Exploration and Exploitation of Hidden PMU Events

Yihao Yang[†], Pengfei Qiu[†§*], Chunlu Wang[†], Yu Jin[†], Qiang Gao[†], Xiaoyong Li[†],
Dongsheng Wang[‡§], Gang Qu[¶]

[†]Key Laboratory of Trustworthy Distributed Computing and Service (BUPT), Ministry of Education, Beijing, China
[‡]Tsinghua University, Beijing, China [§]Zhongguancun Laboratory, Beijing, China [¶]University of Maryland, USA
khaosyg@gmail.com, {qpf, wangcl}@bupt.edu.cn, lambda.jinyu@gmail.com,{2020140851, lixiaoyong}@bupt.edu.cn
wds@tsinghua.edu.cn, gangqu@umd.edu

***Abstract*—Performance Monitoring Unit (PMU) is a common hardware module in modern processors that monitors the processor's architectural and microarchitectural events (PMU events) for CPU performance analysis and optimization. Vendors publish PMU events in documents such as Intel's Software Development Manual (SDM) and ARM processor technical reference manuals. In this paper, we report our findings that these documented PMU events are only a very small portion of the PMU event space. We define hidden PMU events as those that can be triggered in the instruction's execution but are not documented by the vendors. The hidden PMU events may not be as useful as the documented ones for CPU performance analysis. However, they might introduce security vulnerabilities. We develop an automated tool to traverse all the possible PMU events during the execution of each valid instruction to locate the hidden PMU events. On six Intel processors with different micro-architectures, where there are about 307 documented PMU core events on average, our tool finds an average of 17,361 hidden PMU events. We further demonstrate the security implications in both defense and attack of these hidden PMU events. Our experimental results show that up to 6,613 hidden PMU events on the i7-6700 can be used to detect transient execution attacks and 1,192 hidden PMU events can be exploited for side-channel attacks.**

***Index Terms*—Performance Monitoring Unit, Microarchitecture Security, Transient Execution Attack**

I. Introduction

Hardware Performance Counter (HPC) is a widely utilized hardware monitoring tool in today's computer architectures. These counters can be used to measure CPU-level events, including instruction execution, cache hits or misses, branch prediction, and so on. HPC holds significant importance in performance analysis, code debugging, and optimization. Most modern processor vendors offer HPC support for their processors [1]–[3]. In the case of Intel processors, the functional unit utilized to support HPC is known as the Performance Monitor Unit (PMU) [3].

The Intel official documents disclose hundreds of PMU events [4] (different on different architectures) for software developers to measure various architectural and microarchitectural events. Those events are beneficial in debugging codes and improving performance. PMU has also been utilized for security purposes such as malware detection and defense [5], [6], microarchitectural attack detection [7]–[9], and reverse engineering [10]–[13]. In addition, some evaluation tools have been designed using PMU for different application scenarios such as PAPI [14], perf_event [15], and VTune [16]. These tools also provide ways for software developers to easily monitor the execution process of their codes [17], [18]. Besides being used for positive work, PMUs have been utilized to speculate the keys of encryption algorithms such as AES [19], RSA [20], and ECC [21]. In addition, Qiu et al. [22] discovered that PMU counters not only record the behaviors of genuinely committed instructions but also capture the actions of transient instructions. They leveraged this characteristic to create a new PMU-based side channel, based on which they replicated the Foreshadow attack to compromise the security of Intel SGX [23].

Intel provides a 32-bit Model Specific Register (MSR) named `IA32_PERFEVTSELx` for configuring PMU [3], the functions of every bit are illustrated in Fig. 1. The higher 16 bits specify the working model of PMU and the lower 16 bits are used for PMU event selection. In the lower 16 bits of this MSR, the `EventSelect` field determines the general category of PMU events, while the `UMask` field specifies the event selection conditions. The combination of these 16 bits encompasses a substantial portion of all the possible PMU events. Theoretically, the complete event selection space amounts to 2^{16} possibilities. However, take Intel's latest Alderlake microarchitecture CPU as an example, there are only 292 publicly available PMU events in the Intel Software Development Manual (SDM) [4]. This represents a very small subset of the entire PMU event space and leaves room for numerous undocumented PMU events.

Zhao et al. [24] discovered a new contention-based side-channel attack. They then utilize PMU to analyze the performance and reverse engineer the vulnerability source of their proposed side-channel attack. They found two unrecorded PMU events in this process and named them `L1D.READ_REQS` and `L1D_BLOCKS.FALSE_DEPS`. Nick Gregory et al. [25] traversed the 2^{16} PMU event space to find PMU events that are Spectre-sensitive, and they eventually came up with 81 unrecorded PMU events. However, their work does not aim at hidden PMU events and therefore does not delve deeper into research in exploring and exploiting the hidden PMU events.

In this work, we systematically traversed the x86 instruction set and meticulously recorded the PMU events triggered by

*Corresponding author

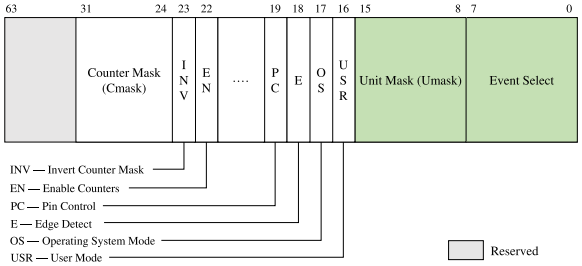

Fig. 1: Layout of IA32_PERFEVTSELx MSR [3].

each instruction. Our experimental results illustrate that there are tens of thousands of undocumented PMU events across various microarchitectures, which is a couple of orders of magnitude more than the existing disclosed PMU events.

We designed a method to collect the hidden PMU events, which traverses all the valid x86 instructions obtained by the uops.info [26] dataset, and then observes the change in the count of the entire event space after the execution of the different instructions. We identify the hidden PMU events as those recorded by their corresponding PMU counters but not documented in the Intel SDM [3]. Through this approach, we collected a large number of hidden PMU events on each of six different microarchitectures (Skylake, Kabylake, Cascadelake, Coffeelake, Haswell, and Alderlake) of Intel processors.

We conducted real experiments to demonstrate the effectiveness of using these PMU events to detect transient execution attacks or construct side-channel attacks. In the former, we considered the well-known transient execution attacks including Meltdown [27], Spectre [28], [29], and Zombieload [30] attacks as the detect targets, and found that there are up to 6,613 available hidden PMU events on i7-6700 that can detect these attacks. In the latter, we aim to implement these attacks using the PMU side-channel attack proposed by [22]. The experimental results show that the number of hidden PMU events that can achieve the attack is 1,192.

In summary, this paper has the following contributions:

- We developed a method to automatically locate hidden PMU events. The proposed method identified a large amount (the average number is 17,361) of hidden PMU events on each of the six Intel processors with different microarchitectures.
- We utilized the collected hidden PMU events to detect transient execution attacks. Through our real experiments, we identified 454 events suitable for meltdown detection, 1,979 for spectre_v1 detection, 4,488 for spectre_v2 detection, 3,696 for spectre_v4 detection, 1,545 for zombieload_v1 detection, 761 for zombieload_v2 detection.
- We employed the hidden PMU events to construct side channels for microarchitecture attacks. Specifically, we discovered 357 hidden PMU events suitable for building side-channel for Meltdown, 1,094 for Spectre, and 139 for Zombieload.

II. Background

A. Performance Monitor Unit

The Performance Monitoring Unit (PMU) is a crucial hardware module embedded in modern processors. It consists of a collection of performance counters responsible for recording a wide range of hardware performance events that take place at the CPU level during system runtime [31]. Intel categorizes the hardware events supported by its performance counters into two types: architectural performance events and non-architectural performance events, also known as microarchitectural events [3]. Architectural performance events refer to events that have consistent behavior across processor architectures. Non-architectural performance events are specific to the microarchitecture of the processor and demonstrate distinct behavior across various microarchitectures. Furthermore, these events may vary even further with processor enhancements. Non-architectural performance events can be classified into core events and uncore events. Core events refer to events on the CPU logical core, while uncore events mean events outside the CPU core. Additionally, starting with the Cascadelake-X microarchitecture, offcore events are introduced, which are specific types of core events requiring special configuration [4]. In this paper, our focus will be exclusively on core events, excluding offcore events and uncore events.

Intel offers users both fixed counters and programmable counters for monitoring performance [3], [32]. The fixed counters are dedicated to tracking predetermined events such as logical cycles, reference cycles, and others. The programmable counters are supported by a set of one-to-one correspondence event selection MSRs (`IA32_PERFEVTSELx`, shown in Fig. 1) and performance count MSRs (`IA32_PMCx`). The `IA32_PERFEVTSELx` MSRs start at address `186H` and occupy a contiguous block of MSR address space. Each `IA32_PERFEVTSELx` register starting at this address corresponds to an `IA32_PMCx` register to start at `0C1H`. To obtain the value of the performance counter, Intel provides two methods: Polling or Processor Event-Based Sampling (PEBS) [3], [32].

a) Polling: The user selects the specified event by changing the value of `IA32_PERFEVTSELx` and then reads from `IA32_PMCx` to obtain the number of times the event occurred. For this purpose, Intel provides specific instructions (`RDMSR`, `WRMSR`) to do reads and writes to the MSR.

b) PEBS: This is a sampling method based on the Performance Monitoring Interrupt (PMI). `IA32_PEBS_ENABLE` provides 4 bits of data indicating which `IA32_PMCx` overflow condition to enable will trigger the PMI, resulting in the capture of the PEBS record.

B. Side Channel Attacks

Within the microarchitecture, various shared resources exist, including Cache, Translation Lookaside Buffer (TLB, execution ports, and more [33]. Through side-channel attacks, the attacker does not directly target the victim's data but

instead deduces secret information by analyzing unintended side information (e.g., changes in voltage frequency, cache timing, etc.) that the microarchitecture unintentionally leaks.

In microarchitecture, the most prevalent side-channel attacks involve the exploitation of cache, such as Flush+Reload [34], Prime+Probe [35], CacheBleed [36], and others. Additionally, there are side-channel attacks that target other shared resources, including TLBLeed [37], PortSmash [38], Binoculars [24], etc. Moreover, there are side-channel attacks resulting from contention or switching in the front-end decoding component, such as SecSMT [39], Leaky Frontends [33], and others. The fundamental principle behind these attacks often relies on timing differences resulting from shared resource contention. Qiu et al. [22] introduced a PMU-based side channel, where the PMU captures and records diverse microarchitectural states. By analyzing the PMU event counts, the attacker can deduce the victim's information.

C. Transient Execution Attacks

Transient execution attacks arise from aggressive optimization techniques employed by modern processors to enhance performance such as Out-of-Order Execution and Branch Prediction. These techniques can result in the execution of instructions that should not be executed, which is called transient execution. Even though these transient instructions are not officially committed, they can affect the microarchitecture state. Attackers can exploit side channels to capture these microarchitectural state changes and infer the victim's confidential data. Typical transient execution attacks include Meltdown [27], Spectre [28], [29], Foreshadow [40], [41], Zombieload [30], etc.

III. Hidden PMU Events Collector

A. Motivation

As mentioned in Section I, PMUs are capable of capturing specific types of hardware events in the CPU. Nowadays, PMUs are extensively utilized in various work scenarios. However, the events available for monitoring through PMUs represent only a small portion of the entire event space. It is crucial to investigate whether the hidden PMU events can also be useful in these scenarios. Furthermore, PMUs play a significant role in reverse engineering tasks, and it is worth exploring whether Intel's undisclosed PMU events hint at the existence of undisclosed CPU hardware components. By delving into these aspects, we can gain valuable insights into the potential capabilities and hidden features of the CPU architecture.

Furthermore, due to the fine granularity of PMUs, there are potential security risks associated with them. An attacker could exploit PMUs to detect processor data and instruction flows or create side channels to leak sensitive information. Considering the existence of numerous hidden PMU events, the security risks they pose could be even more significant. Therefore, it is crucial to thoroughly investigate and analyze hidden PMU events, both in terms of their potential positive applications and their potential negative implications for security.

B. Challenges

As we mention in Section I, the lower 16 bits of `IA32_PERFEVTSELx` register in Fig.1 determines most of the known PMU events. Therefore, we first consider traversing this 2^{16} event space. However, a mere traversal does not ensure thoroughness and rigor in our investigation. Considering that the PMU records the CPU's behavior, which directly correlates with the executed instructions. So, we endeavor to execute different instructions to trigger different behaviors to cover the entire hidden event space as much as possible. However, we have also encountered some challenges in this process:

a) x86 Instructions Traversal: To collect hidden PMU events, we attempted to execute all x86 instructions to trigger as many PMU events as possible. However, the complexity of the x86 instructions presented us with a substantial challenge. We have compiled 5,492 instructions based on the uops.info [26] dataset, which have different behaviors depending on the processor mode and privilege level. In addition, the x86 architecture has evolved with numerous instruction set extensions that mandate specific floating-point units and registers, potentially varying across different CPUs. Moreover, there exist multiple operand types for the same instruction, further exacerbating the complexity of the instruction set as a whole. Undoubtedly, the intricate and diverse nature of the x86 instruction set has presented us with numerous challenges.

b) Non-deterministic: Previous research by Weaver et al. [42] has demonstrated that PMU counting inherently possesses non-deterministic characteristics and may lead to overcounting, attributable to its architectural design. This non-deterministic poses a significant challenge when attempting to ascertain the validity of hidden PMU events, especially when their counts approach zero during the collection process. Regarding this non-deterministic nature, Das et al. [32] have highlighted that works on malicious attack defense and detection are more susceptible to such effects. This is due to their reliance on detecting subtle hardware-level impacts caused by attacks. Therefore, in section IV and V, we filter out this non-deterministic by constructing microarchitectural attack detection models and leveraging side-channel attacks utilizing the hidden PMU events.

C. Collect Hidden PMU Events

To begin, we process the uops.info [26] data set. We strive to confine the usage of registers within a limited range, which facilitates operand filling. It is worth mentioning that certain instruction extensions may necessitate the use of specific registers. Therefore, we need to adapt them according to the instruction extensions supported by the CPU. Furthermore, careful consideration is given to the arrangement of jump instructions. The jump target position is positioned after the jump instruction to prevent dead loops within the program. Finally, we compiled a list of 5,492 instructions.

Given the lack of detailed information regarding instruction execution, it becomes necessary to address all potential exceptions that may arise during execution. The most effective approach for handling exceptions is to leverage the Intel

TABLE I: Hidden PMU Collection Results.

Micro-Architecture	CPU	Compilation Success*	Execution Success*	Hidden PMU Events**(≥)	Documented Core Events**
Sky lake	i7-6700	3576	3484	20599	298
Kaby lake	i7-7700	3574	3478	20230	298
Alder lake	i9-13900k	3628	3529	12503	292
Cascade lake	Xeon Silver 4210R	4724	4626	18438	322
Haswell	Xeon E5-2678 v3	3558	3466	16996	338
Coffee lake	Xeon E-2224	3580	3485	15401	298

* Total instructions tested is 5492. Instructions that Compilation Success but do not Execution Success mean they abort during execution due to exceptions.
** Only Core Events have been discussed in this paper.

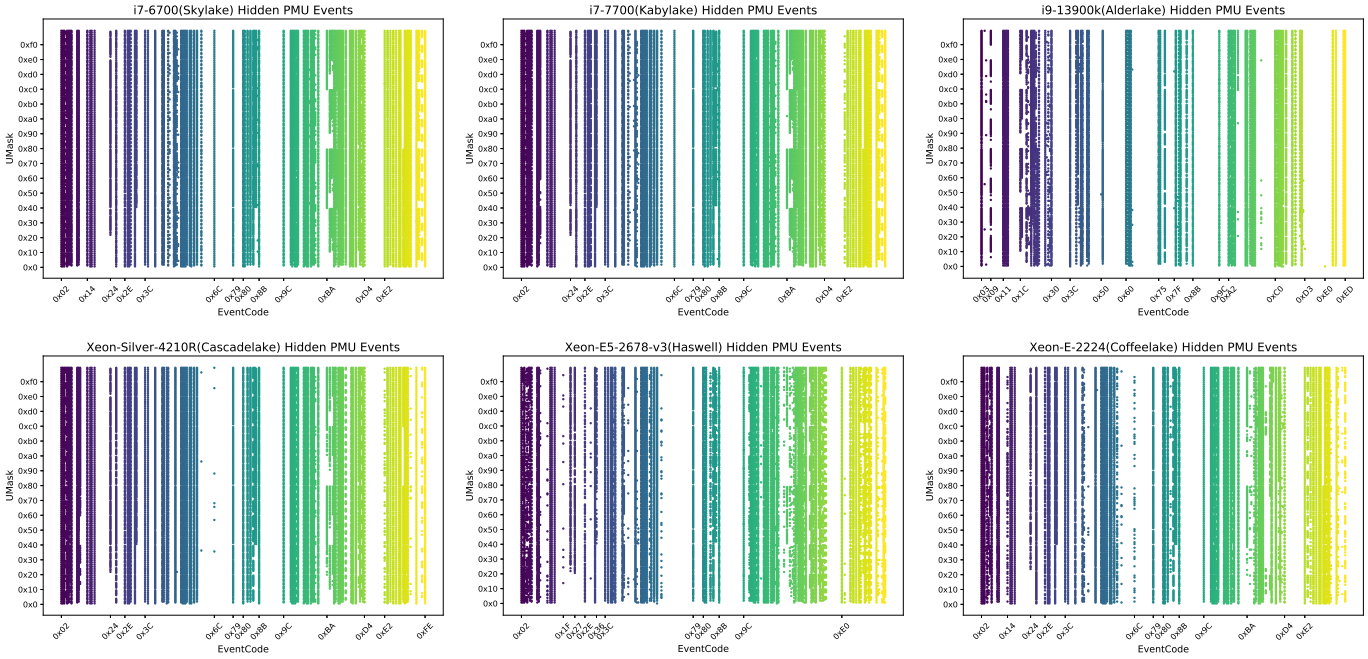

Fig. 2: Distribution of Umaks and EventCode for Hidden PMU Events on Different Microarchitectures.

Transactional Synchronization Extensions (TSX) [43], which allows for efficient and speedy suppression of exceptions. Unfortunately, because the success rate of transient execution attacks can be greatly improved with TSX, many new Intel CPUs do not support this extension. Therefore, we bind all exception signals to custom exception handlers to prevent program crashes.

Then, we proceed by populating a limited number of registers with appropriate values or addresses, based on the CPU's supported instruction set extensions, to accommodate the operand types of the instructions. Finally, we monitor the changes in count within the 2^{16} event space both before and after the execution of each instruction. We record the corresponding instructions and the events in which the count is not equal to 0 and not in Intel's officially public PMU list [3], [4].

D. Collection Result Analysis

As shown in Table.I, We performed collection experiments on six CPUs with different microarchitectures. This is one of our collection results, which may fluctuate due to the Non-deterministic PMU event counts, but will not change by orders of magnitude. As emphasized in Section II, our collection is done on the same logical core, so we are only concerned with core events in this paper. Finally, We successfully compiled 3,412 instructions on the i7-6700 (Skylake) and collected 20,599 hidden PMU events. Even the least i9-13900k (Alder-lake) still has 12,503 hidden PMU events.

However, we do not suppose that each of these PMU events corresponds to a distinct microarchitectural behavior. For the EventCode, we found that it is not continuous, this may mean that these events do exist. In the case of UMask, its distribution makes us wonder if it is determined only by specific bits. For instance, when analyzing the hidden PMU event with EventCode `0x6C` on the i7-6700, we observe a consistent incremental pattern in Fig. 2. Further analysis reveals that its UMask values tend to follow a progressive pattern, with values like `0x*1`, `0x*3`, `0x*5`, `0x*7`, `0x*9`, `0x*B`, `0x*D`, `0x*F`. From a binary perspective, it is noteworthy that the lowest bit of the UMask values associated with these events is consistently set to `1`. Moreover, there are several other instances, such as `0x9C`, `0xBA`, and others, that exhibit a similar pattern. So we suspect that the value of UMask may

be determined by a specific bit.

IV. Application 1:Detecting the Transient Execution Attacks

To further showcase the effectiveness and the exploitability of hidden PMU events, we aim to leverage these hidden PMU events to detect existing transient execution attacks, such as Meltdown [27], Spectre [28], [29], Zombieload [30].

A. Detection Method Design

In our detection approach, since we do not know the microarchitectural behavior corresponding to each hidden event, we cannot select some specific events for multidimensional detection as in previous transient execution detection approaches [7], [44]. Instead, we examine each hidden PMU event and monitor its relationship with transient execution attacks. For each attack, we collect the count changes for each PMU event in the `Clean`, `No-Attack`, and `Attack` states. The classifier is trained offline by a machine learning (ML) algorithm and then analyzes the model training results. This enables us to assess whether a particular hidden PMU event can be utilized effectively for detecting a specific transient execution attack.

B. Detection Experiment Setup

a) Data Collection: For each hidden event, we collect the count of `Clean`, `No-Attack`, and `Attack` states on the i7-6700 (Skylake). The `Clean` state refers to a clean environment where only the victim process is running, to simulate an environment where no malicious attacks exist. For data collection in the `Attack` environment, we separately have various transient execution attacks running on different logical cores of the same physical core as the victim process, and the monitoring program running on the same logical core as the attacker process, for better data collection. The data collection for each transient execution attack is independent. Here we mainly consider 6 transient execution attacks, namely spectre_v1 [29], spectre_v2 [29], meltdown(spectre_v3) [27], spectre_v4 [28], zombieload_v1 [30], and zombieload_v2 [30]. These attacks are selected as representative examples to assess the effectiveness of our detection approach and evaluate the behavior of hidden events in the context of different transient execution vulnerabilities.

Finally, to improve the capability to differentiate false positives (FP), we conduct data collection in the `No-Attack` environment. In this setup, we disable the attack primitive of the attacker process while keeping the rest of the code unchanged. We collect the data under the same settings as in the `Attack` environment. Additionally, we simulate real scenarios by engaging in activities such as browsing websites, reading, and writing text files. To further enhance the ability to distinguish FP, we incorporate stress tests into the system. These stress tests include (1) a memory-intensive load that executes `memcpy` to copy 2MB of data from a shared region to a buffer, followed by `memmove` to move the data in the buffer, and (2) a CPU-intensive load that performs complex floating-point operations in a loop [7]. This additional data helps us refine the accuracy and reliability of our detection model.

b) Data Processing & Model Training: In the field of classification, various machine learning models can be used, such as logistic regression (LR) [45], support vector machine (SVM) [46], [47], and more. Considering that our task involves detecting transient execution attacks using a substantial number of hidden PMU events (20,599 events for each attack), we opt for the logistic regression algorithm due to its relatively shorter training time. Logistic regression is a linear classification algorithm that estimates the probability of a given input using a `Sigmoid` function [45]. It is a straightforward and efficient algorithm with fewer parameters, making it suitable for our scenario. While more complex algorithms are theoretically possible, we prioritize LR as it serves as a baseline to determine if attacks can be detected effectively. If LR proves capable of detecting attacks, further exploration with more sophisticated algorithms can be pursued.

Next, we proceed with data labeling. The PMU count changes collected in the `Attack` environment are assigned a label of `1`, indicating the occurrence of an attack. Conversely, data collected in other environments are labeled as `0`, indicating the absence of an attack. Each type of attack is recorded separately, and data is collected in multiple independent runs. To ensure fairness and mitigate bias, we maintain an equal number (2,000) of samples for both attack and non-attack categories.

After that, we split the collected dataset into training data (70% of the samples) and test data (30% of the samples). The training data is utilized to train the logistic regression model, while the test data is used for evaluating the model's performance. By analyzing the model's performance on the test data, we can assess the effectiveness of hidden PMU events in detecting transient execution attacks.

C. Experiment Results

To comprehensively evaluate the performance of the detection model, we employ several metrics in addition to the commonly used $Accuracy$ metric. These additional metrics include $Precision$, $Recall$, FNR, FPR ,$F1\_Score$, and $Area\ Under\ Curve(AUC)$. $Precision$ measures the proportion of true positive (TP) samples predicted correctly as positive out of the total samples predicted as positive. It indicates the model's ability to avoid false positives (FP). $Recall$, also known as the $TP\ Rate$ or $Sensitivity$, measures the proportion of positive samples that are correctly identified as positive. It assesses the model's ability to detect TP samples effectively. FNR represents the proportion of FN to the overall positive sample. FPR indicates the proportion of FP to total negative samples which indicates the false alarm rate. $F1\_Score$ is the harmonic mean of $Precision$ and $Recall$. It provides a balanced assessment of the model's performance by considering both $Precision$ and $Recall$. A higher $F1\_Score$ indicates better overall performance. AUC is a metric derived from the $Receiver\ Operating\ Characteristic(ROC)$ curve.

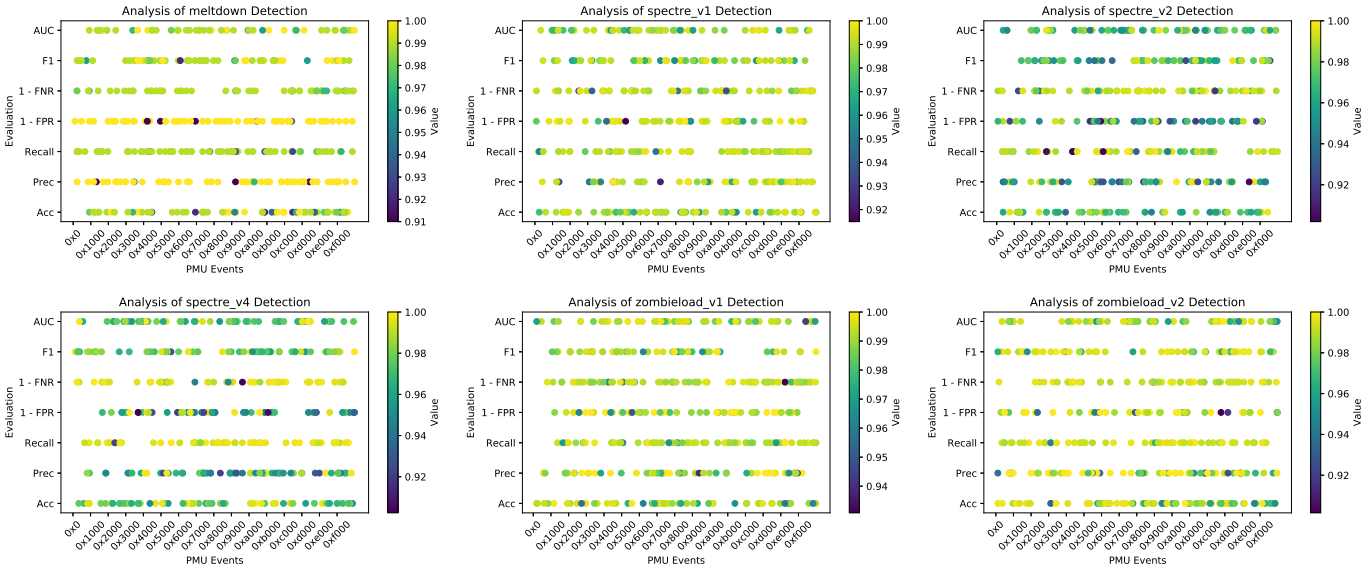

Fig. 3: Transient Execution Attack Detection Model Evaluation.

```
zero_pmu();
if(xbegin()==(~0u)){
    asm volatile(
        "cmp (%0), %1"
        "jz equal"
        "nop"
        "jmp end"
        "equal:"
        "   ins1 (eg. movq (%%rax),%%rax)"
        "end:"
        "   ins2 "
        :
        :"r"(The address of Secret),
         "r"(Controllable Variable V)
        :
    );
    xend();
}
read_pmu();
```

Listing 1: Encoding Secret into PMU Side Channel.

The ROC curve illustrates the trade-off between $Recall$ and FPR at various classification thresholds. AUC measures the overall performance of the detection model in distinguishing between malicious and normal executions. A higher AUC value signifies a more effective model in correctly classifying samples.

We screened the detection models with $Accuracy > 0.9, F1 > 0.9, FNR < 0.1, FPR < 0.1, AUC > 0.9$ and their PMU events Number, and then randomly sampled 400 points to draw a scatter plot as Fig. 3. As a result, we got 454 hidden PMU events suitable for meltdown detection, 1,979 for spectre_v1 detection, 4,488 for spectre_v2, 3,696 for spectre_v4, 1,545 for zombieload_v1, and 761 for zombieload_v2.

V. Application 2:Implementing the Side Channel Attacks

In order to demonstrate the potential security threat of hidden PMU events, in this section, we attempt to recover private data leaked by transient execution attacks using the hidden PMU events to construct the side channel.

A. Encoding The Secret Data into PMU

The work conducted by Qiu et al. [22] revealed that certain instructions executed within the transient window can impact the PMU count. Building upon this observation, they devised an instruction gadget capable of encoding confidential data into the PMU side channel, as shown in Listing 1. First, the side channel state is cleared, as in the first line. Then, in the fourth line, a transient execution is triggered through a comparison operation between the secret data and a controllable variable `V`. If the secret data is equal to `V`, the execution path of the instructions changes, leading to the execution of `ins1`. The PMU event associated with `ins1` can be used to deduce whether `ins1` was executed or not by monitoring the change in the PMU count. Exactly, if there is a change in the PMU count, it indicates that `ins1` has been executed. Based on this information, we can infer that the secret data is equal to the controllable variable `V`.

B. Threat Model

To demonstrate the potential security threat of hidden events, we attempt to construct side channels to recover data leaked by transient execution using the method described in Section V-A. On the one hand, since `RDMSR` and `WRMSR` are privileged instructions, this makes the PMU side channel only available to privileged attackers. Thus in our threat model, the victim is the data in the Intel SGX Enclave [23]. When the victim is in a trusted execution environment (TEE), it is reasonable

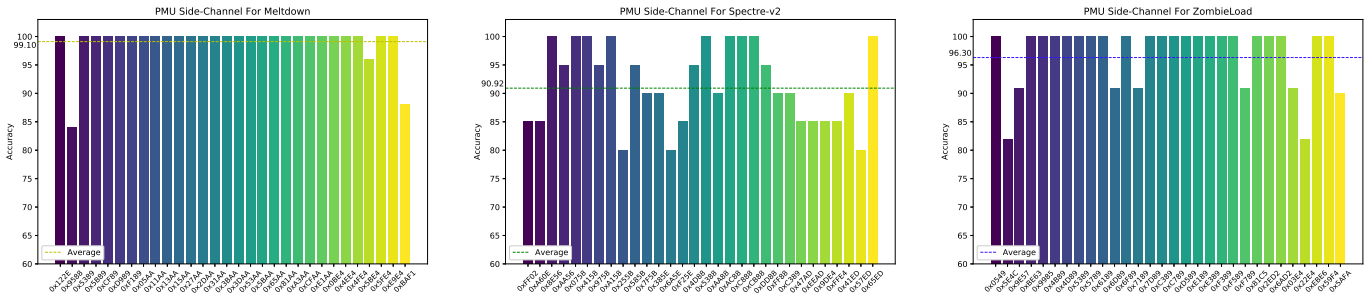

Fig. 4: The Accuracy of Recovering Leaked Data from Transient Execution Vulnerabilities Using Hidden PMU Build Side Channels.

to assume that the attacker has root privileges [23], [30], [40]. Furthermore, there are many legitimate and illegitimate ways for an attacker to gain this privilege [48], [49]. To this end, we reproduced the Zombieload [30] attack to steal data from SGX. On the other hand, we also reproduced the Meltdown and Spectre attacks, even though the attacker did not have root privileges in their threat models. However, to demonstrate the security threat of hidden PMU events, we take them into account as well. Here we only read and write PMUs with root privileges and do nothing to favor the attacker.

C. Attack Experiment Setup

We selected an i7-6700 (Skylake) processor with 32 KiB, 8-way L1 data cache, and SGX Extension as the victim device for our experiments. Using the instruction gadget in Listing 1 we successfully replicated three transient execution attacks: meltdown [27], spectre_v2 [29], and zombieload [30], on this device. However, we were unsuccessful in trying to reproduce spectre_v1 [29]. A preliminary analysis suggests that the presence of conditional branching instructions in the gadget of Listing 1 may interfere with branch mistraining, which is a crucial component of the spectre_v1.

It's worth noting that for the PMU Side Channel, iterating through all the events and associated instructions is theoretically the best approach. However, based on the data in Table.I of Section III, even if we set `ins2` to a `NOP` instruction, this would require approximately $3484 * 20599 \approx 7.1 * 10^7$ iterations with an average time of 0.4 seconds per iteration, which would take about a year to complete. This is not an acceptable timeframe. As an alternative, we attempt to record the specific instructions that trigger a PMU event during the collection process in Section III. We can significantly reduce the iteration space to approximately 11 million iterations by focusing on this subset of instructions. However, even with this optimization, it would still take more than 50 days to complete the traversal. To expedite the traversal process, we decide to set `ins1` as one or more memory access instructions. By focusing on this specific type of instruction, we can limit the traversal to only $20,599$ PMU events. As a result, the overall traversal time is reduced to approximately one hour. While this approach may sacrifice some precision, it still effectively demonstrates the potential security threat posed by these hidden PMU events.

D. Experiment Results

$Throughput$ and $Accuracy$ are two significant metrics used to evaluate side-channel attacks. The $Throughput$ is primarily influenced by factors such as the execution time of the instruction gadget, the number of iterations, the exception handling time, or the branch training time. In our experimental setup, to mitigate the effects caused by system noise and non-deterministic of PMU, for every byte of the secret data, we perform the attack 10 rounds. For the meltdown attack with PMU Side-Channel on i7-6700, the average $Throughput$ can reach 2.7 KB/s when utilizing Intel TSX [43] exception suppression. For the spectre_v2 attack, the average $Throughput$ is approximately 0.6 KB/s because the branch training takes are longer than the exception processing. For the zombieload attack, the average $Throughput$ is 2.4 KB/s when using the exception handler. If TSX exception suppression is used, it can be increased to 12.2 KB/s.

$Accuracy$ is another significant metric used to evaluate the effectiveness of an attack. In our case, the $Accuracy$ is determined by the individual PMU events. To calculate the $Accuracy$, we iterated through 20,599 hidden PMU events and filtered out the event numbers with $Accuracy \geq 80\%$. We then computed the average $Accuracy$ based on these selected events. For meltdown attacks, a total of 357 events out of 20,599 hidden PMU events had an $Accuracy \geq 80\%$. And their average $Accuracy$ is 99.10%. For spectre_v2 attacks, 1,094 events satisfy the criteria with an average $Accuracy$ of 90.92%. For zombieload attacks on SGX, 139 events met the criteria and the average $Accuracy$ is 96.30%. To provide a visual representation, we randomly sampled 30 samples for every type of attack and displayed them in Fig. 4.

E. Mitigations

In this section, we will present some possible mitigations and recommendations for constructing side-channel attacks with hidden PMU events.

a) Hardware-based Mitigations: An obvious approach would be to disable the PMU module altogether, but this would make it extremely difficult for software optimizers. Therefore, our recommendation is to change the counting behavior of the PMU, e.g., by clearing out the counts within the transient window, or by recording only the behavior on the CPU architecture. However, this would make the behavior within

the CPU microarchitecture invisible, which would complicate the cause analysis of various microarchitectural vulnerabilities.

b) Software-based Mitigations: On the software side, since PMU side-channel attacks target data in the Intel SGX, they can be protected by strengthening the SGX security domain. A privilege bit can be added via a microcode update to prohibit OS-level processes from accessing PMU counts triggered within the TEE. This may increase the performance overhead inside the TEE but will have less of an impact on performance at other privilege levels.

VI. Discussion & Future Work

In this paper, we designed a hidden PMU events collection method on Intel CPUs. And we demonstrate their exploitation potential and security threats by using these hidden events for transient execution attack detection and side-channel attacks. At the same time, some limitations need to be discussed.

A. Limitations

First, although this paper collects hidden PMU events, it does not provide a detailed analysis of their corresponding microarchitectural behaviors. Understanding the exact nature and purpose of these hidden events could shed light on their potential applications in various domains and unveil additional security threats. The paper primarily focuses on transient execution attack detection and side channel attacks as applications of hidden PMU events. However, it is worth investigating whether these events have broader applications or implications in other domains, such as performance analysis, energy optimization, or hardware security. These are the limitations that we believe exist in this paper and the possible extension of the work in the future.

B. Future Extensions

a) Hidden PMUs On Other Vendors' CPUs: This paper focused on exploring and exploiting the hidden PMU events present in Intel CPUs. However, during our initial experiments, we did come across undocumented PMU events on AMD and ARM machines as well. Although we did not extensively validate these events' validity, it suggests the possibility of hidden events existing in CPUs from other vendors. Therefore, in future work, it would be possible to explore the hidden events in CPUs from various manufacturers to gain a comprehensive understanding of their presence and potential security implications across different hardware platforms.

b) Reverse-Engineering: In this study, we discovered a significant number of hidden PMU events and established their associations with various microarchitectural attacks. But we did not conduct a detailed analysis of the specific behaviors exhibited by these instructions. As we mentioned in Section I, a PMU event is represented by 16 bits. In the collection of events, we observed that most of the possible values appeared in the higher 8 bits (UMask), while the lower 8 bits (Event Select) were confined to a fixed range, which determines the general class of events. This observation leads us to believe that it is feasible and necessary to reverse-engineer the hidden Event Select code in future research endeavors. By uncovering and understanding the microarchitectural behaviors associated with these hidden PMU events, we can more effectively exploit their capabilities and gain valuable insights into their potential security threats.

c) Specific bit decision UMask: As previously mentioned, we observed that the change in UMask value has a certain regularity. However, we do not consider each UMask value to represent a distinct event condition. Instead, we suspect that the CPU selectively examines a specific bit of the UMask when checking the event number. Our analysis briefly explored the distribution pattern of UMask in Section III-D, but further investigation was not conducted. Further research is required to explore the underlying principles of UMask selection. Indeed, investigating the selection principles of UMask is crucial for reverse engineering the hidden events.

VII. Conclusion

In this paper, we discovered a significant number of hidden PMU events on different microarchitecture CPUs. Building upon this finding, we have leveraged these hidden PMU events to achieve the detection of various transient execution attacks and to construct side channels by encoding private data into PMU events during transient execution.

Our experiments have demonstrated that hidden PMU events possess substantial exploitation potential and pose potential security threats. Furthermore, at the end of our paper, we analyzed the limitations inherent in our study and discussed future research and exploration of hidden PMU events. In summary, our research reveals the existence of undocumented PMU events and showcases their potential for exploitation space and security threats.

Acknowledgment

We thank the anonymous reviewers for their constructive feedback. This work was supported in part by the NSFC General Technology Fundamental Research Joint Fund (Grant No. U1836215), National Natural Science Foundation of China (Grant No. 62072263), Tsinghua University Initiative Scientific Research Program, BUPT Innovation and entrepreneurship support program (2023-YC-A163), and Zhongguancun Laboratory.

References

[1] AMD, "Processor programming reference (ppr) for amd," May. 2023, https://www.amd.com/en/support/tech-docs.

[2] ARM, "Arm neoverse v2 pmu guide," Apr. 2023, https://developer.arm.com/documentation/PJDOC-1063724031-660094/1/.

[3] "Intel 64 and ia-32 architectures software developer's manual: Volume 3," https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html.

[4] Intel, "Intel perfmon," Oct. 2022, https://github.com/intel/perfmon.

[5] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH computer architecture news*, vol. 41, no. 3, pp. 559–570, 2013.

[6] X. Wang and R. Karri, "Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 1–7.

[7] Li, Congmiao and Gaudiot, Jean-Luc, "Detecting spectre attacks using hardware performance counters," *IEEE Transactions on Computers*, vol. 71, no. 6, pp. 1320–1331, 2021.

[8] J. Cho, T. Kim, S. Kim, M. Im, T. Kim, and Y. Shin, "Real-time detection for cache side channel attack using performance counter monitor," *Applied Sciences*, vol. 10, no. 3, p. 984, 2020.

[9] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2016, pp. 406–418.

[10] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*. Springer, 2015, pp. 48–65.

[11] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1852–1867.

[12] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 88–105.

[13] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: a tool to analyze speculative execution attacks and mitigations," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 747–761.

[14] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[15] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd international workshop on performance analysis of workload optimized systems, FastPath*, vol. 13, 2013, p. 5.

[16] Intel, "Intel vtune profiler," 2023, https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[17] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.

[18] Intel, "Top-down microarchitecture analysis method," Dec. 2022, https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html.

[19] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, 2017, pp. 1–6.

[20] S. Bhattacharya and D. Mukhopadhyay, "Who watches the watchmen?: Utilizing performance monitors for compromising keys of rsa on intel platforms," in *Cryptographic Hardware and Embedded Systems–CHES 2015: 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings 17*. Springer, 2015, pp. 248–266.

[21] B. Asvija, R. Eswari, and M. Bijoy, "Template attacks on ecc implementations using performance counters in cpu," *Microelectronics Journal*, vol. 106, p. 104935, 2020.

[22] P. Qiu, Y. Lyu, H. Wang, D. Wang, C. Liu, Q. Gao, C. Wang, R. Sun, and G. Qu, "Pmuspill: The counters in performance monitor unit that leak sgx-protected secrets," *arXiv preprint arXiv:2207.11689*, 2022.

[23] S. D. Victor Costan, "Intel sgx explained," 2016, https://eprint.iacr.org/2016/086.pdf.

[24] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Binoculars:{Contention-Based}{Side-Channel} attacks exploiting the page walker," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 699–716.

[25] N. Gregory, H. Kannan, R. Harang, and E. Rudd, "Using undocumented hardware performance counters to detect spectre-style attacks," 2021.

[26] A. Abel and J. Reineke, "uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 673–686.

[27] 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 user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[28] J. Horn, "Speculative execution, variant 4: Speculative store bypass, 2018," *URl: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528*, 2018.

[29] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[30] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[31] Intel, "Intel® performance counter monitor - a better way to measure cpu utilization," Nov. 2022, https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html.

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

[33] S. Deng, B. Huang, and J. Szefer, "Leaky frontends: Security vulnerabilities in processor frontends," 2022.

[34] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium*, 2014. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

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

[36] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, pp. 99–112, 2017.

[37] B. Gras, K. Razavi, H. Bos, C. Giuffrida *et al.*, "Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks." in *USENIX Security Symposium*, vol. 216, 2018.

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

[39] M. Taram, X. Ren, A. Venkat, and D. Tullsen, "{SecSMT}: Securing {SMT} processors against {Contention-Based} covert channels," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3165–3182.

[40] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution," in *Proceedings fo the 27th USENIX Security Symposium*. USENIX Association, 2018.

[41] J. Van Bulck, F. Piessens, and R. Strackx, "Sgx-step: A practical attack framework for precise enclave execution control," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017, pp. 1–6.

[42] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 215–224.

[43] Intel, "Intel® transactional synchronization extensions (intel® tsx) overview," Aug. 2021, https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/tsx-programming-considerations-ov.html.

[44] C. Li and J.-L. Gaudiot, "Detecting malicious attacks exploiting hardware vulnerabilities using performance counters," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2019, pp. 588–597.

[45] T. M. Mitchell *et al.*, *Machine learning*. McGraw-hill New York, 2007, vol. 1.

[46] B. Schölkopf, A. J. Smola, F. Bach *et al.*, *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[47] N. Cristianini, J. Shawe-Taylor *et al.*, *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[48] S. Niu, J. Mo, Z. Zhang, and Z. Lv, "Overview of linux vulnerabilities," in *2nd International Conference on Soft Computing in Information Communication Technology*. Atlantis Press, 2014, pp. 225–228.

[49] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-vm row hammer attacks and privilege escalation." in *USENIX Security Symposium*, 2016, pp. 19–35.