Detector Based on Precise Event Monitoring. In *IEEE SEED*. 25–36.

[92] Hui Yan and Chaoyuan Cui. 2022. CacheHawkeye: Detecting Cache Side Channel Attacks Based on Memory Events. *Future Internet* 14, 1 (2022), 24.

[93] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. 2020. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *USENIX Security Symposium*. 2003–2020.

[94] Mengjia Yan, Jen-Yang Wen, Christopher W. Fletcher, and Josep Torrellas. 2019. SecDir: a secure directory to defeat directory side-channel attacks. In *ISCA*. 332–345.

[95] Yuval Yarom. 2016. Mastik: A Micro-Architectural Side-Channel Toolkit.

[96] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*. 719–732.

[97] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2016. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. In *CHES*. 346–367.

[98] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. 2016. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *RAID*. 118–140.

[99] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS*. 305–316.

[100] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: Taking Speculative Load Hardening to the Next Level. In *USENIX Security*.

[101] Beilei Zheng, Jianan Gu, Jialun Wang, and Chuliang Weng. 2022. CBA-Detector: A Self-Feedback Detector Against Cache-Based Attacks. *IEEE TDSC* 19, 5 (2022), 3231–3243.

[102] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. 2018. Hardware Performance Counters can Detect Malware: Myth or Fact?. In *AsiaCCS*. 457–468.