Diminisher: A Linux Kernel based Countermeasure for TAA Vulnerability

Ameer Hamza, Maria Mushtaq, Muhammad Khurram Bhatti, David Novo, Florent Bruguier, Pascal Benoit

Diminisher: A Linux Kernel based Countermeasure for TAA Vulnerability

Ameer Hamza[1], Maria Mushtaq[3], Khurram Bhatti[1], David Novo[2], Florent Bruguier[2], and Pascal Benoit[2]

[1] ECLab, Information Technology University, Lahore, Pakistan
[2] LIRMM, Univ. Montpellier, CNRS, Montpellier, France
[3] LTCI, Télécom Paris, Institut Polytehcnique de Paris, France

Abstract. TSX Asynchronous Abort (TAA) vulnerability is a class of Side-Channel Attack (SCA) that allows an application to leak data from internal CPU buffers through asynchronous Transactional Synchronization Extension (TSX) aborts that are exploited by the recent Microarchitectural Data Sampling (MDS) attacks. Cross-core TAA attacks can be prevented through microcode updates where CPU buffers are flushed during Operating System (OS) context switching, but there is no solution to our knowledge that exists for hyper-threaded TAA attacks in which the attacker leaks data from sibling hardware threads through asynchronous abort. In this work, we have proposed Diminisher, a Linux kernel-based detection and mitigation solution for both hyper-threaded and cross-core TAA attacks. Diminisher can be logically divided into three phases, i.e., scheduling, detection, and mitigation. Diminisher is a lightweight tool to prevent TAA vulnerability. The novelty lies in the methodology that we propose enabling easy extensions to cover other hyper-threaded attacks for which no satisfactory solutions exist yet. Diminisher detects and mitigates the TAA attacks around 99% of the time at a low-performance overhead of 2.5%.

Keywords: Side-Channel Attacks · Intel's x86 Architecture · Linux Kernel · Intel TSX, · Hyper-threading · Caches

1 Introduction

Side-Channel Attacks (SCA) have gained a lot of popularity in the last decade, and there are a variety of threatening attacks [6–9, 20–22] that are proposed in the recent past. One of the most recent developments in the SCA domain is the advent of transient execution attacks [20–22] in which the attacker leaks the data by exploiting speculative and out-of-order execution. The Microarchitectural Data Sampling (MDS) vulnerability [4–8] is derived from the same class of transient execution attacks where data is leaked speculatively from internal CPU buffers. Data leakage through SCA is also widespread in virtualized environments [37–39] where many users are bound to share the same physical hardware in the form of Virtual Machines (VMs). SCA can be prevented at various levels including hardware layer, kernel layer, and the application layer. Kernel-based/OS-based mitigation for SCA is more convenient due to additional privileges and the kernel ability to access all hardware resources. One of the drawbacks of hardware-based mitigation is that it requires modifications in hardware, which makes it inconvenient to implement. Moreover, userspace based mitigation techniques suffer from lack of privileges and the lack of access to the system resources.

Recent MDS attacks like Zombieload [7], RIDL [8] and CacheOut [6] exploit the TAA vulnerability [2] that allows unprivileged speculative access to leak data from internal CPU buffers by using asynchronous aborts within an Intel TSX transaction. This vulnerability is present on all Intel CPUs that ship with Intel TSX extension [31], which is added to aid hardware-based locking. The TAA vulnerability, when initially proposed, allows applications to leak data from either cross-core or hyper-threaded core scenarios. Moreover, leaking data from a sibling hardware thread is easier to exploit due to the same physical socket sharing between logical cores.

Intel proposed a microcode update where they used the legacy VERW instruction to overwrite the internal CPU buffers, which Linux uses during context switching to mitigate cross-core TAA attacks [11, 12]. However, this mitigation is not viable for hyper-threaded TAA attacks, where data can be leaked well before context switching [12]. The only workaround that Intel has suggested for this is to either turn off hyper-threading or disable Intel TSX extensions. However, both of these options serve a considerable performance penalty, i.e., Intel hyper-threading provides a performance boost up to 30% [32] and similarly, the Intel TSX provides a performance boost up to 40% [36].

In this work, we propose Diminisher, a Linux kernel-based detection and mitigation solution for the TAA vulnerability. To our best knowledge, there is no prior work published on TAA mitigation. This paper represents the first step towards full TAA mitigation. Our proposed model is divided into three phases, i.e., scheduling, detection, and mitigation. Scheduling utilizes the Intel hyper-threading feature to schedule Linux kernel threads on all available cores in the system to detect the TAA vulnerability on the sibling hardware threads. Detection is performed by monitoring the features that cause TSX abort, which is followed by the mitigation phase. We have proposed two different techniques for mitigation, in the first technique, we terminate execution of the attacker's process. Alternatively, in the second technique, we replace the vulnerable instructions in the attacker's address space. Since the kernel is the most privileged software in the system, it is advantageous to handle runtime detection and mitigation in the kernel space. Diminisher is loaded as a Linux kernel module and covers the scope of the TAA vulnerability [6–8]. In this paper, we make three main contributions:

1. We present Diminisher, a Linux kernel-based countermeasure for hyper-threaded and cross-core TAA vulnerability.

2. We demonstrate the capability of Diminisher to detect and mitigate the TAA vulnerability with high accuracy, low-performance overhead, low latency, and high scalability.
3. We demonstrate that the Diminisher is resilient to the noise generated by the system under various loads.

Diminisher successfully mitigates the TAA vulnerability around 99% of the time at a performance overhead of 2.5%. We have tested our solution under various system load scenarios. In Diminisher, the scheduling and mitigation phases are generic, but the detection phase is specific to the TAA vulnerability. We have designed a scalable solution to enhance it for other hyper-threaded attacks as future work. To add support for a new side-channel attack in Diminisher, only the detection part needs to be updated by analyzing the features according to the nature of the attack. For instance, to add support for Flush+Reload attack [9] in Diminisher, kernel threads can be facilitated to monitor the number of cache flushes by reading the hardware performance counters [24–30], considering the attacker is leaking the data from a sibling hyper-threaded core.

The rest of the paper is organized as follows: Section 2 provides the necessary background to demonstrate the TAA vulnerability. Section 3 discusses the related work. Section 4 describes the methodology for Diminisher. Section 5 demonstrates the experimental results. Section 6 elaborates the discussion and finally, we conclude our paper in Section 7.

2 Background

2.1 Intel TSX

To support transactional memory for speeding up multithreaded applications, Intel introduced Transaction Synchronization Extensions (TSX) as part of the x86 instruction set for hardware transactional memory [13], which is introduced in the Haswell Architecture. TSX allows memory transactions to be set up through XBEGIN and XEND instructions and the code is placed between these instructions to execute transactionally. Transactions are either committed or aborted, committed to the CPU only when all instructions within a transaction are completed successfully. When a transaction is aborted, the microarchitecture state is rolled back to the previous point, i.e., before the transaction, and all executed operations within the transaction are reverted. Transaction aborts can be caused due to various reasons; most commonly, due to memory address conflicts with the sibling hardware thread, where the other sibling hardware thread tries to read or modify the same address used in the transaction. Some instructions and system events like SMIs also abort the transaction [13]. Moreover, the amount of data accessed within a transaction should not exceed L1 and last level cache, respectively [14].

2.2 LFB

Line Fill Buffers (LFBs) are internal CPU buffers along with load and store buffers to keep track of L1 cache misses at the cache line level. LFB entry is allocated for each L1 data cache miss to retrieve data from higher-level cache. This is implemented to avoid cache line stall when multiple load and store misses happen on the same cache line. Instead, waiting happens within the LFB entry until data is retrieved [10, 15]. Despite fetching data from L1 data cache, there also exists an undocumented path, where data evicted from the L1 data cache occasionally ends up inside the Line Fill Buffers (LFBs) [6].

2.3 TAA Vulnerability

A cache line conflict during a clflush operation is one of the reasons for the TSX transaction to abort [1]. Reading the same address that is recently flushed by the clflush instruction causes a TSX abort. In TSX Asynchronous Aborts (TAA) [2, 6–8], the attacker first flushes the address and then attempts to read the data from the flushed cache line, which causes the transaction to abort. However, the processor already allocates an LFB entry for the load instruction just before the faulting load. When the transaction aborts, the load instruction is allowed to proceed speculatively with data from LFB. Since the load is never completed, the load proceeds with the stale value from LFB, which might be the previous memory address loaded by the victim, allowing the attacker to sample LFB data [2, 3]. The processor state reverts back to a point before the transaction, but this leaves a footprint in the cache that can be retrieved by a timing attack such as Flush + Reload [9]. Fig. 1 depicts the TAA vulnerability where the attacker process is continuously leaking the secret data that is accessed by the victim process through the faulting load.

The TAA vulnerability is exploited by multiple MDS attacks [6–8]. RIDL [8] leaks the data from LFBs and mainly focuses on hyper-threaded attacks where the attacker leaks the data from sibling hardware threads as the victim accesses it. RIDL implemented a TAA attack to recover passwords from Linux `/etc/shadow` file by passively listening to all LFB entries and matching the data with previous observations. By this approach, RIDL was able to recover 26 characters from the shadow file after 24 hours. Zombieload [7] extends the RIDL findings to show leakage even without faulting loads. Zombieload also demonstrates LFB leakage from the Cascade architecture, which Intel claimed to be the first MDS-resistant architecture. Zombieload argued that the leakage from TAA is negligible and limited to 0.1 bytes per second. CacheOut [6] also leaks data from LFB but provides an attacker the additional control over LFB entries by selecting the cache line with leakage rate peeks out at 2.85 KiB/s. CacheOut claimed itself to be the first attack to mount the TAA on Whiskey Lake that Intel shipped with MDS mitigation at the hardware layer. In the TAA vulnerability [6–8], the data is leaked from the internal CPU buffers. To address the buffer leakage issue in existing CPUs, Intel released a microcode update that

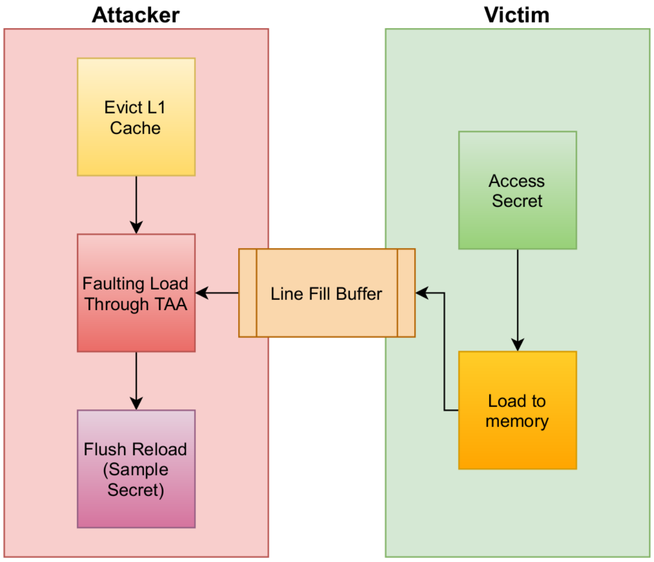

Fig. 1. TAA Vulnerability

reused the legacy VERW instruction to clear the CPU buffers. Operating Systems (OS) like Linux, issue the VERW instruction on each context switch to flush entries in CPU buffers [11, 12]. According to Intel, Whiskey Lake architecture contains the hardware mitigation for MDS attacks, but CacheOut claims to mount a TAA attack due to its ability to select the entry in LFB [6].
However, for hyper-threaded TAA attacks, the buffer overwrite countermeasure is not sufficient as hyper-threading provides the attacker an opportunity to mount the TAA attack well before context switching. Intel suggested that TAA can be mitigated by ensuring only trusted code is ever executed in the sibling threads [8]. However, this strategy introduces nontrivial complexity, as it requires scheduler modifications as well as synchronization at system call entry points. It is also insufficient to protect sandbox applications and SGX enclaves. As a last resort for TAA mitigation in hyper-threaded environments, Intel proposed to disable either the hyper-threading or the TSX component, but both of these options come at the cost of substantial performance overhead [32, 36].
OS-based solutions for side-channel attacks are widespread. Hardware performance counters keep track of all system-level activities like cache hits, cache misses, branch misprediction, etc., which is the reason that they are widely used for SCA detection [24–30, 40, 41]. SmokeBomb [18] protects the sensitive code by manipulating the cache lines in such a way that the attacker is not able to leak the sensitive data. StealthMem [19] is a solution to mitigate cache-based SCA in a cloud environment by locking the cache lines. However, none of the OS-based solutions are proposed for the TAA vulnerability. There are some inspirations from SOA which can be opted for mitigation in OS-based solutions [16, 17].

Thus, the available solutions for these recent attacks (2020-2021) are not satisfactory. Accordingly, in this paper, we propose Diminisher, which is the first OS-based solution that resolves the TAA vulnerability through a novel approach. Diminisher is a lightweight solution that efficiently works for both cross-core and hyper-threaded attack scenarios. The novelty lies in the methodology that we propose for hyper-threaded TAA attacks, where our solution is scalable enough to be expanded for various other side-channel attacks in hyper-threaded environments. Moreover, the proposed mitigation can be separately used to mitigate most of side-channel attacks from the Linux kernel.

3 Methodology

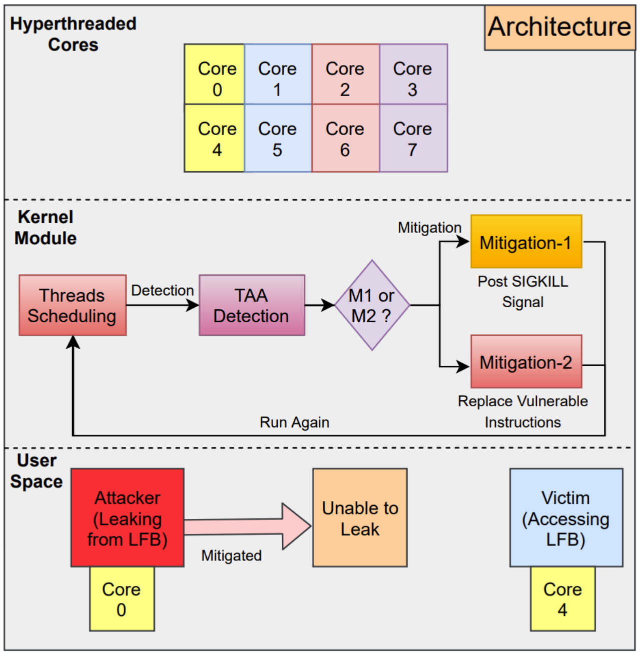

Fig. 2. System Methodology

Our proposed methodology is illustrated in Fig. 2. The upper part of Fig. 2 shows the hyper-threaded core architecture, where the same color cores represent sibling cores, i.e., Core-0 shares the same physical socket with Core-4, Core-1 with Core-5, and so on. The middle part of Fig. 2 represents the architecture of Diminisher, which is logically separated into three phases, i.e., scheduling, detection, and mitigation. Scheduling relies on Intel hyper-threading in which the kernel threads schedule on all available cores in the system. Once scheduled, the next phase is detection which is performed by monitoring the TAA aborts. After the successful detection, the final phase is the mitigation. For mitigation, we have proposed two approaches, i.e., SIGKILL to the attacker's process, and the vulnerable instructions replacement. The lower part of Fig. 2 represents the userspace layer, where the attacker mounts a TAA attack by running on the victim's sibling hardware thread. However, as soon as the Diminisher is scheduled on Core-4, it detects the TAA attack mounted on Core-0 and instantly applies the mitigation, which prevents the attacker to leak any further data.

3.1 Scheduling

Diminisher loads as a Linux kernel module and relies on Intel hyper-threading for scheduling. Hyper-threading is a hardware innovation that allows more than one hardware thread to run on the same physical socket to aid parallel processing through multi-threaded applications [32]. Since hyper-threaded cores share the same physical socket, if an attacker's process is scheduled on one logical core, and one of the kernel threads is scheduled on its sibling hardware thread, that kernel thread can detect the vulnerable process by analyzing the attacker's execution patterns.

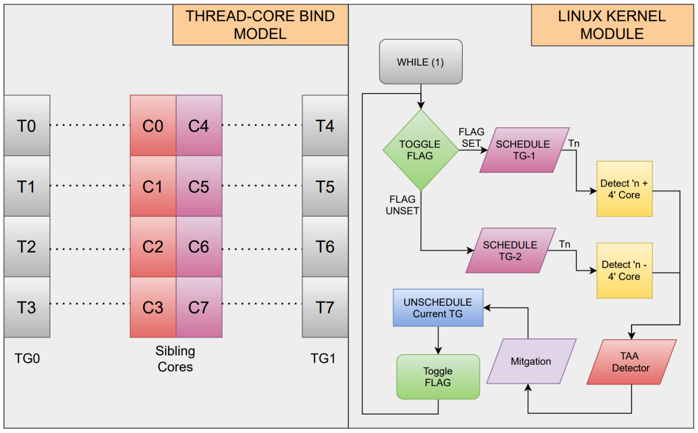

Fig. 3. OS-Based Scheduling

As shown on the left-hand side of Fig. 3, we have affined separate kernel threads (T0 to T7) to all available logical cores (C0 to C7) in the system. C0 and C4 belong to the same physical socket, and so on. Hence, a kernel thread running on C0 may analyze the application's execution patterns running on C4. Since we have four physical cores and eight logical cores in our test machine, we have spawned eight kernel threads (T0 to T7) that are associated with the logical cores (C0 to C8). We have divided the kernel threads into two groups, i.e., thread-group-0 (TG0) and thread-group-1 (TG1). T0 to T3 are assigned to

TG0, whereas T4 to T7 are assigned to TG1.

The scheduling algorithm is based on a Boolean flag which makes two thread groups (TG0 and TG1) to toggle. We schedule TG0 on C0-C3 and detect the attacker's processes on the sibling cores (C4-C7) for a time quantum. Once the time quantum is expired, we toggle the boolean flag which lets TG1 to execute, and TG0 has to wait. This alternate switching is maintained as long as the kernel module is loaded, as shown on the right-hand side of Fig. 3. The scheduling algorithm that we have proposed monitors all cores efficiently with the condition that the processor must support Intel hyper-threading technology.

3.2 Detection

After scheduling, the second phase is a TAA attack detection. Detection is performed by monitoring the number of cache conflicts observed by a kernel thread that is caused due to the sibling hyperthreaded core. In the kernel thread, we continuously mount the TSX transactions to check for cache conflicts, considering that the attacker is running on its sibling hyperthreaded core to leak the data through a TAA attack. If the kernel thread observes a large number of cache conflicts that exceed our calculated threshold, which will be discussed in Section-5.2, then the detection module decides that the TAA attack is mounted on its sibling hardware thread.

We are using three features that are provided as the input to the detection module for efficient detection. We are relying on cache conflicts for TAA-based detection, which are monitored through TSX aborts. Let us suppose that the attacker's process is running on Core-0, the kernel thread instantly detects a TAA attack when it is scheduled on Core-4 by taking into account the thresholds for all three features. Following are the details of the three features:

1. Feature-1 provides the total number of cache conflicts on all the cache sets. This feature provides an overall count of cache conflicts that always exceeds a certain threshold whenever an attack is mounted.
2. Feature-2 provides the aborts count for a particular cache set that observes the maximum number of cache conflicts. When a TAA attack is mounted, few cache sets observe a considerably larger number of aborts than the rest of the cache sets, which are exploited in this feature.
3. Feature-3 provides the aborts count for a cache set that observes minimum cache conflicts. This feature provides a measurement for the cache set that is least affected by a TAA attack, as some cache sets observe a lesser number of conflicts than the rest of the cache sets when the TAA attack is mounted.

The proposed detection framework is reliable because it takes all three features into account, and then it also takes at least 3 consecutive detection decisions to conclude that the system is under attack, thus reducing the chances of false positives. We have calculated separate thresholds for each of the features which will be discussed in Section-5.2.

Detection Implementation Algorithm 1 presents the abstract pseudocode for the TAA detection.

Algorithm 1: TAA Attack Detection

```
$abrt \leftarrow 0$, $no\_abrt \leftarrow 0$, $abrt\_per\_set[SETS] \leftarrow 0$, $samples \leftarrow 0$
while $samples \leq 50000$ do
    for $set \leftarrow 0$ to 64 by 1 do
        Begin_TSX_Transaction();
        Access_All_Ways(set);
        End_TSX_Transaction();
        if $ABORT\_REASON\_CACHE\_CONFLICT$ then
            $abrt \leftarrow abrt + 1$;
            $abrt\_per\_set[set] \leftarrow abrt\_per\_set[set] + 1$;
        else if $NO\_ABORT$ then
            $no\_abrt \leftarrow no\_abrt + 1$;
    $samples \leftarrow samples + 1$;
    Sleep_Detection();
if $((abrt\ /\ no_abrt) * 50000 \geq F1\_THRESHOLD$ &&
$Get\_Max(abrt\_per\_set) \geq F2\_THRESHOLD$ &&
$Get\_Min(abrt\_per\_set) \geq F3\_THRESHOLD)$ then
    /* Attack Detected */
    return 1;
/* No-Attack Detected */
return 0;
```

We are collecting 50,000 samples for each detection, which is a hypothetical limit that is decided after performing the bulk of experimentation. If we go below this limit, we must compromise on accuracy. Similarly, going above this limit increases the latency. Hence, 50 thousand samples are suitable for the TAA detection. For every detection sample, we check for cache conflicts in each of the cache sets. For this, we start the TSX Transaction, access all ways of a particular cache set, and end the transaction. If we get a TSX abort due to cache conflict, the abort count and per set abort count are incremented, otherwise we increment no abort count. After each sample, we suspend the kernel thread to avoid CPU starvation. After collecting all samples, the detection framework makes a valid detection decision by taking into account the measurements of all three features as a mandatory requirement.

3.3 Mitigation

Mitigation is the final phase of our proposed solution. We have proposed two approaches for mitigation that are specific to the Linux kernel and rely on the Linux process descriptor. Our proposed mitigations are not dependent on any hardware resource or a TAA vulnerability, thus, can be used as an independent solution to mitigate most of the side-channel attacks from the Linux kernel. Following are details of the proposed mitigations:

3.4 Mitigation-1 (SIGKILL to the attacker's process)

Linux kernel has control over all userspace processes running in the system including their execution state. After the successful detection, we post a SIGKILL signal from the kernel thread to terminate the execution of the attacker's process.

Algorithm 2 shows the pseudocode for mitigation-1. After getting the task list of the attacker's processes, we lock the task structure and send the SIGKILL signal to the userspace attacker's process. As soon as the attacker application gets the SIGKILL signal, it has no choice but to terminate its execution. Finally, we unlock the task structure and conclude the mitigation. The termination of vulnerable applications avoids any further leakage.

Algorithm 2: Mitigation-1 (SIGKILL to Vulnerable Process)

```
if Is_TAA_Detected() then
    task_struct ← Get_Task_Struct();
    Task_Lock(task_struct);
    ret ← Send_SIG_KILL_Signal(task_struct);
    Task_Unlock(task_struct);
    if ret == SUCCESS then
        /* Killed vulnerable process */
        return 1;
    /* Unable to kill vulnerable process */
    return 0;
```

3.5 Mitigation-2 (Instruction Replacement)

In Linux, each of the instructions that a userspace process executes reside in the code section of the process descriptor. In this mitigation technique, we parse the code section of the attacker's process and check for the vulnerable instructions that cause the attack. As discussed in Section-2.1, XBEGIN and XEND instructions are used to mount the transaction in the case of a TAA attack, i.e., both these instructions are vulnerable in our scenario. Thus, we replace XBEGIN and XEND instructions with the NOP instructions. A NOP instruction in Intel ISA is a harmless instruction that does not update the program flow, which makes it the best choice for the replacement.

Algorithm 3 presents the pseudocode for mitigation-2. We first iterate through the process list to get the task structure of the vulnerable process for the detected core. Afterwards, we acquire the task lock to avoid any race condition and get the userspace pages for the text section of the detected process. Subsequently, we map the user pages to the kernel space, followed by the examination of vulnerable instructions by parsing the code section. For the TAA case, we look for XBEGIN and XEND instructions. We replace all vulnerable instructions with NOP instructions to mitigate the attack. Finally, we unmap the kernel mapping and release the task lock and return.

Algorithm 3: Mitigation-2 (Vulnerable Instruction Replacement)

```
if Is_TAA_Detected() then
    task_struct ← Get_Task_Struct();
    Task_Lock(task_struct);
    user_pages ← Read_Text_Section(task_struct);
    text_ptr ← Map_Kernel_Space(user_pages);
    for i ← 0 to code_size by 1 do
        if text_ptr[i] == VULNERABLE_INSTRUCTION then
            Replace_To_NOP(text_ptr[i]);
            ret ← SUCCESS;
    Unmap_And_Release_Pages(user_pages);
    Task_Unlock(task_struct);
    if ret == SUCCESS then
        /* Replaced Vulnerable Instruction */
        return 1;
    /* Unable to find Vulnerable Instruction */
    return 0;
```

The overall mitigation framework deals with two key challenges; firstly, the possibility of innocent processes being killed due to false positives, and secondly, the possibility that the attacker figures out the presence of mitigation. The first issue is dealt with by a high accuracy detection framework that yields minimum false positives, while for the second challenge, we further improve the mitigation process using the instruction replacement feature as discussed in this section.

4 Experimental Results

4.1 System Model

We have tested the proposed solution on Intel's Core i7-6700 CPU running on Linux Ubuntu 18.04.5 LTS with kernel version 5.4.0-74 at 3.40-GHz. Our threat model is an open-source TAA attack code that leaks the data from sibling hardware threads. The attack is mounted with all the latest mitigation in place for both BIOS and the Linux kernel. The results are obtained by running the TAA attack 25,000 times on different hardware cores and relying on the Diminisher to detect and mitigate the TAA attack.

We observed some variation in the readings for the loaded system compared to the idle system. We call the system loaded when there are 100+ userspace processes running, although we could not reproduce the loaded system scenario with standard stress tester tools. For the loaded system, we executed 40+ Google Chrome tabs, each running YouTube videos, video players, Skype call with video and screen sharing, and various other applications. For the idle system, no additional userspace applications were running except the Diminisher module, attacker process, and the victim process.

4.2 Detection Threshold

As a first step, we calculated the detection thresholds for each of the features. Fig. 4, 5 and 6 show detection thresholds for all three features. The right-hand side of each graph shows the stable readings for the idle system scenario. We hardly get any abort when no TAA attack is mounted for the idle system scenario and in the case of the TAA attack, we mostly get constant readings. In the case of a TAA attack mounted on an idle system, Fig. 4 shows a constant reading line at around 150 value, Fig. 5 shows some variation around 700 value, and the Fig. 6 shows some variations at around 280 value.
In the left-hand graph of Fig. 4, the variations in the readings can be seen for the loaded system. There are quite a few aborts in the case of no TAA attack scenario as cache conflicts are higher when a lot of applications are running in parallel, which causes transactions to abort. If we specifically talk about Fig. 4, after analyzing both loaded and idle graphs, we chose a lower bound of 90 from the loaded scenario, when no attack is mounted, and chose the upper bound of 150 from the idle scenario, where the TAA attack is mounted. Between the lower bound and upper bound, it is safe to detect the attack, so we choose a mid-value of 120 for the feature-1 detection threshold. Similarly, for feature-2 and feature-3, we have calculated the detection thresholds as 500 and 200, respectively, by analyzing Fig. 5 and Fig. 6. Since the behaviors of attack and no attack are quite discernible, a simple threshold determination is very helpful to detect the attack even in load conditions. We have calculated the offline thresholds for all system loads and used the calculated thresholds for the run-time detection.

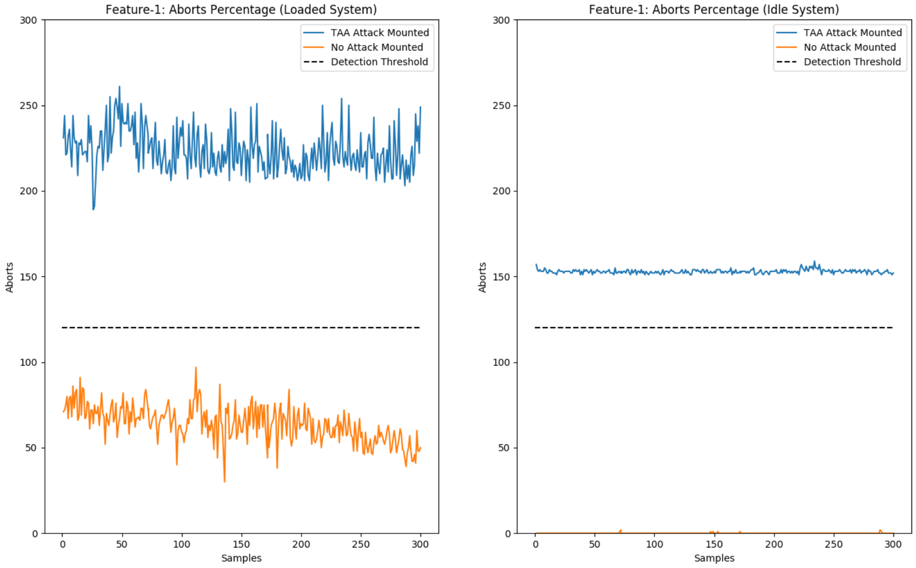

Fig. 4. Feature-1 Threshold Detection

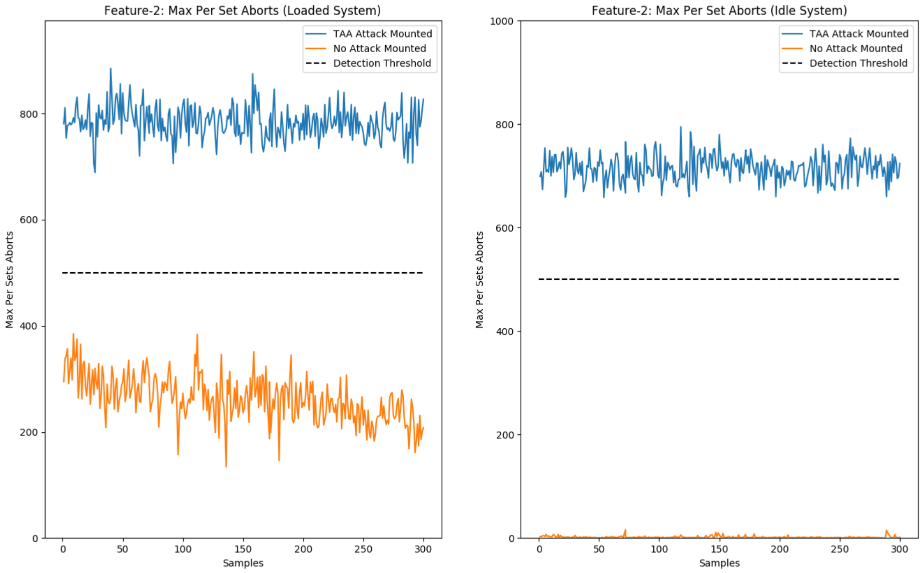

Fig. 5. Feature-2 Threshold Detection

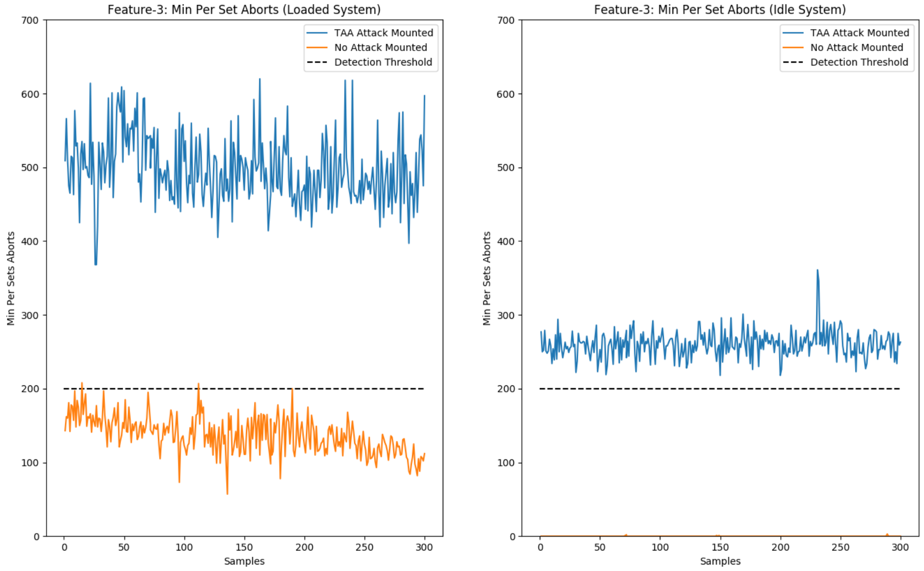

Fig. 6. Feature-3 Threshold Detection

4.3 Results

We have evaluated the proposed solution for both detection and mitigation as shown in table 1. The readings are taken for the loaded system and idle system separately. Diminisher takes three consecutive detections before reporting an attack and extensive experimentation elaborated that on average, 3-times detection results are sufficient to report the attack with reduced inaccuracies in the results. The performance metrics as depicted in Table 1, which shows that the detection proved to be fast and efficient in terms of latency, FPs, and FNs.

Table 1. Experimental Results

System State	**Type**	**Accuracy (%)**	**FP (%)**	**FN (%)**	**Overhead (%)**	**Latency (us)**
IDLE	**Detection**	97.31	2.64	0.03	2.5	5264232
	Mitigation-2	99.94	0.03	0		306
	Mitigation-1	99.85	0.03	0.18		86
LOADED	**Detection**	98.26	1.73	0.03	2.5	6916110
	Mitigation-2	99.91	0.03	0.06		452
	Mitigation-1	99.82	0.03	0.21		106

Accuracy Accuracy is the most convenient way to measure the effectiveness of any runtime detection tool. We have used percentages to demonstrate the accuracy results. We have used the same number of samples for loaded-system, idle-system, attack-mounted, and no attack-mounted scenarios. We have calculated the accuracy in terms of False Positives (FP) and False Negatives (FN). FP is the condition that shows the presence of an attack when there is no attack mounted. On the other hand, FN shows no attack even though when there is an attack mounted. FNs are more critical than FPs as in the case of FNs, the attacker can continue to mount the attack without being detected or mitigated. Table 1 shows the experimental results for both detection and mitigation. We are getting an overall accuracy of around 99% with very few false positives and false negatives for both detection and mitigation scenarios. To reduce the false positives for detection, the detector module detects three consecutive times in a row before making a decision.

Latency Latency is also an essential parameter for the detection and mitigation of any Side-Channel Attack. Latency should be good enough to detect and mitigate the attack before its completion. Latency is also directly related to the performance overhead, i.e., a faster solution would generally cause some performance overhead. This is because the CPU is utilized the most for the faster

solutions, which depletes the CPU resources for general-purpose tasks.
The TAA attack is slower by nature as it takes around 24 hours to leak the root password for the RIDL attack as discussed in Section 2.3. Therefore, latency is not a very big concern for us. Detection takes around 5 seconds in idle conditions and around 7 seconds when the system is loaded. Moreover, our proposed mitigation techniques are significantly faster, i.e., mitigation-2 takes around 500 uS on average as we have to parse the code section, whereas mitigation-1 is even quicker as we just have to post a SIGKILL signal to the vulnerable process.

Overhead Another important design parameter for scheduling and mitigation tools is the performance overhead. The runtime detection tools should keep the overhead minimum, i.e., there should be no significant effect on general-purpose applications running in the system. Furthermore, the adaptability and scalability of the scheduling and mitigation solution are highly dependent on the performance overhead. As discussed earlier, going more fine-grained would result in a faster response but that comes at the cost of higher performance overhead.
We measure the performance overhead based on the CPU utilization after loading the kernel module. The performance overhead of Diminisher is very low, i.e., 2.5% on average. The low system overhead is because we do not let kernel threads to run continuously for all the samples, instead we unschedule the threads after we do a single round of detection on the whole L1 cache. We also did not find performance degradation for any user-space application while Diminisher was loaded.

5 Discussion

Experimental results show that the Diminisher efficiently detects and mitigates the TAA vulnerability with 99% accuracy at 2.5% of performance overhead. Moreover, 2.5% overhead is acceptable as it does not cause any performance degradation in the overall system and Diminisher has a very low count of false positives and false negatives. Threshold-based detection proves to be useful to counter the noise under system load as we perform offline analysis for both the loaded system and the idle system cases to select the appropriate threshold. Since we already know the thresholds for both the load/noise case and idle case, detecting in between the two limits helps us to mitigate the noise. Diminisher is equally effective for both cross-core and hyperthreaded core scenarios.
As discussed in Section 3, there is no prior mitigation proposed for the TAA vulnerability except for the Intel microcode update, however, the microcode update is not effective for hyper-threaded TAA attacks. There are some proposed mitigations for SCAs, e.g., Jonathan Behrens [33] proposed a mitigation for transient attacks and proposed a novel kernel design that is safe for the process to expose. SmokeBomb [18] and StealthMem [19] proposed the mitigations for cache-based Side Channel attacks, but no mitigation is proposed for hyper-threaded TAA attacks or MDS attacks. Hyper-threaded attacks are hard to mitigate because of the sibling core architecture, due to which there is no satisfactory solution for

such attacks until now. Therefore, it is usually suggested to disable the hyper-threading feature to promote secure computing [34].

Diminisher successfully accomplishes all three objectives that were discussed in Section-2 with the following novelties:

- Diminisher resolves the TAA vulnerability with very promising results of over 99% accuracy at 2.5% latency.
- Diminisher can efficiently mitigate hyper-threaded TAA attacks and it is scalable enough to be expanded for other hyper-threaded attacks.
- Proposed mitigation techniques can be used independently to mitigate most of the SCAs from the Linux kernel.
- The mitigation-2 technique helps to reduce DOS attacks as the attacker is unable to figure out the presence of the mitigation framework in the case of this technique.
- Since kernel-based solutions are more autonomous than userspace due to higher privileges, Diminisher effectively detects and mitigates the userspace applications from kernel space.
- We provide simplicity, i.e., Diminisher is simply loaded as a kernel module and does not require any change in kernel code.

Although the proposed solution efficiently resolves the TAA vulnerability, there are some possible improvements that we would like to discuss. Firstly, the TAA detection phase is slower and takes between three and five seconds on average due to the collection of a large number of samples and the detection functionality, which utilizes an iteration of three consecutive detections. Secondly, the CPU buffer monitoring can be added to make TAA detection more robust, which can also help to detect other MDS attacks. Lastly, Diminisher is limited to kernel space only, i.e., it cannot mitigate cross VM SCAs, which can be opted as an extension to this work.

As a future work, we are planning to extend Diminisher to cover other side-channel attacks. The proposed module is efficient in terms of scalability, i.e., more Side-Channel Attacks like Flush+Reload, Prime+Probe, etc., can be added to our module, and for that, only the detection phase needs to be updated according to the type of attack. We were able to successfully run recent transient execution attacks including Spectre [20], Meltdown [21], and Foreshadow [22] in our environment, and Diminisher can be expanded for such attacks as a future work. Diminisher can also be expanded to the hypervisor layer to mitigate cross VM SCAs. There can also be some work done to make the current solution more efficient, i.e., detection latency improvement, feature addition, and performance improvement.

6 Conclusion

In this paper, we have proposed a solution for the TAA vulnerability that works very efficiently at a low-performance overhead of around 2.5% and the accuracy of over 99%. Furthermore, the experimental results depict that there are very few false positives and false negatives in the case of both detection and mitigation. Generally, hyper-threaded attacks are hard to mitigate since multiple logical cores share the same physical socket, and an attacker using one hyper-threaded core can leak data from its sibling hardware thread without being noticed by the OS. With the proposed solution, we can efficiently mitigate hyper-threaded and cross-core TAA attacks, and the solution is scalable enough to mitigate other attacks with some modification in the detection phase. The proposed mitigations can also be used as standalone mitigation techniques for future research.

References

1. Intel. Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 3 (3A, 3B 3C): System Programming Guide, 2016.
2. Intel. Deep Dive: Intel Transactional Synchronization Extensions Asynchronous Abort, http://www.software.intel.com/content/www/us/en/develop/articles/software-security-guidance/technical-documentation/intel-tsx-asynchronous-abort, January 2021.
3. Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling, http://www.software.intel.com/content/www/us/en/develop/articles/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling, May 2021.
4. D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in USENIX Security, Aug. 2020.
5. C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on Meltdown-resistant CPUs," in CCS, 2019.
6. S. V. Schaik, M. Minkin, A. Kwong, D. Genkin, Y. Yarom, "CacheOut: Leaking Data on Intel CPUs via Cache Evictions," IEEE Symposium on Security and Privacy (SP), 2021
7. M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in CCS, 2019.
8. S. van Schaik, A. Milburn, S. Osterlund, P. Frigo, ¨ G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Rogue in-flight data load," in IEEE SP, 2019.
9. Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in USENIX Security, 2014.
10. "Cache memory system having data and tag arrays and multi-purpose buffer assembly with multiple lines buffers," US Patent 5,680,572, Jul 1996.
11. Linux Kernel, Microarchitectural Data Sampling (MDS) mitigation, http://www.kernel.org/doc/html/latest/x86/mds, April 2021.
12. Linux Kernel, TAA - TSX Asynchronous Abort, http://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/tsx_async_abort, April 2021.
13. Intel TSX Overview, http://scc.ustc.edu.cn/zlsc/tc4600/intel/2016.0.109/compiler_c/common/core/GUID-FB2F2539-18F5-4D5A-B814-F29FD0C32326, January 2021.
14. "Intel 64 and IA-32 architectures optimization reference manual," Jun 2016.
15. H. Akkary, J. M. Abramson, A. F. Glew, G. J. Hinton, K. G. Konigsfeld, P. D. Madland, M. S. Joshi, and B. E. Lince, "Methods and apparatus for caching data in a nonblocking manner using a plurality of fill buffers," US Patent 5,671,444, Oct 1996.

16. M. Chiappetta, E. Savas, and C. Yilmaz, “Real time detection of cache- based side-channel attacks using hardware performance counters,” Journal of Applied Soft Computing, vol. 49, pp. 1162–1174, Dec. 2016.
17. M. Mushtaq, D. Novo, F. Bruguier, P. Beniot, M. K. Bhatti, ”TransitGuard: An OS-based Defense Mechanism Against Transient Execution Attacks,” 26th IEEE European Symposium (ETS 2021), May 2021”
18. H. Cho, J, Park, D. Kim, Z. Zhao, Y. Shoshitaishvilli, A. Doupe, G. J. Ahn, ”SmokeBomb: effective mitigation against cache side-channel attacks on the ARM architecture,” MobiSys ’20: Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services, June 2020
19. Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In Proceedings of the 21st USENIX Security Symposium (Security). Bellevue, WA, 189–204.
20. P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in IEEE SP, 2019.
21. M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, “Meltdown: Reading kernel memory from userspace,” in USENIX Security, 2018.
22. J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-oforder execution,” in USENIX Security, 2018.
23. E. Goktus, K. Razavi, G. Portokalidis, H. Bos, C. Giuffrida, ”Speculative Probing: Hacking Blind in the Spectre Era,” CCS ’20: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, October 2020
24. X. Wang and R. Karri, “Reusing hardware performance counters to detect and identify kernel control flow modifying rootkits,” IEEE TCAD, vol. 35, pp. 485–498, March 2016.
25. J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. J. Stolfo, “On the feasibility of online malware detection with performance counters,” in ISCA, 2013.
26. A. Tang, S. Sethumadhavan, and S. J. Stolfo, “Unsupervised anomaly based malware detection using hardware features,” CoRR, 2014.
27. M. A. et al, “Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel attacks.” Crypt. ePrint Arch., 2017. https://eprint.iacr.org/2017/564.
28. M. Chiappetta, E. Savas, and C. Yilmaz, “Real time detection of cache-based side-channel attacks using hardware performance counters,” Appl. Soft Comput., vol. 49, pp. 1162–1174, Dec. 2016.
29. G. Torres and C. Liu, “Can data-only exploits be detected at runtime using hardware events?: A case study of the heartbleed vulnerability,” in HASP, pp. 2:1–2:7, 2016.
30. T. Zhang, Y. Zhang, and R. B. Lee, “Cloudradar: A real-time side-channel attack detection system in clouds,” in RAID 2016.
31. Intel TSX Overview, http://software.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-intel-advanced-vector-extensions-2/intrinsics-for-intel-transactional-synchronization-extensions-intel-tsx/intel-transactional-synchronization-extensions-intel-tsx-overview, January 2021.

32. Intel Hyperthreading https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading, June 2021
33. J. Behrens, A. Cao, C. Skeggs, A. Belay, M. F. Kaashoek, N Zeldovich, "Efficiently Mitigating Transient Execution Attacks using the Unmapped Speculation Contract, " 14th USENIX Symposium on Operating Systems Design and Implementation, 2020
34. M. Mushtaq, M. A. Mukhtar, V. Lapotre, M. K. Bhattic, G. Gogniat, "Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA", Information Systems, Volume 92, September 2020, 101524
35. O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," https://foreshadowattack.eu/foreshadowNG.pdf, 2018.
36. R. M. Yoo, C. J. Hughes, K. Lai, R. Rajwar, "Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing" 14th USENIX SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013
37. Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An Exploration of L2 Cache Covert Channels in Virtualized Environments," in Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, ser. CCSW '11. New York, NY, USA: ACM, 2011, pp. 29–40. [Online]. Available: http://doi.acm.org/10.1145/2046660.2046670
38. Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack," in Proceedings of the 23rd USENIX Conference on Security Symposium, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 719–732. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671271
39. Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM Side Channels and Their Use to Extract Private Keys," in Proceedings of the 2012 ACM Conference on Computer and Communications Security, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 305–316. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382230
40. M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat, "NIGHTs-WATCH: A cache-based side-channel intrusion detector using hardware performance counters," in Proc. 7th Int. Work- shop Hardw. Architectural Support Secur. Privacy, New York, NY, USA, Jun. 2018, pp. 1–8.
41. M. Mushtaq, J. Bricq, M. K. Bhatti, A. Akram, V. Lapotre, G. Gogniat, and P. Benoit, "WHISPER: A tool for run-time detection of side-channel attacks," IEEE Access, vol. 8, pp. 83 871–83 900, 2020.