Lightweight Hardware-Based Cache Side-Channel Attack Detection for Edge Devices (Edge-CaSCADe)

Pavitra Bhade, Joseph Paturel, Olivier Sentieys, Sharad Sinha

Lightweight Hardware-Based Cache Side-Channel Attack Detection for Edge Devices (Edge-CaSCADe)

PAVITRA BHADE, Computer Science, Indian Institute of Technology Goa, Ponda, India
JOSEPH PATUREL, INRIA, University of Rennes, Rennes, France
OLIVIER SENTIEYS, INRIA, University of Rennes, Rennes, France
SHARAD SINHA, Computer Science, Indian Institute of Technology Goa, Ponda, India

Cache Side-Channel Attacks (CSCAs) have been haunting most processor architectures for decades now. Existing approaches to mitigation of such attacks have certain drawbacks, namely software mishandling, performance overhead, and low throughput due to false alarms. Hence, *"mitigation only when detected"* should be the approach to minimize the effects of such drawbacks. We propose a novel methodology of fine-grained detection of timing-based CSCA using a hardware-based detection module.

We discuss the design, implementation, and use of our proposed detection module in processor architectures. Our approach successfully detects attacks that flush secret victim information from cache memory like Flush+Reload, Flush+Flush, Prime+Probe, Evict+Probe, and Prime+Abort, commonly known as cache timing attacks. Detection is on time with minimal performance overhead. The parameterizable number of counters used in our module allows detection of multiple attacks on multiple sensitive locations simultaneously. The fine-grained nature ensures negligible false alarms, severely reducing the need for any unnecessary mitigation. The proposed work is evaluated by synthesizing the entire detection algorithm as an attack detection block, Edge-CaSCADe, in a RISC-V processor as a target example. The detection results are checked under different workload conditions with respect to the number of attackers and the number of victims having RSA-, AES-, and ECC-based encryption schemes like ECIES, and on benchmark applications like MiBench and Embench. More than 98% detection accuracy within 2% of the beginning of an attack can be achieved with negligible false alarms. The detection module has an area and power overhead of 0.9% to 2% and 1% to 2.1% for the targeted RISC-V processor core without cache for one to five counters, respectively. The detection module does not affect the processor critical path and hence has no impact on its maximum operating frequency.

CCS Concepts: • **Security and privacy → Side-channel analysis and countermeasures;**

Additional Key Words and Phrases: Cache side-channel attacks, fine-grained monitoring, RISC-V

ACM Reference Format:
Pavitra Bhade, Joseph Paturel, Olivier Sentieys, and Sharad Sinha. 2024. Lightweight Hardware-Based Cache Side-Channel Attack Detection for Edge Devices (Edge-CaSCADe). *ACM Trans. Embedd. Comput. Syst.* 23, 4, Article 56 (June 2024), 27 pages. https://doi.org/10.1145/3663673

Authors' Contact Information: Pavitra Bhade, Computer Science, Indian Institute of Technology Goa, Ponda, Goa, India; e-mail: pavitra19231101@iitgoa.ac.in; Joseph Paturel, INRIA, University of Rennes, Rennes, France; e-mail: joseph.paturel@inria.fr; Olivier Sentieys, INRIA, University of Rennes, Rennes, France; e-mail: olivier.sentieys@inria.fr; Sharad Sinha, Computer Science, Indian Institute of Technology Goa, Ponda, Goa, India; e-mail: sharad@iitgoa.ac.in.

ACM 1539-9087/2024/06-ART56
https://doi.org/10.1145/3663673

1 INTRODUCTION

One of the most significant concerns in the field of Computer Security is the protection from **Cache Side-Channel Attacks (CSCAs)**. By carefully observing the cache behavior, an attacker can deduce valuable insights into the execution of cryptographic algorithms or private keys or even extract sensitive data from other processes running on the same hardware. Cache timing-based side-channel attacks are a class of CSCAs that leverage the timing behavior of cache memory to extract sensitive information from a target system. These attacks take advantage of the variations in access times to cache memory based on whether the data is already present in the cache or needs to be fetched from the main memory. By carefully measuring the time taken to access specific cache lines, an attacker can infer patterns and deduce information about the data being accessed, such as encryption keys or passwords. Our focus in this article is on the detection of such timing-based CSCAs.

The cache memory structure of most processors is such that one level in the hierarchy (the last level in multi-level cache) is shared between multiple cores or between user threads in a single-core setting. In such a case, if the attacker and victim processes reside on the different cores sharing this cache space, the attacker is able to trace the pattern of victim process execution and correspondingly retrieve secret information of the victim by monitoring the victim cache activity. The presence or absence of information in the cache is tracked by the attacker using timing analysis based on cache hits and misses, thereby understanding the execution pattern of the victim and correspondingly retrieving its secret information.

In this article, we propose a fine-grained monitoring approach, which monitors processor events at the level of instructions, functions, and so forth, as opposed to the current approaches, which monitor events at the system or application level.

Modern processor systems have **Hardware Performance Counters (HPCs)** that track the counts of various microarchitectural events during the program execution. Detection of attacks by analyzing the monitored counts for suspicious events is an important research area. However, the research works done in the past, such as [1–13], involve the use of HPCs that count the events generated at a coarse granularity such as at the level of the entire system, application, or process with the help of software tools. The detection of attacks is critical to the application of mitigation techniques and hence should be highly accurate and precise with minimal performance overhead. Usage of microarchitectural event data at a coarse granularity has chances of raising false alarms. Previous works [2–5, 7, 11, 12, 14] have reported false alarms at rates of 47%, 5%, 4%, 0.77% to 1.15%, 1%, 5%, 1%, and 4%, respectively.

False alarms lead to erroneous attack detection and then mitigation is unnecessarily applied. This causes unnecessary performance overload of increased (compared to baseline) execution time of an application though it is not under any attack. Our proposed fine-grained detection technique overcomes these limitations by focusing only on the event counts of the sensitive part of the code in victim applications.

Direct management of HPCs is not possible from user-level accesses to a processor. Privileged access is needed. However, **operating system (OS)** system configurations or usage of software tools allows the filtering and management of the HPCs [15]. Tools like LIKWID [16], Intel Vtune Profiler [17], PAPI [18], PerfMon [19], OProfile [20], and Perf[21] provide data about microarchitectural event counts. They also have the option to filter the counts generated by the HPCs based on some criteria to take a fine-grained view instead of a system-level (application, process) view. However, such tools have certain limitations. They either work as profilers used after execution and hence do not work in real time or have high-performance overhead due to the system calls made by them to fetch the hardware event counts. These tools themselves can have significant code sizes and hence require their own installation and configuration. This approach

is unsuitable for real-time detection *where periodic sampling of the code section is involved,* which we aim at through our proposed lightweight approach.

In this article, we discuss the design and implementation of a cache timing attack detection framework, **Edge-CaSCADe**, that includes a hardware-based attack detection module, associated software-based identification, and marking of the sensitive code. Custom hardware performance counters, part of the attack detection block, count the cache misses of only the sensitive sections of victim code. Since the processor events corresponding to only the sensitive sections of victim code are considered, we think of this as a fine-grained and lightweight approach. To our knowledge, there does not exist prior work that monitors the microarchitectural events in a fine-grained manner for attack detection, though coarse-grained approaches exist, as discussed in the related work.

The following are the major contributions of this article:

(1) We propose a custom hardware block, Edge-CaSCADe, inclusive of hardware performance counters, for hardware-based runtime detection of timing-based CSCAs that evict the victim information from cache memory, e.g., Flush+Reload, Flush+Flush, Prime+Probe, Prime+Abort, Evict+Probe, and so forth. The details of this Edge-CaSCADe are mentioned in Section 4.
(2) We implement the custom hardware block as a fine grained detection module to count address-level events for attack detection. This approach reduces false alarms when compared with the current approaches that use system or process level hardware performance counters. The demonstration is discussed in Section 2 and through Figure 1.
(3) We further propose an algorithm (Algorithm 3) to identify and count only the cache misses that could be raised due to suspicious activity, to reduce false alarms, as discussed in Section 4.2.1.
(4) We demonstrate that our proposed method can detect multiple attacks by monitoring multiple secrets simultaneously, as discussed in Section 5.2.
(5) Our detection algorithm can detect an attack after extracting only 5% of the bits from a key by an attacker in the best case, and less than 15% in the worst case, as demonstrated in Table 6.
(6) We have evaluated the efficacy of our solution on multiple use cases with varying workloads, including the MiBench [22] and Embench [23] benchmark suite along with the RSA-, AES-, and ECC-based encryption algorithm ECIES as test cases. Our results show more than 98% detection accuracy with negligible false alarms and at a detection time of 2% from the beginning of the attack for the cases we tested.
(7) Our method works with minimal performance overhead for victim application as the detection framework works at the microarchitecture level, without involving any system calls to measure the event counts. Our experiments show that area overhead over a very simple core goes up to only 2% and power overhead goes up to only 2.1%, in the case of five hardware performance counters, with a maximum path delay of 0.62 ns, thus making the framework very lightweight and efficient.

The rest of the article is structured as follows. In Section 2, we discuss the related work in the area of detection of CSCAs, mainly the ones using hardware performance counters. In Section 3, we talk about the attack types we have targeted in our detection approach. In Section 4, we discuss all the steps in our proposed Edge-CaSCADe framework. In Section 5, we mention the details of our experimental setup and show the results from our experiments on the Comet RISC-V simulator. We also discuss the hardware implementation and related overhead. In Section 6, we discuss how our methodology addresses the issues and challenges of existing HPC-based detection approaches. Lastly, we conclude in Section 7 by shedding some light on the future scope of this research.

Table 1. Comparison to Existing Hardware Techniques of Defense against Cache Side-channel Attacks

Research	Year	Methodology	Drawback
[24]	2020	Bypass the Last Level Cache access for CLFLUSH instructions	Does not mitigate non-flush-based attacks
[25]	2020	Instruction comparisons with attack-prone instructions	Only detects attacks caused due to specific instructions
[26]	2021	Cache is compressed, fits more data in each set	Reduces impact of attack but does not mitigate it
[27]	2022	Micro-instructions are compared with pre-existing attack snippets	Detects attacks with known snippets, storage overhead
[28]	2022	Reconfigurable framework that monitors internal signals and events to detect attacks	Done at coarse granularity of system level, could lead to false alarms

2 RELATED WORK

Detection of CSCAs has been a key area of research, especially in Intel and ARM processor architectures. However, recently, this domain has been gaining pace in the RISC-V community as well, as they are prone to timing-based cache attacks too. Hardware, software, and hybrid approaches have been adopted by the researchers comparing the pros and cons of the methodology used. For example, software approaches provide better results on existing architectures but have performance overhead. On the other hand, hardware approaches prove to have minimal performance overheads but need architectural modifications, proving them unuseful for existing devices.

In [24], the authors tried to bypass the cache access from the Last Level Cache for the instructions that are flushed by the CLFLUSH instruction. However, this technique does not defend against other attacks that evict the secret information from the cache using other methods like Evict+Probe, Prime+Probe, and Prime+Abort attacks. In [25], a reconfigurable hardware module is used that checks for underlying suspicious events to detect attacks by comparing it with a timer or flush instruction as they are the ones used by the attacker to perform the attack. However, this may not work when the attacker tries some other instructions or uses these instructions in a different order compared to the one mentioned in their detection algorithm prototypes. Hence, the authors claim to be able to detect only two types of attacks.

In [26], the effectiveness of Flush+Reload attack is seen to be reduced by 30% to 50% by using a compressed cache design. However, this design does not completely mitigate the attack and also impacts the cache performance by a factor of 2.9. In [27], a hardware detection module is implemented that checks the micro-instructions under execution and compares them against a set of suspicious instruction snippets to detect attacks. However, there is a memory requirement to store these snippets for comparison, which would increase with the number of attacks to be detected, leading to area and power overheads. In [28], a reconfigurable framework is developed to monitor internal signal events and detect suspicious activities. However, this work monitors the events at the system level and not at the fine-grained level, which may cause false alarms. Also, there is no mention of multiple attacks being detected simultaneously.

Table 1 summarizes the above-mentioned work.

Another category of interesting research work relies on attack detection techniques using HPCs. HPCs are used not only to detect CSCAs but also for exploit detection [29–32], malware detection [33–38], firmware verification [39, 40], integrity checking [41, 42], and vulnerability analysis [43]. Our work, however, only overlaps with the focus on the detection of CSCAs. Table 2 shows a list of works based on coarse-grained HPCs for the detection of CSCAs. In [1], three different

Table 2. Existing Coarse-grained Hardware Performance Counter-based Attack Detection Approaches

Research	Year	Tool
[1]	2016	Perf and PAPI
[2]	2018	Intel CMT
[3]	2019	Intel CMT
[4]	2019	Intel PCM
[5]	2019	Intel PCM
[6]	2021	Intel PCM
[7]	2021	Custom RISC-V tool
[8]	2021	Perf and ARM PMU
[9]	2021	Perf
[10]	2021	Intel Vtune Profiler
[11]	2022	Intel PT and Perf
[12]	2023	PAPI
[13]	2023	Intel PCM

machine learning techniques are adopted on counts by HPCs to detect the attacks. The counts are fetched using the Linux *Perf* tools and cause an overhead of 2.3%. In [2], the authors use the **Intel Cache Monitoring Technology (ICMT)** counters to detect cross-vm cache attacks by applying the Gaussian anomaly approach. In [3], again ICMT is used to fetch hardware counter values to detect cache attacks. In [4], a machine learning approach is used on counters of the Intel **Performance Counter Monitor (PCM)** to detect the attacks in real time. In [5], unsupervised deep learning is performed on the Intel PCM to predict microarchitectural attack risks. In [6], the non-cache-related event counts from HPCs are extracted in real time, which are then fed to machine learning techniques to detect attacks. Similarly, in [7], detection of transient execution-based attacks is achieved using machine learning techniques on HPCs of an out-of-order RISC-V processor. A similar approach is adopted in [8], where the edge classifier prototype is implemented on ARM and x86-based SoCs. In [9], a fully supported *Perf* engine is used to detect spectre attacks with more than 90% accuracy.

In [10], the authors have fine-grained the counts, improving the system-level count approach. However, the drawback is that it does not work in real time, nor is the detection in hardware. In [11] also, the authors have claimed that by only using existing HPCs that give counts at the entire system level, the accuracy as well as false alarm rate in attack detection is hampered. Hence, they have used the locality of the **Control Flow Graph (CFG)** along with HPC data in the detection approach to map HPCs with the attack-sensitive locations in the target program CFGs. However, their method uses tools like Angr [44], Intel Processor Trace [45], and Perf [46] to fetch HPC counts and CFG mapping, which leads to a high performance overhead of about 12%. In [12], the authors use the PAPI tool to extract HPC values and then find the best fit features to detect the attack based on anomalies. In [13], machine learning classifiers are trained with the system-wide HPC counts to detect suspicious programs or attacks. While our primary focus remains on eviction-based CSCAs, it is pertinent to note the existence of fault attacks and fault detection schemes within cryptographic contexts. [47, 48] and [49] focus on detection of such fault attacks in RSA- and ECC-based algorithms. However this study does not overlap with our target attacks and is out scope. In Table 2, we have listed the main works that deal with the detection of CSCAs using coarse-grained HPCs and the tools used to fetch the counter data.

The work in [50] also mentions the advantage of using tailor-made HPCs over the existing HPCs for the purpose of attack detection. The custom counters mentioned in their work are a combination of multiple events in an order being treated as a single event, which increases the chances of accurate attack detection due to more accurate patterns being monitored. However, in this work too, the counts are still being taken at coarse granularity and fetched using system calls, which involve events generated by all the processes under execution and cause performance overhead. Also, in [51], the Intel Precise Event-based sampling approach is also said to show lags and shadowing, causing errors in event capturing.

In [52, 53], and [54] the authors have summarized the drawbacks and pitfalls of using HPCs with software tools for attack detection. First, the tools used to fetch the HPC counts make system calls each time to fetch the counts and then analyze them by feeding them to the classifiers. These system calls cause high performance overhead. Also, the training samples and the testing environment of the machine learning classifiers used differ when the HPC values are obtained in a virtualized and bare-metal environment. This impacts the overall accuracy of classification. They also point out that the division of data for offline training and testing sets can lead to errors in real-life testing. While mentioning the experimental drawbacks, they also state that no sufficient cross-validation for the machine learning classifiers is performed. Inconsistencies during context switching also are highlighted. Our proposed method addresses most of these drawbacks, as discussed in Section 6.

To summarize, our proposed method detects the timing-based CSCAs where the attacker evicts the key-sensitive section of the code from the cache memory periodically to understand the victim activity for that section, correspondingly revealing the key. The order in which the instructions are executed does not matter in this case, as periodic eviction will be done by the attacker to reveal each key bit, causing a high number of cache misses experienced by the victim. Hence, our technique is applicable to in-order as well as out-of-order processors, prone to timing-based CSCAs. We focus on monitoring only the secret sections of the victim, giving it a fine-grained approach to reduce false alarms. We have also improvised the counting method to only count events that could be suspicious, further reducing false alarms. Also, our detection module is performed as an addition to the hardware, leading to minimal performance overhead. Our work also overcomes the other above-listed drawbacks that include real-time monitoring and concurrent multiple attack detection.

3 BACKGROUND ON FLUSH-BASED CACHE SIDE-CHANNEL ATTACKS

Our proposed framework, Edge-CaSCADe, targets attacks that evict secret data from the cache memory repeatedly to understand the activity of the victim. This methodology is adopted in attacks like Flush+Reload, Flush+Flush, Prime+Probe, Prime+Abort, or Evict+Probe. To understand the working of such attacks, the most prominent Flush+Reload (F+R) attack is discussed in this section.

Public-key cryptographic algorithms, e.g., RSA, AES, DSA, Diffie Hellman key exchange, and ECC-based algorithms used for encryption of plain text data to cipher text involve some computation on the secret keys. They either use modular exponentiation or elliptical curve scalar addition on the key bits or perform a series of data access manipulations. The applied modification depends on the value of the key currently being processed. The reverse mapping of the key bit extraction can be done if the attacker is given access to the trace of the computation done. Consider the *Encrypt* function of the Square and Multiply algorithm, adopted by RSA encryption, as shown in Algorithm 1 [55]. Here b is the base message and e is the encryption key. Each bit value of e will determine the computation to be applied to encrypt the message. We can see that steps 6 and 7 are executed only when the value of e is 1.

In the Flush+Reload attack, the attacker flushes these instructions 6 and 7 from the cache memory initially and then waits for a period of time (until the victim executes these instructions again). The attacker then again tries to access these instructions and monitors the time needed

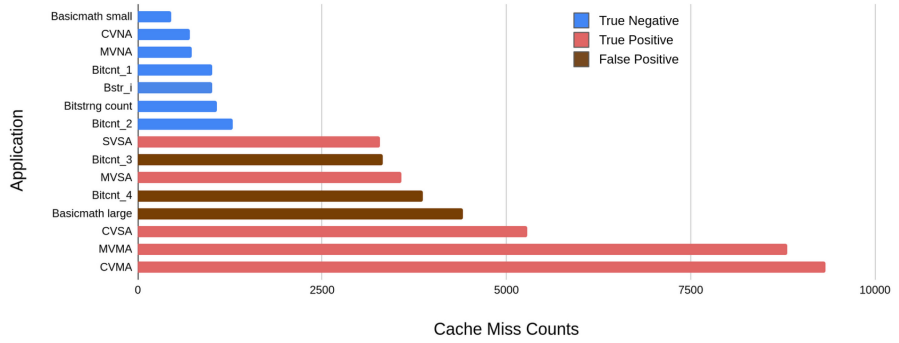


Fig. 1. Coarse-grained system-level cache miss counts.

ALGORITHM 1: Encrypt function of the Square and Multiply in RSA encryption: Encrypt (b,e,m)

```
1: a ←1
2: for i ← |e| to 0 do
3:     a ← a^2
4:     a ← a mod m
5:     if (e_i = 1) then
6:         a ← ab
7:         a ← a mod b
8:     end if
9: end for
10: return a
```

to fetch them. If the access is fast, it denotes a cache hit. This shows that the victim had accessed these instructions and hence led to its presence in cache memory, revealing the key value to be 1. However, if the attacker experiences slow access, it denotes a cache miss, eventually revealing the key value to be 0. In Flush+Reload and Flush+Flush attacks, the attacker makes use of the architecture-specific Flush instruction to flush the secret-dependent instructions from cache. However, for other attack types, the attacker makes use of other instructions that would in turn evict the required sensitive victim instruction from the cache memory.

For these attacks to be successful and reveal 100% key bits, the attacker has to flush the secret data from the cache periodically to get fresh information of a hit or miss, for every key bit. This increases the number of cache misses experienced by the victim only for this particular set of instructions. In our work, we are monitoring cache misses for only such sets of instructions to get a fine-grained detection of the attack.

4 PROPOSED FRAMEWORK OF FINE-GRAINED HARDWARE-BASED ATTACK DETECTION: EDGE-CASCADE

We propose a fine-grained hardware-based attack detection block that counts cache misses of only secret sections of the victim applications. Existing processor HPCs perform the event counting of the overall system. Our proposed detection module includes counters that count cache misses of only mapped secret addresses, making it a fine-grained monitoring approach. Figure 1 shows the system-level cache miss counts while running applications from the MiBench benchmark suit [22] and our test cases (details in Table 4). These counts were fetched on Intel x86 processor architecture using the Intel Vtune profiler tool [17]. The true negatives are the non-attack cases, where the HPC counts are shown to be low and hence there is no attack detection. True positives are the

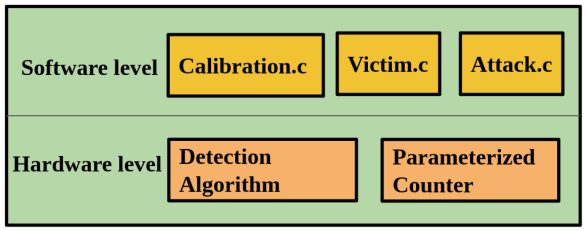


Fig. 2. Overview of the Edge-CaSCADe framework.

attack cases where the HPC counts are high, leading to attack detection accurately. However, false positives are the non-attack cases, whose HPC counts turn out to be high, due to the in-built execution pattern of the programs. Hence, these cases are wrongly classified as attack cases.

Finally, Figure 1 shows that when the counts are at the system level, some applications may mimic attack-like behavior in the microarchitecture, causing false alarms.

This motivates us to focus on fine-grained monitoring that only focuses on event counts of target sections in code, instead of counts at the system level.

Existing works use processor HPCs that count different types of microarchitectural events. By using performance monitoring tools like Intel VTune Profiler, these counts can be filtered to the finest granularity, i.e., at the level of a process or a section of the code. On the other hand, we propose fine-grained counters that target only the sensitive secret section of the given code/process. It needs no further process monitoring software and completely does the fine-grained counting in hardware, helping with accurate attack detection in case of anomalies detected in the targeted section of the code. This section discusses the details of the proposed framework in detail.

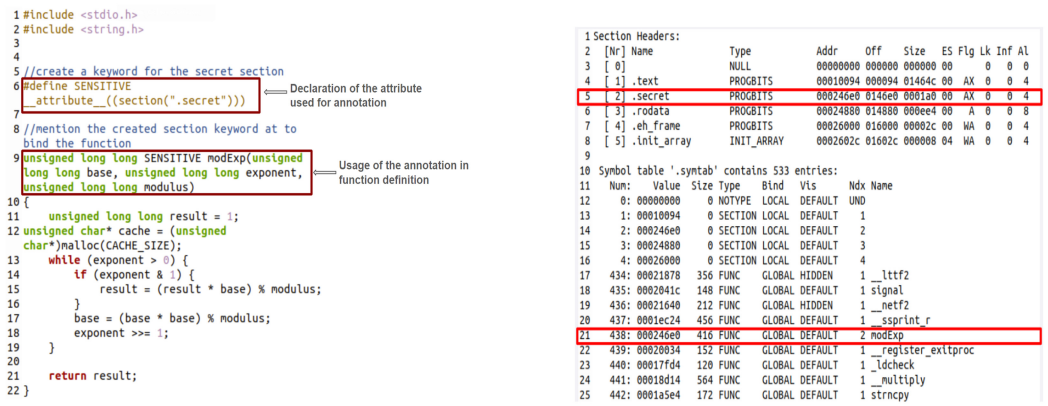
Figure 2 presents the overall framework of our Edge-CaSCADe system. A minimum of three components are involved at the software level. The victim code is the one that carries sensitive information and needs protection from attack. The victim will have to do some updates on his or her code before launching it in the multi-user environment, as discussed in Section 4.1.1.

Attack code is the code that launches CSCAs on the victim to retrieve the secrets. Calibration code is used to derive certain parameters from the victim code in order to run the detection process and is further discussed in Section 4.1.2. At the hardware level, we have implemented our detection module to perform the monitoring and detection task, using counters. In other words, our proposed detection methodology involves certain software-level activities and certain hardware-level phases, as discussed in the following sections.

4.1 Software-level Activities

The victim has to perform certain software-level activities, one time, on the application source code that the victim wants to protect.

4.1.1 Marking the Secret Code. Initially, the victim has to mark the sensitive section of the code that needs to be protected before the code is compiled. The sensitive section is that part of the code that is dependent on the secret key of the victim. This is a one-time task done by the victim offline in the encryption algorithm. For example, steps 6 and 7 from the RSA algorithm snippet shown in Algorithm 1 are sensitive to attack, as these instructions have the potential to reveal the key bit value. Similarly, in the case of AES encryption, the tables involved in the key-dependent table lookup are sensitive sections to be protected. Since our detection method involves a fine-grained monitoring approach, the proposed counter will count the cache misses of only this marked secret section.



(a) Marking the sensitive function in the code

(b) ELF file showing the section binding

Fig. 3. Program annotation for marking a single secret section.

This marking can be done by binding a chosen section to a separate thread or function or annotating the section, as explained further. The victim should also add a *syscall* at the beginning of his or her code to begin the monitoring process when the victim execution starts and at the end of the code to clear the detection module registers. The first *syscall* initializes the respective registers with the values found from calibration and begins the detection process.

A victim may use either one type of encryption in his or her application or multiple encryption algorithms. The encryption algorithms may use one secret key during encryption, like in AES, RSA, and so forth, or use multiple secrets for encryption, like in PQC encryption algorithms. In either of the cases, the instructions in the algorithm whose execution depends on the value of the key should be marked as secret sections. It could be single or multiple, as explained further. The victim should mark the respective sensitive sections as discussed below.

(1) Marking single section: From Figure 3(a), we can see that the `modExp` function, which does the modular exponentiation in the RSA cryptographic algorithm, for example, is marked as `SENSITIVE` and bound to the `.secret` section. The address to this section will be traced in the generated ELF file (as shown in Figure 3(b)) during the calibration phase for further monitoring purposes.
(2) Marking multiple sections: When the victim uses multiple encryption algorithms, all the sensitive sections should be marked so that they are bound to a single secret section in memory. We can see from Figure 4(a) that two different tables (e.g., in AES encryption) are marked as sensitive. The generated ELF file in Figure 4(b) shows that the two tables are bound to the secret section (considering the size and address of the secret section and the tables). During calibration, all the individual addresses bound to this secret section will be fetched and multiple counters and registers will be used for monitoring and detection purposes.

4.1.2 Calibration of Execution Environment. Calibration is the phase where the victim code is executed in a real environment with encryption keys as deemed appropriate to get the parameters of detection. This is carried out offline. The algorithm for the calibration phase is shown in Algorithm 2. The related code has a size of less than 5 kB. To begin with the calibration, we need the victim code and also assume a fixed *sample_time* to decide the refresh rate of the counter used for detection. The counter will be incremented for every cache miss corresponding to the mapped secret section within a given *sample_time*. After the *sample_time* period, the counter is reset to

```
// Number of rounds for AES-128
#define AES_NUM_ROUNDS 10
#define SENSITIVE
__attribute__((section(".secret")))
// AES State (4x4 matrix)
typedef uint8_t state_t[4][4];
static const uint32_t SENSITIVE td[256];
static const uint32_t SENSITIVE td2[256];
```

Declaration of the attribute used for annotation

Usage of the annotation in variable declaration

(a) Marking multiple secret sections in the code

```
[ 3] .secret           PROGBITS        000251c4 0151c4 000800
  00   A  0   0  4
[ 4] .eh_frame         PROGBITS        000269c4 0159c4 000004
  00  WA  0   0  4

Symbol table '.symtab' contains 527 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
    38: 000269cc     0 OBJECT  LOCAL  DEFAULT    5 frame_dummy_
    39: 00000000     0 FILE    LOCAL  DEFAULT  ABS new_test_aes.c
    40: 000251c4  1024 OBJECT  LOCAL  DEFAULT    3 td
    41: 000255c4  1024 OBJECT  LOCAL  DEFAULT    3 td2
```

(b) ELF file showing the section binding

Fig. 4. Program annotation for marking multiple secret sections.

ALGORITHM 2: Calibration

Require: Victim Code, *sample_time*

1: Trace the address of the marked section from the generated elf file
2: *secret_addrs* ← address found in step 2
3: Measure time for each access of the secret section
4: *ISAT* ← Maximum time difference between each subsequent access of secret, as measured in step 3
5: *secret_access_count* ← Average No. of secret instruction cache accesses per *sample_time* (considering cache misses)
6: **return** *secret_addrs*, *ISAT*, *secret_access_count*

zero. In our experiments, we have set the *sample_time* to one-tenth ($\frac{1}{10}$) of the total victim encryption code execution time. The analysis behind this selection is discussed in Section 5.1. The data generated in the calibration phase will be used in the monitoring phase. The generated data comprises:

(1) *Address of the secret section to be monitored:* Since the victim has already annotated the secret part of the code, as seen in Section 4.1.1, the address corresponding to this secret part is found by tracing the generated ELF file, as shown in Figures 3 and 4. The corresponding physical address will be entered in the address register by *syscall* done by the victim using the address translation mechanism.
(2) ***Inter-Secret Access Time (ISAT)*** *of the secret section:* This denotes the maximum time between subsequent accesses of the secret section. It depends on the victim algorithm. It is used to note the pattern of execution of the secret section by the victim. The attacker will try and flush the secret from the cache memory following this victim pattern only, due to which the inter-cache miss time (caused by the attack) will also be approximately similar to the ISAT. Hence, noting this time would help us in identifying the suspicious cache misses from the genuine ones, further reducing the false alarms. To assume some lag or any other delays, we have increased this time by a margin of 25% in our detection technique.
(3) *Secret access count:* This is the number of secret section cache accesses for a sample period of time. During the attack, the attacker will flush the secret from the cache forcefully, causing cache misses for every fetch of secret instruction. We calibrate the number of secret cache accesses that would be done within a sample time, considering all cache misses. The *threshold* value used in the monitoring phase, in order to detect the attack, is set to 50% of the *secret access count*. The analysis behind this selection is further discussed in Section 5.1

4.2 Hardware-level Architectural Changes

Figure 5 shows the integration of Edge-CaSCADe with the processor core (here is the Comet RISC-V five-stage core—more details are given in Section 5, but the added hardware could be integrated in any core design). The detection module includes the following major components:

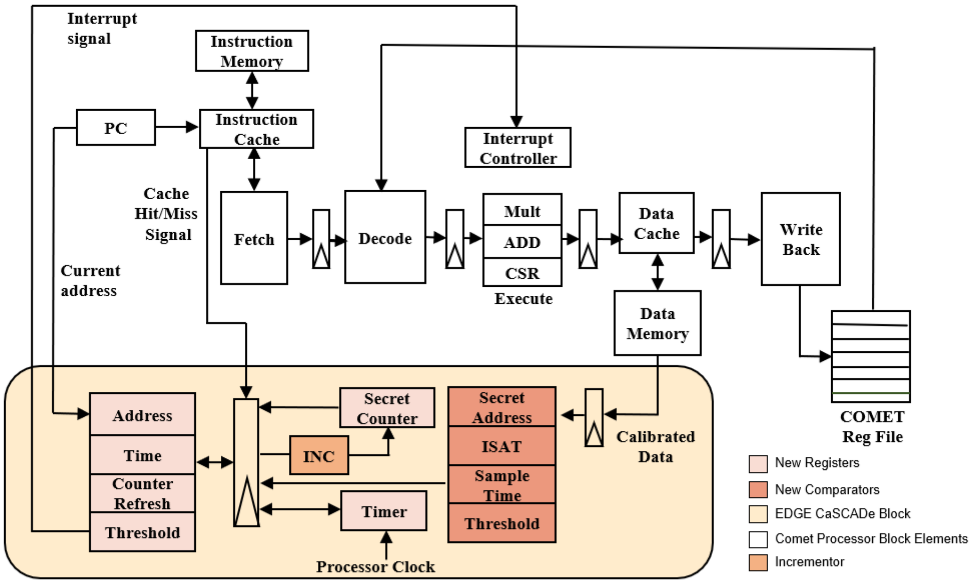


Fig. 5. Edge-CaSCADe integrated with the COMET RISC-V processor core, as an implementation example.

- *Secret Counter* to count cache misses of the marked secret section. The counter wraps around on reaching the maximum limit; however, our sample time and threshold values are selected in such a way that the counter is refreshed before reaching the maximum limit.
- *A register set* to store the calibrated parameters used for detection, such as secret addresses, ISAT, sample time, and threshold.
- *A set of comparators* to compare the calibrated parameters with the runtime counts. The address comparator compares the current address causing a cache miss with the secret address to be monitored. The time comparator checks for the time difference between subsequent secret address accesses and matches with the calibrated ISAT value. The counter refresh comparator compares the current time with the sample time to understand if the counter needs to be refreshed. And the threshold comparator compares the secret counter value with the calibrated threshold value for detection of the attack.
- *A timer* to keep track of the counter refresh time.
- *An increment unit* to increment the counter in the event of a secret section cache miss that could be suspicious.

Our proposed detection module performs all three tasks of monitoring, detection, and mitigation at the hardware level.

4.2.1 Monitoring. In this phase, the cache misses caused by the secret address are counted at runtime. During an attack, the attacker flushes the sensitive section of the victim code, thereby causing the microarchitecture changes captured by our monitoring module. Monitoring follows Algorithm 3. From Figure 5, we can see that the cache miss signal and the address from the program counter are fed to the comparator block at runtime. In the event of a cache miss, the address from the PC is compared with the secret address deduced from calibration. Once matched, the ISAT is also compared, which, when matched, causes an increment in the secret counter. This value of the secret counter is then compared with the predetermined threshold to detect the attack. After every

ALGORITHM 3: Algorithm to count suspicious cache misses

Require: *secret_counter*, *ISAT*, *sample_time*, *threshold*, *secret_addrs*

```
1:  In the event of a cache miss
2:  if current_sample_time > sample_time then
3:      secret_counter ← 0
4:  end if
5:  if current_address == secret_addrs then
6:      if current_inter_secret_access_time ≤ 1.25 ∗ ISAT then
7:          increment secret_counter
8:          if secret_counter ≥ threshold then
9:              "Raise the Attack Detection Signal"
10:         end if
11:     end if
12: end if
```

cache miss, the module timer is compared with the sample time to determine if the counter needs to be refreshed. This is done to ensure all the comparisons update the counter at a granularity of sample time.

4.2.2 Detection. From Algorithm 3, line 8, we see that when the *secret_counter* value exceeds the predetermined threshold value, the attack detected signal is raised. This signal is fed to the processor as an Interrupt. Following that, the processor performs an **Interrupt Service Routine (ISR)**, which determines which process was attacked, and then takes appropriate action. This is a highly fine-grained and lightweight attack detection and hence highly accurate. Its fine granularity and light weight come from the following facts:

(1) The cache misses related to only the secret sections are considered as against the overall cache misses of the system.
(2) The inter-secret access time ensures that the misses have occurred due to suspicious monitoring activity of the attacker and not any other genuine application running in the background.
(3) The counting and comparisons happen at the hardware level and no repeated *syscalls* are involved in fetching the counts. This ensures very minimal performance overhead, making it lightweight and applicable on edge devices.

Along with high accuracy, the detection is very quick. This is because the threshold is decided for a preset sample time, within which, if the threshold is crossed, the attack is detected. In this proposed algorithm, we only introduce a single system call at the beginning of victim execution, unlike the system calls involved in the tools used in past work (refer to Table 2) to fetch the microarchitectural counts continuously. Since this detection algorithm is synthesized at the microarchitecture level of the processor core, it has very minimal performance overhead, making it suitable even for edge deployment.

4.2.3 Mitigation. Once the attack is detected, the processor performs an ISR, which determines which process was attacked, and then takes appropriate action. Currently, the action that we take is to stop the victim's execution. This can also be done through a control application on notification to the victim. The focus of our technique is on successful and timely detection of the attack. Any mitigation similar to what is done in the current literature, such as the ones mentioned in [26, 27, 56], could be applied. We have summarized the steps involved in the detection mechanism along with activities involved in each step in Table 3.

Table 3. Proposed Detection Method Steps and Related Activities

Steps	Activities in the Step
Marking the secret	Victim marks the secret sections of his or her code
Calibration	Find the marked address Derive *sample_time* Derive *ISAT* Derive *threshold*
Monitoring	Count cache misses of the mapped address Refresh the counter after every *sample_time*
Detection	If counter value exceeds the threshold, attack is detected
Mitigation	On attack detection, take appropriate action as per ISR

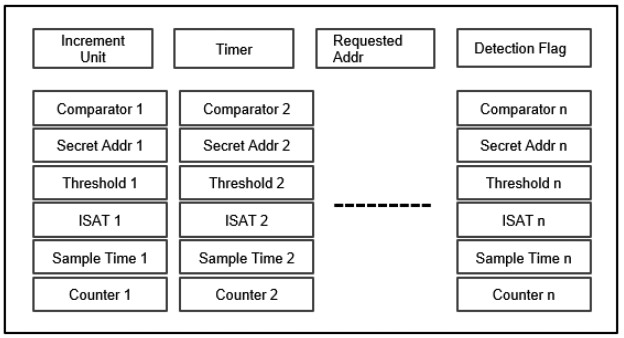


Fig. 6. Components needed in case of multiple victim monitoring.

4.3 Adopting the Detection Module on Single- and Multiple-issue Processors

Figure 6 shows the components needed in case of multiple victim monitoring. The parameters fetched during calibration will be unique for each secret section to be monitored and hence the respective components will replicate with the increased number of secret sections. The common components like increment unit, timer, requested address, and detection flag could be shared among the different monitored sections.

In *Single-issue Processors*, at the most one instruction is executed at a time. However, multiple victims can be executed in a multi-processing system with one victim instruction under execution at a time. When there is a context switch from, say, victim A to victim B, the victim B counter registers will be updated upon its secret address matching with the comparator. Hence, only those registers related to victim B's secret address will be involved in the detection process. The calibrated parameters of all the victims will be placed in components shown in Figure 6. Thus, the number of register sets in the architecture determines the number of victims or identified code sections that can be monitored simultaneously.

On account of a cache miss, the address comparators in our detection module will check for the secret address that caused the cache miss and accordingly update only that respective counter.

The victim may have multiple secret sections to be protected. These sections have to be marked individually so that their respective parameters identified after calibration are stored in respective registers, as shown in Figure 6. In the monitoring and detection phases, only those corresponding registers that are linked with the secret section address that caused the current cache miss will be involved.

However, in *Multiple-issue Processors*, more than one instruction can be executed simultaneously. Hence, at a time, multiple comparators may match with the secret addresses simultaneously, causing updates in multiple counters. When any of the counter values exceed the threshold, the detection interrupt signal would be raised.

The accurate working of our model depends on the calibrated parameters, which in turn depend on the workload of the system during calibration. Hence, we advise running the calibration every time the victim changes the host architecture or experiences a high workload on the system. By architecture, we assume that the processor is embedded with Edge-CaSCADe. We have discussed the detection accuracy on high- and low-load conditions during calibration in Section 5.2 in detail.

5 EXPERIMENTAL SETUP AND RESULTS

We demonstrate Edge-CaSCaDE on the Comet RISC-V processor simulator. Comet is an open-source 32-bit five-stage pipeline processor written in C++ for **High-Level Synthesis (HLS)** that can operate at a frequency of 700 MHz when synthesized [57]. The simulator is fast, cycle-accurate, and bit-accurate, as it mirrors the hardware microarchitecture of the processor. Comet is an in-order processor. It has a four-way associative cache that uses the Least Frequently Used replacement policy. In our experiments, the system-level counts are found by the in-built performance counters (HPCs) of the Comet processor, whereas the fine-grained secret section counts are found by the proposed detection module. The OS that we use for the experiments is ZephyrOS [58]. It is an open-source OS dedicated to embedded systems. C++ language is used to describe the architecture of our proposed cache monitor block, which is further transformed into the **Register-Transfer Level (RTL)** description using the Catapult HLS Toolchain. This RTL description is further fed to the Synopsys Design Compiler to transform it into a Gate-Level Netlist. We have generated area, power, and timing overhead reports from this implementation using a 28 nm FDSOI technology.

We consider RSA, AES, and ECIES encryptions to demonstrate our framework. The maximum key size in RSA, AES, and ECIES is 2,048, 256, and 256, respectively [59, 60]. As per research in [61], recovering cryptographic keys from partial information needs at least 25% of keys known at a stretch. Hence, in our experiments, we have kept the counter refresh rate of 10% of the victim encryption code execution time, which covers this possibility. For a key size of 2,048 bits, 10% execution would at the most complete processing of 204 bits. Further, the threshold value 50%, when reached, detects the attack, thus stopping the counter. Hence, synthesis of an 8-bit counter would suffice in this case. As presented in Table 4, we have constructed various test cases to scrutinize the behavior of the detection module on multithreaded programs in different scenarios. We mention the name of the test case scenario along with the number and type of threads involved. The details of each thread are mentioned in Table 5. When several options are reported for a thread, they are randomly selected among the number of experiments.

The attack is launched on the RSA following the methodology described in [62], on the AES following the technique mentioned in [63] and [64], and on the ECC-based cryptographic algorithm ECIES using [65].

We have broadly divided the use case scenarios into three categories, namely:

(1) Single victim
(2) Single complex victim
(3) Multiple complex victims

The single victim scenarios include only the victim code without the MiBench benchmark application, with only one secret-dependent encryption section to be monitored. The complex victim scenarios include benchmark and complex user applications along with the victim encryption applications. We have performed these complex victim test experiments in single as well as multiple

Table 4. Test Cases

Name	No. of Threads	Details of Each Thread
Single Victim (SV)	2	T1 -> Main code, T2 -> Victim Encryption code
Single Victim Single Attacker (SVSA)	3	T1 -> Main code, T2 -> Victim Encryption code, T3 -> Attacker code
Complex Victim No Attacker (CVNA)	3	T1 -> Main code, T2 -> Victim Encryption code, T3 -> Benchmark Code
Complex Victims Single Attacker (CVSA)	4	T1 -> Main code, T2 -> Victim Encryption code, T3 -> Benchmark Code, T4 -> Attacker code
Complex Victim Multiple Attacker (CVMA)	5	T1 -> Main code, T2 -> Victim Encryption code, T3 -> Benchmark Code, T4 -> Attacker1 code, T5 -> Attacker2 code
Multiple Victims No Attacker (MVNA)	5	T1 -> Main code, T2 -> Victim Encryption code1, T3 -> Victim Encryption code2, T4 -> Victim Encryption code3, T5 -> Benchmark Code
Multiple Victims Single Attacker (MVSA)	6	T1 -> Main code, T2 -> Victim Encryption code1, T3 -> Victim Encryption code2, T4 -> Victim Encryption code3, T5 -> Benchmark Code, T6 -> Attacker code
Multiple Victims Multiple Attacker (MVMA)	7	T1 -> Main code, T2 -> Victim Encryption code1, T3 -> Victim Encryption code2, T4 -> Victim Encryption code3, T5 -> Benchmark Code, T6 -> Attacker code1, T7 -> Attacker code2

Table 5. Details of Threads Involved in Use Cases

Type of Thread	Details
Main code	The code that invokes encryption on base messages
Victim encryption code	RSA encryption codes with key lengths 512 and 1,024, AES encryption codes with key lengths 128 and 192, and ECIES with key length 256
Attacker code	Code that repeatedly performs flushing of secret sensitive sections with variations to retrieve 25% to 100% key bits
Benchmark code	MiBench benchmark (tested on RSA, AES, ECIES) codes that include Test 1:rad2deg, Test 2:basicmath small, Test 3:basicmath large, Test 4:bitcnt_1, Test 5:bitcnt_2, Test 6:bitcnt_3, Test 7:bitcnt_4. Embench-IoT benchmark codes (tested on ECIES) like md5su (along with Tests 1 to 3) and prime-count (along with Tests 4 to 5), and user-defined complex codes having recursive function calls and loops, large data accesses, and syscalls

victim cases. In the single complex victim case, there is one secret section to be monitored, whereas the multiple complex victim case has multiple victim secret sections to be monitored.

Tables 4 and 5 provide more details on the various test cases. The codes for test cases used can be found at https://github.com/Pavitra07/Use-Cases.

5.1 Calibration Parameters

In our approach, calibration is a very important step that determines the parameters used during the detection, like the threshold, secret address, *ISAT*, or sample time. We have selected

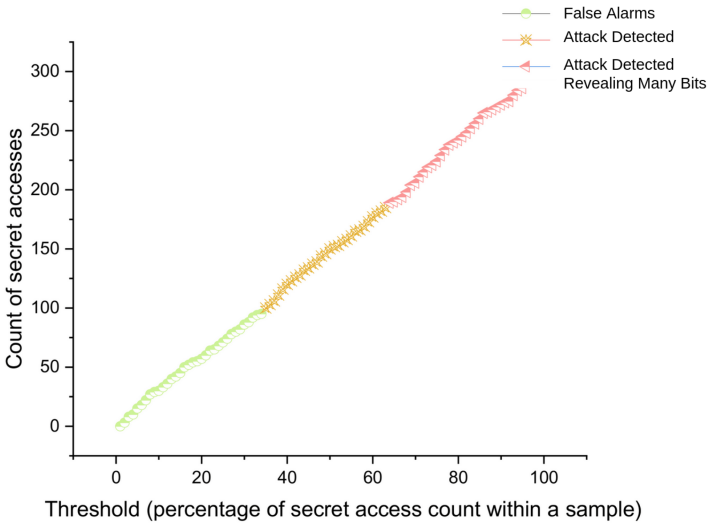


Fig. 7. Example of a threshold analysis.

sample time to be 10% of victim encryption code execution time and threshold to be 50% of the number of secret access counts within a sample time. This selection is based on the following discussion.

5.1.1 Determining the Threshold. The *Threshold* is defined as the maximum number of cache misses counted by the counter before raising the detection flag. We conducted experiments on our use cases by varying the threshold values from 1% to 100% of a number of secret accesses within a sample time (assumed 10% of victim encryption code execution time).

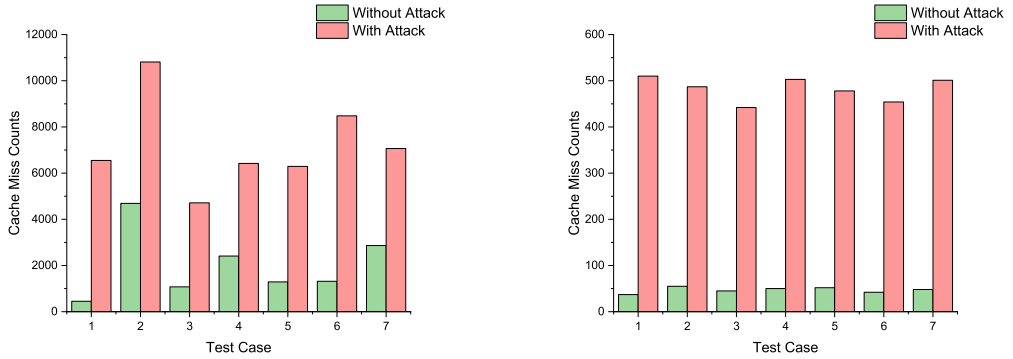
Figure 7 represents the count of secret accesses in a sample time corresponding to the threshold value when set to different percentages of total secret counts within a sample for the 512-bit RSA encryption CSVA case. From the figure, we can see that when the threshold value is set to $< 30\%$, false alarms are raised.

For values from 30% to 60%, attacks are detected successfully. However, when the threshold is $> 60\%$, the attack is detected successfully without false alarms. Such latter cases reveal more bits to the attacker in case of borderline attacks, before being detected. By borderline attack, we mean if an attacker tries to fetch partial keys just before and after the counter is refreshed.

5.1.2 Determining Sample Time. The *Sample Time* is the time period after which the counter gets refreshed. We conducted experiments on our use cases by varying the sample time from 5% to 25% of the total victim encryption code execution time. Table 6 shows the effect of variation in sample time on the number of key bits involved in each time slot, and the corresponding risk of bits revealed, for the same 512-bit RSA encryption CSVA case considered for threshold variation. We have also examined a situation in which the attacker initiates an attack at a moment of sample time frame that results in a counter value lower than the threshold, even though the attacker has retrieved every key bit attacked in that sample, consequently evading detection. We have named this condition as the most challenging or worst-case scenario. From Table 6 we see that when the sample time is set to 5% of victim encryption code execution time, the counter is refreshed too frequently. A sample time of 10% and 15% can be considered for attack detection. In the case of 20% and 25%, more bits are revealed to the attacker, having the risk of revealing the full key using partial key bits [61].

Table 6. Sample Time Analysis

Percentage of Victim Encryption Code Execution Time (A)	Number of Secret Accesses within a Sample Time (B)	Threshold (50% of B) (C)	Bits Revealed until Detection (D)	Bits Revealed in Worst Case (E)	Remarks (F)
5%	28	14	2.7%	8.2%	High counter refresh rate
10%	51	26	5.07%	15.03%	Can be considered
15%	77	39	7.61%	22.65%	Can be considered
20%	102	51	9.96%	29.88%	Risk of revealing more bits
25%	126	63	12.3%	36.9%	Risk of revealing more bits



(a) Total cache misses

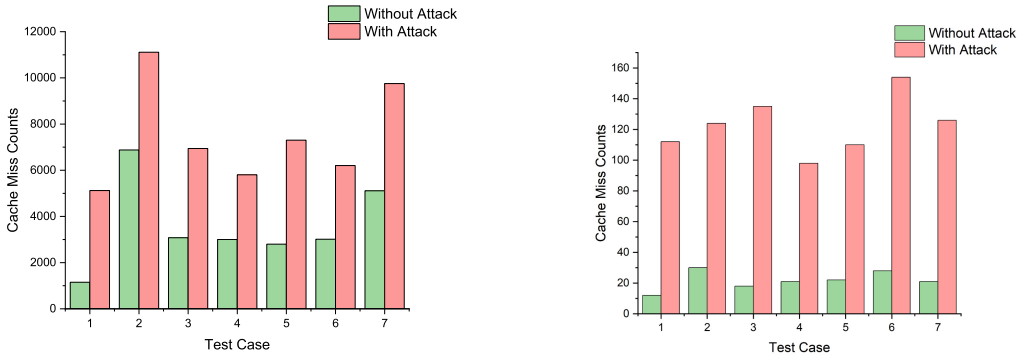
(b) Secret section cache misses

Fig. 8. Cache miss counts for RSA encryption.

5.2 Test Results

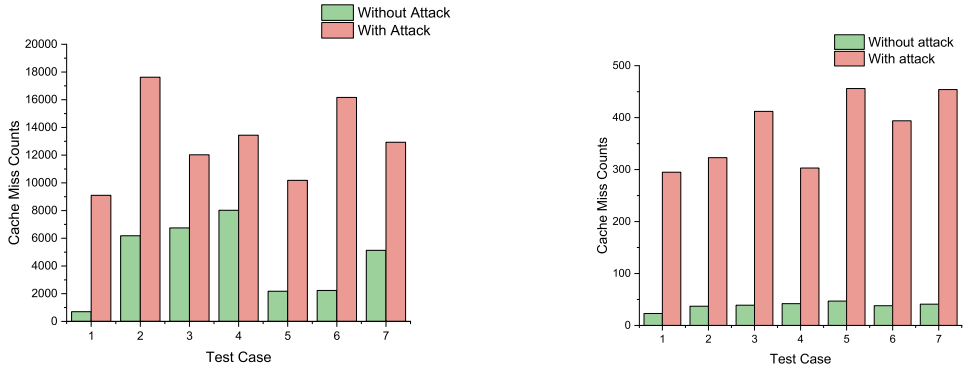
Figures 8, 9, and 10 show the cache miss counts for victim applications with RSA (512), AES (128), and ECIES (256) encryption algorithms, respectively. We have plotted the cache miss counts for seven different test cases that have a victim application along with MiBench and Embench benchmark codes.

The seven test cases are test cases taken from the Mibench benchmark suit for the RSA and AES cases. For the ECIES case, two of the Embench use cases were also tested along with the Mibench test cases (refer to Table 5). We can see that counts in the absence of an attack, in some test cases, are comparable to those in the case of an attack. For example, in Figure 8(a), counts for test 2 in the no-attack scenario are comparable to counts for test 3 in the attack scenario. Similarly, in Figure 9(a), the counts for tests 2 and 7 with no attack are comparable to counts for tests 3, 4, 5, and 6 with attack. Also, in Figure 10(a), counts for tests 2, 3, and 4 with no attack are comparable to counts of tests 5 with attack. Hence, when the counting granularity is coarse, accurate classification becomes difficult with chances of raising false alarms. Figures 8(b), 9(b), and 10(b) show cache miss counts only related to the secret section marked by the victim for the same use cases. We can see that, in all three plots, the counts go extremely high in case of attack, as compared to the no-attack scenario. This shows that the fine-grained monitoring approach proves to be more efficient in terms of attack detection, reducing the false alarms.



(a) Total cache misses (b) Secret section cache misses

Fig. 9. Cache miss counts for AES encryption.



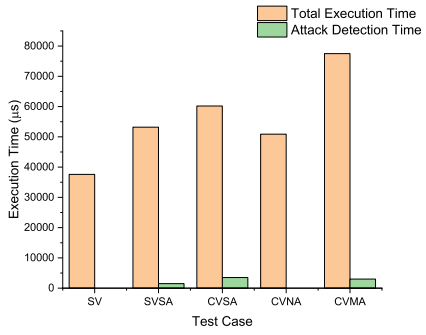
(a) Total cache misses (b) Secret section cache misses

Fig. 10. Cache miss counts for ECIES encryption.

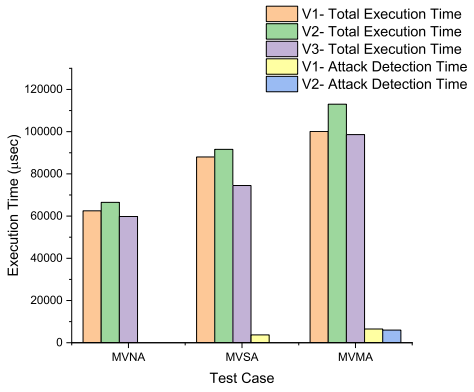
Figures 11, 12, and 13 report the execution time of the victim applications along with the time at which the attack is detected. This only includes the execution time during the encryption run, and not the calibration phase, as the calibration is done by the victim offline. In Figures 11(a), 12(a), and 13(a), we can see the total execution time needed for a single victim code to complete execution and the time at which the attack is detected. Similarly, in Figures 11(b), 12(b), and 13(b), we can see the execution time of multiple victim codes and the time for detecting the attacks on them (one and two victims under attack in the MVSA and MVMA test cases, respectively). We can see that the detection speed is approximately within 2% to 5% of the start of the attack. From Figures 11, 12, and 13, we can see that there is no attack detection in the no-attack case (SV, CVNA, MVNA).

In our test scenarios, we performed the calibration on the victim with and without the benchmark codes to cover the possibilities of low- and high-load conditions during calibration.

Detection performance is reported as true positive, false positive, true negative, and false negative. Figures 14(a) and 14(b) show the performance of Edge-CaSCADe with calibration done on low- and high-load conditions, respectively, for all use cases. When the calibration is done in low load, i.e., only the victim code running without benchmarks or other complex codes (Figure 14(a)), we observe some false negatives reducing the number of true positives. When the calibration is done

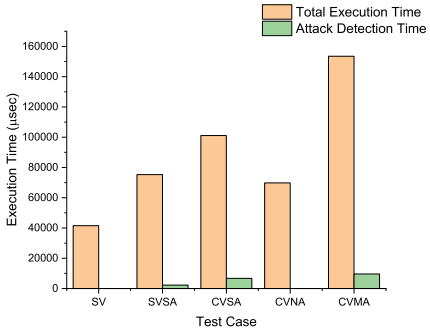


(a) Total victim execution time (μs) versus time to detect the attack for single secret cases

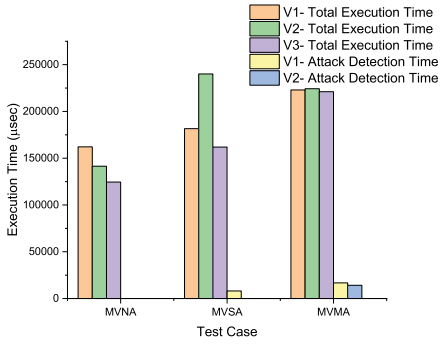


(b) Total victim execution time (μs) versus time to detect the attack for multiple secret cases

Fig. 11. Victim cases with RSA encryption.

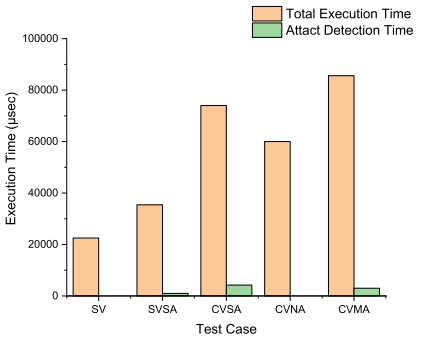


(a) Total victim execution time (μs) versus time to detect the attack for single secret cases

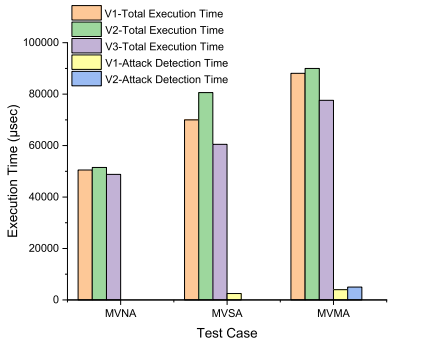


(b) Total victim execution time (μs) versus time to detect the attack for multiple secret cases

Fig. 12. Victim cases with AES encryption.

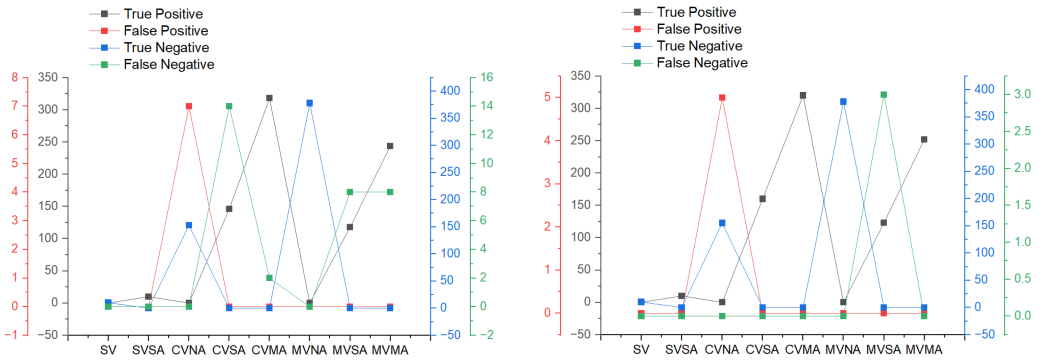


(a) Total victim execution time (μs) versus time to detect the attack for single secret cases



(b) Total victim execution time (μs) versus time to detect the attack for multiple secret cases

Fig. 13. Victim cases with ECIES encryption.



(a) Calibration on low load (b) Calibration on high load

Fig. 14. Detection results on all use cases.

Table 7. Consolidated Confusion Matrix for All Test Cases

		Predicted	
		Positive	Negative
Actual	Positive	98%	2%
	Negative	1.08%	98.9%

in high load (Figure 14(b)), fewer false negatives are observed in comparison. However, the rest of the performance metrics show comparable results in both cases. Hence, the victim system load during calibration plays an important role in deciding the parameters for detection. We suggest that the calibration should be done on a real system load, comparable to a multi-user environment, to get accurate parameters applicable for attack detection.

Table 7 shows the detection outcome for all the test cases considered in our experiments. We see that all the attack cases considered are accurately detected in our detection model, with only 1.9% false-positive outcomes on average. The other detection metrics, namely *Precision*, *Recall*, and *F1 Score*, are as follows:

$$Precision = TP/(TP + FP) = 99.3\% \tag{1}$$

$$Recall = TP/(TP + FN) = 97.885\% \tag{2}$$

$$F1Score = 2 \times Precision \times Recall/(Precision + Recall) = 98.58\%. \tag{3}$$

5.3 Discussion on Hardware Overhead

In Figure 15 we show the microarchitectural components of Edge-CaSCADe added to the existing Comet microarchitecture. The calibrated information of the victim is stored in respective registers. The current address under execution that caused the cache miss is stored in the Requested Address register for the purpose of comparison with the secret address. The secret counter does the counting of cache misses for the dedicated secret section. The comparators perform the runtime comparison of the calibrated parameters with the current status of execution as discussed in Section 4.2. Figure 15 shows the components needed for attack detection on a single victim secret section. However, our proposed solution can be used to simultaneously detect multiple attacks on the victim by using multiple counters. Figure 6 shows the additional components added for multiple attack detection. Each counter monitors a separate secret section. The victim only needs

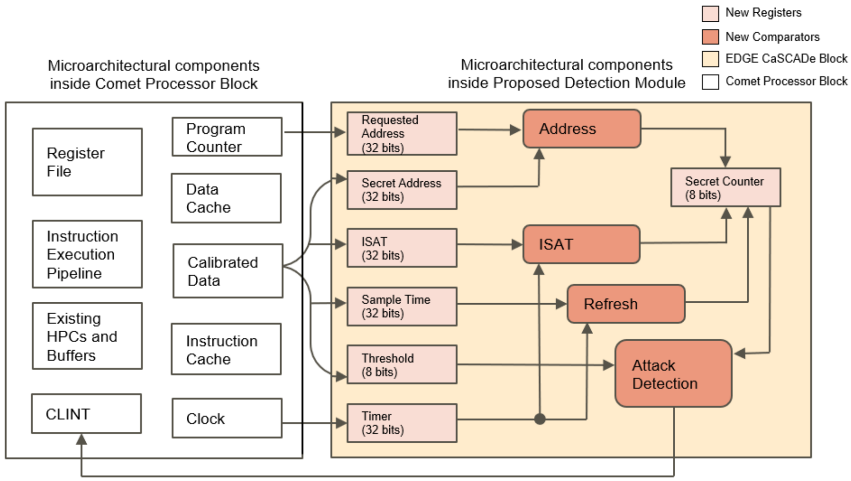


Fig. 15. Microarchitectural details of the Edge-CaSCADe.

Table 8. Area and Power Overhead with Path Delay after the Addition of the Hardware Module for Multiple Counters

Number of Counters	**Area of Proposed Detection Module (μm^2)**	**Total Area Overhead (%)**	**Power Consumption of Proposed Detection Module (Vectorless Estimation) (μW)**	**Total Power Overhead (%)**	**Critical Path Delay (ns)**
1	358.06	0.90	29.0	1.09	0.6149
2	462.72	1.17	36.2	1.36	0.6164
3	569.73	1.44	43.3	1.62	0.6173
4	689.52	1.74	51.4	1.88	0.6204
5	789.0	1.99	58.3	2.14	0.6207

to run the calibration code and derive the parameters related to each secret. Edge-CaSCADe will then simultaneously check for the count of misses for each section individually. On account of an attack, the victim is notified.

We have implemented Edge-CaSCADe with a number of counters from one to five, which could monitor one to five secret sections, respectively. Table 8 shows the area, power, and critical path delay of the detection module for one to five counters. The table also provides the area and power overheads over the processor core, without including the cache memory. Including the cache memory would reduce these overheads even further. The area and frequency results are synthesized using a 28 nm FDSOI technology node. The area of Comet RISC-V core used in experiments as the reference is $39,549 \mu m^2$, and its power consumption is $2.67 mW$.

To the best of our knowledge, this work is the only CSCA detection method that performs the detection completely in the hardware at runtime. Other approaches [1, 13] use HPCs to monitor events, but the detection methods fetch these HPC counts and perform analysis in software to detect the attacks.

As expected, the critical path delay is significantly lower than the one of the processor core. Furthermore, there is no impact on the maximum operating frequency or overhead as the monitoring hardware is not in the critical path of the processor.

As part of our performance analysis, we have examined the module's effectiveness by focusing on the following key features:

(1) Area Overhead: As shown in Table 8, the area of a single-counter-based module is 358.06 μm^2, whereas that of the Comet processor core is 39,549 μm^2, leading to a 0.9% overhead. This overhead is very minimal when the overall area of the core and the cache memory and other peripherals are considered.
(2) Power Overhead: The power consumption (vectorless estimation) for a single-counter-based module is 29 μW, whereas the Comet core power is 2.67 mW, leading to a 1.09% overhead. This overhead is further reduced when the power dissipation of other components like cache memory and other peripherals is included.
(3) Timing Analysis: The critical path delay of our module is significantly lower than that of the processor core running at 700 MHz.
Furthermore, there is no impact on the maximum operating frequency since the proposed monitoring hardware module is not in the critical path of the processor.
(4) Scalability Analysis: We have increased the number of counters from one to five. As shown in Table 8, the area and power increase with the number of counters, which is expected, whereas the critical path delay in the proposed module is only slightly affected.
(5) Detection Speed: The attack is detected within 2% of the execution time, once the attack is started in all the cases considered in the experiments. This execution time is the total time of victim execution if the attack was not detected.
(6) Detection Accuracy: We have achieved a precision of 99.3%, a recall of 97.885%, and an F1 Score of 98.58%.

Since our module increments the counter on encountering a secret section miss, this event leads to additional power consumption. Now since this additional power consumption occurs only during the secret section execution, this may appear to open another side channel for power analysis. However, this will not be useful information to launch another power analysis attack for the following reasons:

(1) The 2% power overhead that we mention is largely overestimated and is the combined overhead when five counters (secret sections) are considered, along with all the other components (e.g., secret address matches, sample time comparisons, ISAT comparisons). Also, this overhead considers only the core power utilization, and not other peripherals and system modules (e.g., cache, interconnects). Including them, the overhead would be very minimal to leak information for analysis, as it is actually just a counter increment that happens in the module.
(2) Second, even if we consider the slightest possibility of extracting this minimal power by launching attack, it is not going to reveal any other "new" information that has already been leaked by the timing attack taking place during the detection, which is eventually detected within 2% of the execution time from the start of the attack.
(3) Overall, as a mitigation against power side channels, several techniques, such as dummy counters, fake cycle insertion, random delays, or adding noise [66], could be adopted to avoid the slightest possibility of power leakage.

6 DISCUSSION ON ADDRESSING ISSUES WITH CURRENT DETECTION TECHNIQUES

Since our detection technique works completely in hardware and is not based on machine learning, most of the issues faced by current HPC-based detection methods, as mentioned in [52, 53] and [54], are addressed as follows:

(1) The HPCs adopted in previous work give the counts at a coarse-grained level, which may give rise to false alarms. When some genuine applications also behave similarly to the attack application at the microarchitectural level, e.g., causing frequent cache misses, chances of them being misclassified as attacking processes are high. Our detection module, on the other hand, uses custom fine-grained counters that count the events linked at the address level discussed in Section 4.2. Further, we apply a technique to identify and count only the cache misses that could be suspicious (Algorithm 3) to further reduce false alarms.
(2) All the existing methods adopt a profiling tool to fetch the HPC counts and then feed them to the application involving detection classifiers as shown in Table 2. This causes performance overhead due to the interaction with OSs via system calls and so forth. In our approach, the event counts are directly forwarded at the hardware level to the detection module. As discussed in Section 4.1.1, there is only one syscall involved, which initiates the detection module and initializes the respective registers. The next syscall would be to de-initialize the registers after victim execution. The counting and monitoring take place completely in hardware, without involving additional syscalls, incurring no additional performance overhead, delay, or lag, as described in Section 4.2.1.
(3) The victim does calibration on the host system to generate the parameters used for testing. When the environment changes, the victim should run the calibration again, ensuring that the parameters are specific to the real environment, and hence not biased to specific pre-collected data, as observed in machine learning classifiers. The calibration method and its dependence on detection accuracy are discussed in Section 5.1 and through Figures 14(a) and 14(b).
(4) We have conducted experiments with multiple victims, multiple users, simultaneous attacks, and benchmark application cases, covering most of the experiment domain. These use cases are described in Tables 4 and 5.
(5) During the *syscall* by the victim, the victim process *id* will be used to translate and fetch the physical address of the secret section to be monitored, as discussed in Section 4.1.2, hence addressing the issue of inconsistency during process switching.

7 CONCLUSION AND FUTURE WORK

We have presented a methodology to detect timing-based CSCAs in a fine-grained approach. We have created a detection module that interacts with the processor core to monitor cache miss counts for secret sections of the victim, and have also derived a technique to distinguish the suspicious cache misses from the genuine ones, in order to increment the counter. Our techniques have proven to be highly accurate and show negligible false alarms in our test cases. Using this methodology, we are also able to detect multiple attacks on multiple victims happening concurrently. The attacks that can be detected by our method follow the technique of flushing the sensitive section of the victim from the cache to perform timing analysis based on CSCAs. Our results show more than 98% detection accuracy and negligible false alarms with a detection speed of 2% from the start of attacker execution. The area overhead and power overhead after logic synthesis of the detection module is only 0.9% to 2% and 1% to 2.1%, for one to five counters, respectively.

There is no clock cycle overhead, as the detection computations are not in the processor critical path, resulting in no impact on its operating frequency. Our proposed calibration and detection algorithms are processor generic. Though we have demonstrated our method on a specific RISC-V processor core, our method is applicable to other processor architectures with inclusive and shared caches.

Our experiments are conducted on an in-order processor core; however, our approach would show similar outcomes on out-of-order processor architectures for the detection of the same targeted attack types.

Currently, marking of the secret sensitive sections in the code is the responsibility of the user. However, as part of future work, automation through statistical program analysis offers a promising avenue for improvement. As demonstrated in [67], techniques such as control flow extraction, taint analysis, and address analysis can be leveraged to automatically identify and mark these sections, streamlining the process and enhancing overall efficiency. We also look forward to extending such a fine-grained detection technique to detect attacks that exploit other vulnerabilities, like speculative execution and out-of-order execution.

REFERENCES

[1] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. 2016. Real time detection of cache-based side-channel attacks using hardware performance counters. *Applied Soft Computing* 49, C (Dec. 2016), 1162–1174. DOI: http://dx.doi.org/10.1016/j.asoc.2016.09.014

[2] 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 *3rd IEEE International Conference on Fog and Mobile Edge Computing (FMEC'18)*. IEEE, 1–6. DOI: http://dx.doi.org/10.1109/FMEC.2018.8364038

[3] 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 *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP@ISCA'18)*, Jakub Szefer, Weidong Shi, and Ruby B. Lee (Eds.). ACM, 1:1–1:8.

[4] JongHyeon Cho, Taehyun Kim, Taehun Kim, and Youngjoo Shin. 2019. Real-time detection on cache side channel attacks using performance counter monitor. In *2019 International Conference on Information and Communication Technology Convergence (ICTC'19)*. IEEE, 175–177.

[5] Berk Gülmezoglu, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2019. FortuneTeller: Predicting microarchitectural attacks via unsupervised deep learning. *CoRR* abs/1907.03651 (2019). http://arxiv.org/abs/1907.03651

[6] Hodong Kim, Changhee Hahn, and Junbeom Hur. 2021. Real-time detection of cache side-channel attack using non-cache hardware events. In *ICOIN*. IEEE, 28–31.

[7] Anh-Tien Le, Trong-Thuc Hoang, Ba-Anh Dao, Akira Tsukamoto, Kuniyasu Suzaki, and Cong-Kha Pham. 2021. A real-time cache side-channel attack detection system on RISC-V out-of-order processor. *IEEE Access* 9 (2021), 164597–164612. DOI: http://dx.doi.org/10.1109/ACCESS.2021.3134256

[8] Rob Oshana, Mitchell A. Thornton, Eric C. Larson, and Xavier Roumegue. 2021. Real-time edge processing detection of malicious attacks using machine learning and processor core events. In *2021 IEEE International Systems Conference (SysCon'21)*. 1–8. DOI: http://dx.doi.org/10.1109/SysCon48628.2021.9447078

[9] Congmiao Li and Jean-Luc Gaudiot. 2022. Detecting spectre attacks using hardware performance counters. *IEEE Transactions on Computers* 71, 6 (2022), 1320–1331. DOI: http://dx.doi.org/10.1109/TC.2021.3082471

[10] Pavitra Prakash Bhade and Sharad Sinha. 2021. Detection of cache side channel attacks using thread level monitoring of hardware performance counters. In *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC'21)*. 210–217. DOI: http://dx.doi.org/10.1109/MCSoC51149.2021.00039

[11] Limin Wang, Lei Bu, and Fu Song. 2022. Locality based cache side-channel attack detection. In *Proceedings of 10th International Workshop*, Vol. 87. 49–65.

[12] Melis Kapotoglu Koc and Deniz Turgay Altilar. 2023. Selection of best fit hardware performance counters to detect cache side-channel attacks. In *Proceedings of the 2023 ACM Workshop on Secure and Trustworthy Cyber-physical Systems (SaT-CPS'23)*. Association for Computing Machinery, New York, NY, USA, 17–22. DOI: http://dx.doi.org/10.1145/3579988.3585052

[13] Stefano Carnà, Serena Ferracci, Francesco Quaglia, and Alessandro Pellegrini. 2023. Fight hardware with hardware: Systemwide detection and mitigation of side-channel attacks using performance counters. *Digital Threats* 4, 1, Article 5 (Mar. 2023), 24 pages. DOI: http://dx.doi.org/10.1145/3519601

[14] Jonas Depoix and Philipp Altmeyer. 2018. Detecting spectre attacks by identifying cache side-channel attacks using machine learning. *Advanced Microkernel Operating Systems* 4 (2018), 75.

[15] Microsoft. 2023. Manually Rebuild Performance Counters for Windows Server 2008 64 Bit or Windows Server 2008 R2 Systems. February 23, 2023. https://learn.microsoft.com/en-us/troubleshoot/windows-server/performance/manually-rebuild-performance-counters

[16] Thomas Gruber, Jan Eitzinger, Georg Hagerand Gerhard Wellein. 2023. LIKWID. Zenodo. DOI: 10.5281/zenodo.10105559

[17] Hassan Shojania. 2008. Hardware-Based Performance Monitoring with VTune Performance Analyzer Under Linux. April 2, 2008. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.94.118. DOI: http://hassan.shojania.com/pdf/VTuneProjectReport.pdf

[18] J. Dongarra, K. London, and S. Moore. 2001. Using PAPI for hardware performance monitoring on Linux systems. In *Proceedings on the Conference on Linux*, 25–27. http://web.eecs.utk.edu/$\sim$shirley/papers/lci2001.pdf

[19] Craig Marcho. 2014. Windows Performance Monitor Overview. https://techcommunity.microsoft.com/t5/ask-the-performance-team/windows-performance-monitor-overview/ba-p/375481

[20] Freescale Semiconductor. 2007. Performance Monitor on PowerQUICC™ II Pro Processors.

[21] 2011. Linux Kernel Profiling with Perf. (2011). https://perf.wiki.kernel.org/index.php/Tutorial

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization (WWC-4) (Cat. No.01EX538)*. 3–14. DOI: http://dx.doi.org/10.1109/WWC.2001.990739

[23] P. Dabbelt C. Garlati G. S. Madhusudan D. Patterson, J. Bennett and T. Mudge. 2019. Embench™: An evolving benchmark suite for embedded IoT computers from an academic-industrial cooperative (towards the long overdue and deserved demise of dhrystone. In *RISC-V Workshop Zurich Proceedings*. ETH Zurich, Zurich, Switzerland.

[24] Minwoo Jang, Seungkyu Lee, Jaeha Kung, and Daehoon Kim. 2020. Defending against flush+reload attack with DRAM cache by bypassing shared SRAM cache. *IEEE Access* 8 (2020), 179837–179844.

[25] Yuxiao Mao, Vincent Migliore, and Vincent Nicomette. 2020. REHAD: Using low-frequency reconfigurable hardware for cache side-channel attacks detection. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW'20)*. 704–709. DOI: http://dx.doi.org/10.1109/EuroSPW51379.2020.00101

[26] Prashant Mata and Nanditha Rao. 2021. Flush-reload attack and its mitigation on an FPGA based compressed cache design. In *ISQED*. IEEE, 535–541.

[27] Kerem Arıkan, Alessandro Palumbo, Luca Cassano, Pedro Reviriego, Salvatore Pontarelli, Giuseppe Bianchi, Oğuz Ergin, and Marco Ottavi. 2022. Processor security: Detecting microarchitectural attacks via count-min sketches. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30, 7 (2022), 938–951. DOI: http://dx.doi.org/10.1109/TVLSI.2022.3171810

[28] Yuxiao Mao, Vincent Migliore, and Vincent Nicomette. 2022. MATANA: A reconfigurable framework for runtime attack detection based on the analysis of microarchitectural signals. *Applied Sciences* 12 (Jan. 2022), 1452. DOI: http://dx.doi.org/10.3390/app12031452

[29] Gildo Torres and Chen Liu. 2016. Can data-only exploits be detected at runtime using hardware events? A case study of the heartbleed vulnerability. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (HASP'16)*. Association for Computing Machinery, New York, NY, USA, Article 2, 7 pages. DOI: http://dx.doi.org/10.1145/2948618.2948620

[30] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. 2011. Security breaches as PMU deviation: Detecting and identifying security attacks using performance counters. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys'11)*. Association for Computing Machinery, New York, NY, USA, Article 6, 5 pages. DOI: http://dx.doi.org/10.1145/2103799.2103807

[31] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*. 1–12. DOI: http://dx.doi.org/10.1109/DSN.2012.6263958

[32] HongWei Zhou, Xin Wu, WenChang Shi, JinHui Yuan, and Bin Liang. 2014. HDROP: Detecting ROP attacks using performance monitoring counters. In *Information Security Practice and Experience*, Xinyi Huang and Jianying Zhou (Eds.). Springer International Publishing, Cham, 172–186.

[33] 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. *SIGARCH Computer Architecture News* 41, 3 (Jun. 2013), 559–570. DOI: http://dx.doi.org/10.1145/2508148.2485970

[34] Alberto Garcia-Serrano. 2015. Anomaly detection for malware identification using hardware performance counters. *CoRR* abs/1508.07482 (2015). http://arxiv.org/abs/1508.07482

[35] Mikhail Kazdagli, Vijay Janapa Reddi, and Mohit Tiwari. 2016. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–13. DOI: http://dx.doi.org/10.1109/MICRO.2016.7783740

[36] Nisarg Patel, Avesta Sasan, and Houman Homayoun. 2017. Analyzing hardware based malware detectors. In *Proceedings of the 54th Annual Design Automation Conference 2017 (DAC'17)*. Association for Computing Machinery, New York, NY, USA, Article 25, 6 pages. DOI: http://dx.doi.org/10.1145/3061639.3062202

[37] Huicheng Peng, Jizeng Wei, and Wei Guo. 2016. Micro-architectural features for malware detection. In *Advanced Computer Architecture*, Junjie Wu and Lian Li (Eds.). Springer Singapore, Singapore, 48–60.

[38] Xueyang Wang and Ramesh Karri. 2016. Reusing hardware performance counters to detect and identify kernel control-flow modifying rootkits. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 35, 3 (2016), 485–498. DOI: http://dx.doi.org/10.1109/TCAD.2015.2474374

[39] Xueyang Wang, Charalambos Konstantinou, Michail Maniatakos, Ramesh Karri, Serena Lee, Patricia Robison, Paul Stergiou, and Steve Kim. 2016. Malicious firmware detection with hardware performance counters. *IEEE Transactions on Multi-scale Computing Systems* 2, 3 (2016), 160–173. DOI: http://dx.doi.org/10.1109/TMSCS.2016.2569467

[40] Xueyang Wang, Charalambos Konstantinou, Michail Maniatakos, and Ramesh Karri. 2015. ConFirm: Detecting firmware modifications in embedded systems using hardware performance counters. In *2015 IEEE/ACM International Conference on Computer-aided Design (ICCAD'15)*. 544–551. DOI: http://dx.doi.org/10.1109/ICCAD.2015.7372617

[41] James Bruska, Zander Blasingame, and Chen Liu. 2017. Verification of OpenSSL version via hardware performance counters. In *Disruptive Technologies in Sensors and Sensor Systems*, Russell D. Hall, Misty Blowers, and Jonathan Williams (Eds.), Vol. 10206. International Society for Optics and Photonics, SPIE, 102060A. DOI: http://dx.doi.org/10.1117/12.2263029

[42] Corey Malone, Mohamed Zahran, and Ramesh Karri. 2011. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the 6th ACM Workshop on Scalable Trusted Computing (STC'11)*. Association for Computing Machinery, New York, NY, USA, 71–76. DOI: http://dx.doi.org/10.1145/2046582.2046596

[43] Bogdan Copos and Praveen Murthy. 2015. InputFinder: Reverse engineering closed binaries using hardware performance counters. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW-5)*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. DOI: http://dx.doi.org/10.1145/2843859.2843865

[44] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*.

[45] Intel Developer. 2021. Intel Processor Tools.https://www.intel.com/content/www/us/en/developer/tools/overview.html#gs.0mqmz3

[46] Linux Kernel. 2021. Perf Tools. https://perf.wiki.kernel.org/index.php/Main_Page

[47] Ingrid Biehl, Bernd Meyer, and Volker Müller. 2000. Differential fault attacks on elliptic curve cryptosystems. In *Advances in Cryptology (CRYPTO'00), Proceedings of the 20th Annual International Cryptology Conference (Lecture Notes in Computer Science)*, Mihir Bellare (Ed.), Vol. 1880. Springer, 131–146.

[48] Mathieu Ciet and Marc Joye. 2005. Elliptic curve cryptosystems in the presence of permanent and transient faults. *Design, Codes and Cryptography* 36, 1 (2005), 33–43. DOI: http://dx.doi.org/10.1007/S10623-003-1160-8

[49] Agustin Dominguez-Oviedo and M. Anwar Hasan. 2009. Error detection and fault tolerance in ECSM using input randomization. *IEEE Transactions on Dependable and Secure Computing* 6, 3 (2009), 175–187. DOI: http://dx.doi.org/10.1109/TDSC.2008.21

[50] Abraham Peedikayil Kuruvila, Anushree Mahapatra, Ramesh Karri, and Kanad Basu. 2021. Hardware performance counters: Ready-made vs tailor-made. *ACM Transactions on Embedded Computer Systems* 20, 5s, Article 65 (Sep. 2021), 26 pages. DOI: http://dx.doi.org/10.1145/3476996

[51] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. 2020. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'20)*. Association for Computing Machinery, New York, NY, USA, 98–105. DOI: http://dx.doi.org/10.1145/3409963.3410490

[52] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. 2021. A cautionary tale about detecting malware using hardware performance counters and machine learning. *IEEE Design & Test* 38, 3 (2021), 39–50. DOI: http://dx.doi.org/10.1109/MDAT.2021.3063338

[53] Boyou Zhou, Anmol Gupta, Rasoul Jahanshahi, Manuel Egele, and Ajay Joshi. 2018. Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS'18)*. Association for Computing Machinery, New York, NY, USA, 457–468. DOI: http://dx.doi.org/10.1145/3196494.3196515

[54] 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 *2019 IEEE Symposium on Security and Privacy (SP'19)*. 20–38. DOI: http://dx.doi.org/10.1109/SP.2019.00021

[55] Yuval Yarom and Katrina E. Falkner. 2013. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. *IACR Cryptology ePrint Archive* 2013 (2013), 448. http://eprint.iacr.org/2013/448

[56] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A generic approach for mitigating spectre. In *NDSS*. The Internet Society.

[57] Simon Rokicki, Davide Pala, Joseph Paturel, and Olivier Sentieys. 2019. What you simulate is what you synthesize: Designing a processor core from C++ specifications. In *Proceedings of the International Conference on Computer-aided Design (ICCAD'19)*, David Z. Pan (Ed.). ACM, 1–8.
[58] Zephyr Project Contributors. 2021. Zephyr Project. Retrieved August 20, 2023 from https://www.zephyrproject.org/
[59] David A. Padua. 2011. TLS. In *Encyclopedia of Parallel Computing*. Springer, 2055.
[60] Svetlin Nakov. November 2018. Practical Cryptography for Developers. https://cryptobook.nakov.com/
[61] Gabrielle De Micheli and Nadia Heninger. 2020. Recovering cryptographic keys from partial information, by example. *IACR Cryptology ePrint Archive* 2020 (2020), 1506.
[62] Fangfei Liu, Yuval Yarom, Qian Ge 0001, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 605–622. http://doi.ieeecomputersociety.org/10.1109/SP.2015.43
[63] Samira Briongos, Pedro Malagón, Juan-Mariano de Goyeneche, and Jose M. Moya. 2019. Cache misses and the recovery of the full AES 256 key. *Applied Sciences* 9, 5 (2019), 944. DOI: http://dx.doi.org/10.3390/app9050944
[64] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a minute! A fast, cross-VM attack on AES. *IACR Cryptology ePrint Archive* 2014 (2014), 435. http://eprint.iacr.org/2014/435
[65] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA nonces using the flush+reload cache side-channel attack. *IACR Cryptology ePrint Archive.* 2014 (2014), 140. http://eprint.iacr.org/2014/140
[66] Mark Randolph and William Diehl. 2020. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography* 4, 2 (2020). DOI: http://dx.doi.org/10.3390/cryptography4020015
[67] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2021. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2504–2519. DOI: http://dx.doi.org/10.1109/TSE.2019.2953709

Received 3 October 2023; revised 31 March 2024; accepted 6 April 2024