Exploration and Exploitation of Hidden PMU Events

Yihao Yang¹, Pengfei Qiu¹, Chunlu Wang², Yu Jin³,

Dongsheng Wang⁴, Gang Qu⁵

^{1,2,3}Key Laboratory of Trustworthy Distributed Computing and Service (BUPT), Ministry of Education

⁴Tsinghua University ⁵University of Maryland

khaosyg@gmail.com, {qpf,wangcl}@bupt.edu.cn,lambda.jinyu@gmail.com,

wds@tsinghua.edu.cn, gangqu@umd.edu

Abstract

Performance Monitoring Unit (PMU) is a common hardware module in Intel CPUs. It can be used to record various CPU behaviors therefore it is often used for performance analysis and optimization. Of the 65536 event spaces, Intel has officially published only 200 or so.

In this paper, we design a hidden PMU event collection method. And we found a large number of undocumented PMU events in CPUs of Skylake, Kabylake, and Alderlake microarchitectures. We further demonstrate the existence of these events by using them for transient execution attack detection and build-side channel attacks. This also implies that these hidden PMU events have huge exploitation potential and security threats.

1 Introduction

Hardware Performance Counter (HPC) is a popular hardware monitoring tool in today's computer architectures. It has been widely used for more than a decade, and these counters can be used to measure events at the CPU level at a granular level, such as instruction execution, Cache Hit or Miss, branch prediction, etc. HPC is very important for performance analysis, code debugging, and optimization. Most modern processor vendors provide HPC support for their processors. In Intel processors, the functional unit used to support HPC is called the Performance Monitor Unit(PMU) [10].

The Intel official documented more than 200 PMU Events [8](which vary slightly on different architectures) for developers to use. These events are introduced initially for code debugging and performance improvement. However, because it can measure various microarchitectural events at a finegrained level, it has been widely used in various fields, such as malware detection and defense[4, 21], microarchitectural attack detection[12, 13], reverse engineering [16], and so on. In addition, some evaluation tools have been designed using PMU for different environment settings, such as PAPI[17], perf_event[22],and VTune[9]. These tools also greatly facilitate software developers to analyze the performance of their code. In addition to being used for positive work, Qiu et al.[18] found that PMU counters record behaviors of transient instructions besides those of truly committed instructions. They exploited this feature to create a PMU side channel that replicated the Foreshadow attack and compromised the security of Intel SGX[20].

Intel provides 16 bits for PMU event selection[10], as shown in Figure 1. The 8 bits Umask and 8 bits Event Select respectively make up a complete PMU event. This means that the entire event selection space is 2^{16} , but even on Intel's latest Alderlake architecture CPUs, only over 200 publicly available Core PMU events exist^[8]. This is a very small subset of the entire event space, meaning there may be many undocumented PMU events. Zhao et al.[26], in their performance analysis and reverse engineering of their work, mentioned two unrecorded PMU events, which they named L1D.READ REQS and L1D BLOCKS.FALSE DEPS based on their event behavior. However, they did not go further. Nick Gregory et al.[6] traversed the 2¹⁶ space and filtered for Spectre-sensitive events, and they eventually came up with 81 unrecorded PMU events. But in our work, we traversed the x86 instruction set and recorded the PMU events they triggered. In the end, tens of thousands of undocumented PMU events were found on different microarchitectures, which is far more than 81.

In this paper, we design a hidden PMU event collection method and we use the hidden PMU events for transient execution attack detection and construction of side-channel attacks. First, we traverse the x86 instructions using the uops.info[1] dataset. At the same time, the monitor program continuously polls the entire PMU event space to collect all possible PMU Events. We found about 20,000 undocumented PMU Events on the i7-6700, and i7-7700 and about 12,000 on the i9-13900k. Then, we try to perform transient attack detection with each PMU event, including Meltdown, Spectre, and Zombieload. Finally, we further demonstrate the effectiveness of these events by constructing side channels for each hidden PMU event and reproducing the above transient execution attacks[7, 11, 14, 19].



Figure 1: Layout of IA32_PERFEVTSELx MSRs

In general, this paper has the following contributions:

- We have designed a method to collect hidden PMU events. And we found a lot of hidden PMU events on Skylake, Kabylake, and Alderlake.
- We performed detection screening for hidden PMU events. We found 377 hidden PMU events that can be used for Meltdown-Detection, 6346 for Spectre-Detection, and 3837 for Zombieload-Detection.
- We use these hidden PMU events to build side channels for microarchitecture attacks to further demonstrate their effectiveness. We find 357 hidden PMU events that can be used to build side channels for Meltdown attacks, and 1094 for Spectre attacks.

These hidden PMU events, similarly, can record some microarchitectural behavior. They can also be used for malware detection, reverse engineering, or even for attack scenarios. Therefore it is important to prove the validity of undocumented PMU events and to assess their hidden security risks.

2 Background

2.1 Performance Monitor Unit

The Performance Monitoring Unit is an important hardware module on today's processors. It contains a set of performance counters that record various hardware performance events that occur at the CPU level during system runtime. Intel divides the hardware events supported by its performance counters into architectural performance events and non-architectural performance events, which also serve as microarchitectural events[10]. Architectural performance events refer to events that have consistent behavior across processor architectures, such as Instruction retired, Unhalted core cycles, Branch instructions, etc. Non-architectural performance events are processor microarchitecture specific and have different behavior across different microarchitectures and may vary with processor enhancements. For non-architectural performance events, they are further classified as Core Events, and Uncore Events[10]. Core Events are defined as performance events that occur inside the CPU, such as Instruction retired, Cache hit or miss, branch prediction, etc. Uncore Events are events that occur in components outside the CPU core, such as memory accesses, I/O operations, and so on. In this article, we will only discuss Core Events.

Intel provides users with three fixed counters and four programmable counters[3, 10]. The fixed counters always monitor fixed events such as logical cycles, reference cycles, etc. The programmable counters are supported by a set of one-toone event selection MSRs (IA32_PERFEVTSELx) 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. Intel provides two ways to get the value of the performance counter: Polling or Processor Event-Based Sampling (PEBS)[3, 10].

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

PEBS: This is a sampling method based on the Performance Monitoring Interrupt(PMI) interrupt. 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.

2.2 Side Channel Attacks

Side Channel Attacks: There are many shared resources in the microarchitecture, such as Cache, TLB, execution ports, etc. The attacker accesses the victim's information by monitoring the state changes of such shared resources. By Side Channel, the attacker does not directly attack the target data but infer the secret information such as the victim encryption key by analyzing the side information (e.g., voltage frequency change, cache timing, etc.) that the microarchitecture inadvertently leaks.

In microarchitecture, the most common ones are Cache side channel attacks, such as Flush+Reload[24], Prime+Probe[15], CacheBleed[25], etc. There are also side-channel attacks based on other shared resources, such as TLBLeed[5], PortSmash[2], Binoculars[26] etc. The basic principle of these attacks mostly relies on cache time differences. Qiu et al.[18] established a PMU-based side channel. the PMU captures and records various microarchitectural states, so the victim information can be inferred by analyzing the PMU event counts.

2.3 Transietn Execution Attacks

Transient execution attacks are caused by various aggressive optimization strategies introduced by modern processors to improve performance, such as Out-of-Order Execution, Branch Prediction, etc. These strategies may lead to the execution of instructions that should not be executed, which is called transient execution. Although transient instructions are not explicitly committed, they may have some impact on the microarchitecture state. The attacker captures such microarchitecture state changes by establishing side channels and thus inferring the victim's private data. Typical transient execution attacks are Meltdown[14], Spectre[7, 11], Zombieload[19], etc.

3 Hidden PMU Collector

3.1 Motivation

As we described in Section 1, PMUs can capture specified types of CPU hardware events to enable developers to optimize their code by understanding the system's runtime characteristics and performance bottlenecks. Today PMUs are widely used in various work scenarios. However, these events represent only a small fraction of the overall event space, and it is worth investigating whether most of the undocumented PMU events can also be used in these scenarios. In addition, PMU is also useful for all kinds of reverse engineering, and it is worthwhile to pay attention to whether Intel's undisclosed PMU implies the existence of some unrevealed CPU hardware components.

In addition, there are also some security risks because of the granularity of PMUs. An attacker could detect processor data and instruction flows through the PMU, or create side channels to leak information. Therefore, for a large number of unknown PMUs, their security risks may be even more threatening. Therefore, it is necessary to dig and analyze the hidden PMU events both in terms of positive and negative work.

3.2 Challenges

x86 Instructions Traversal: In order to collect hidden PMU events, we tried to execute all x86 instructions to trigger as many PMU events as possible. However, the complexity of the x86 instructions posed a significant challenge to us. We have compiled 5492 instructions based on the uops.info[1] dataset, which have different behaviors depending on the processor mode and privilege level. In addition, various jump instructions may cause the program to dead-end or terminate requiring special handling as well. It is worth noting that x86 has evolved with many instruction set extensions requiring specific floating-point units and registers, which different CPUs may support differently.

Secondly, the Intel assembly syntax is different from the GCC inline assembly syntax (AT&T), so we also need to preprocess these instructions. As well, there are multiple types of operands for the same instruction, which in turn increases the complexity of the entire instruction set. Overall, the complexity and diversity of the x86 instruction set have caused a great many problems for us.

Non-deterministic: As we explained in Section 2, the PMU can monitor various CPU-level events at a fine-grained level. Weaver et al.[23] show that PMU counting is inherently non-deterministic and over-counting, due to its architectural design. Such uncertainty makes it difficult to determine the validity of hidden PMU events with counts close to zero when collecting PMU events.

For such non-deterministic, Das et al.[3] point out that not all PMU-based work is affected. Among them, malware defense and detection works are more likely to be affected. This is because they rely on the small impact of the attack model on the hardware to determine whether it is being attacked. So, we filter this uncertainty by constructing microarchitectural attack detection models and side channel models with these hidden PMU Events.

3.3 Hidden PMU Collect

First, we process the uops.info data set. Because the Intel x86 assembly syntax differs from the GCC inline assembly syntax (AT&T) in some ways, such as the location of operands, and the register representation. At the same time, we try to keep the registers used by these instructions in a limited range as much as possible, which is convenient for us to fill the operands. It is notable that for some specific extensions, specific registers may need to be used. For this reason, we need to adapt them according to the instruction extensions supported by the CPU. Furthermore, for various jump instructions, we must put the jump target position after the jump instruction, otherwise, it may cause the program to enter a dead loop. Finally, we compiled a list of 5488 instructions.

Since we don't know the details of the instruction execution, we need to handle all the exceptions that may arise during the instruction execution. The best way to achieve this is to use Intel Transactional Synchronization Extensions (TSX) to suppress exceptions, which is fast and efficient. Unfortunately, because the success rate of transient execution attacks can be greatly improved with TSXs, many new Intel CPUs do not support this extension. Therefore, we bind all exception signals to custom exception handlers to prevent program crashes. We then fill a limited number of registers with the appropriate values or addresses to adapt the operand types of the instructions.

Finally, we monitor the count changes in the 65536 event space before and after each instruction execution and record the readable events and the instructions that triggered them.



Table 1: Hidden PMU Collector Result

Figure 2: Distribution of Umaks and EventCode for Hidden PMU Events on different Microarchitectures

3.4 Result Analysis

We separately performed collection experiments on three machines, as shown in Table 1. Finally, we successfully executed 3412 instructions on the i7-6700 (Skylake) and collected 20599 hidden PMU events. On the i7-7700 (Kabylake), we successfully executed 3574 instructions and collected 20230 hidden PMU events. On the i9-13900k (Alderlake), 3628 instructions were successfully executed and 12503 hidden PMU events were collected.

However, we do not think that each of these PMU events corresponds to a microarchitectural behavior. For the Event-Code, we found that it is not continuous, this may mean that these events actually exist. In the case of Umask, its distribution makes us wonder if the bit at the specified location determines the event selection condition. As we can see in Figure 2, the distribution of Umask has a segmented regularity in either microarchitecture. Moreover, some Umask values appear to be invalid under the three microarchitectures mentioned above. As an example, the hidden PMU event with EventCode is 0x6C in the i7-6700 shows a certain regular increase in the graph. On further analysis, we find that its Umask values appear to grow as 0x*1, 0x*3, 0x*5, 0x*7, 0x*9, 0x*B, 0x*D, 0x*F. In binary perspective, the lowest bit of their Umask is 1. So we suspect that the value of Umask may be determined by a specific bit.

4 Application 1:Detecting the Transient Execution Attacks

To further demonstrate the effectiveness of hidden events, we try to use these hidden events to detect existing transient execution attacks, such as Meltdown[14], Spectre[7, 11], Zombieload[19], etc.

4.1 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 past transient execution detection approaches[12, 13]. Instead, we must iterate through each hidden PMU event and monitor their association with these transient execution attacks. To do so, for each attack, we need to collect the count changes for each PMU event in the Clean, No-Attack, and Attack states. The classifier is then trained offline by a machine learning (ML) algorithm and then analyzes the model training results. In this way, we can determine whether that PMU event can be used for that transient execution attack detection.

4.2 Detection Experiment Setup

Data Collection: Because most of the transient execution attacks have been fixed on the i9-13900k. So for each hidden event, we collect the count of Clean, No-Attack, and Attack on the i7-6700. The Clean refers to a clean environ-



Figure 3: Transient Execution Attack Detection Model Evaluation

ment where only the victim process is running, to simulate an environment where no malicious attacks exist. In addition, we include text reading and writing to simulate the impact of other third-party processes on PMU counts. 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 collect 7 transient execution attacks, namely spectre_v1[11], spectre_v2[11], meltdown(spectre_v3)[14], spectre_v4[7], zombieload_v1[19], and zombieload_v2[19]. Finally, to better distinguish the false positives (FP), we need to collect data in the No-Attack environment. In the No-Attack environment, we comment out the Attack Primitive of the attacker process, leaving the rest of the code untouched, and collect the data according to the settings in the Attack environment. It adds false positive(FP) noise to our model so that we can better distinguish it.

Data Processing & Model Training: There are many machine learning models used for classification, such as logistic regression (LR), support vector machine (SVM), etc. Since we need to detect transient execution attacks with 20,000 hidden PMU events respectively, we choose the logistic regression algorithm with less training time. LR is a simple linear classification algorithm that estimates the probability of a given input based on a Sigmoid function. LR algorithm has fewer parameters and less training time compared to other classification algorithms. Moreover, other complex algorithms are theoretically possible if the simple LR algorithm can detect attacks.

Then, we label the data. PMU count changes collected in the Attack environment are labeled as 1, and the data in other environments are labeled as 0. This means that 1 indicates that an attack occurred and 0 indicates that no attack. For each type of attack, we collect data in each independent run and use the same number (2000) of samples from both categories to avoid bias. Then, we split the collected dataset into training data (70%) and test data (30%). Finally, we train the model with the training data and analyze it with the test data.

4.3 Experiment Analysis

In order to evaluate the model better, in addition to the most commonly used Accuracy metric, we also calculated other metrics of the detection model, including Precision, Recall, F1-Score, and AUC (Area Under Curve). Precision represents the proportion of true positive (TP) predicted to be positive, i.e. TP / (TP + FP), which is indicative of false positives (FP). Recall represents the proportion of positive samples predicted to be positive, i.e. TP / (TP + FN). F1-Score is the average of the two, which takes into consideration both Precision and Recall. The AUC represents the area under the ROC (Receiver Operating Characteristic) curve. The ROC curve shows the relationship between Recall and FP Rate, and the AUC is used to measure how well the detection model is able to distinguish between malicious and normal executions. In general, the closer the AUC is to 1, the more effective the model is.

We screened the detection models with *Accuracy* > 0.8, F1 > 0.8, AUC > 0.7 and their PMU events Number, and then randomly sampled 400 points to draw a scatter plot as

Figure 3. To prevent model overfitting, we removed the points with index values equal to 1. We also filter out the points with $F1 \in (0.9, 1)$ to consider *Precision* and *Recall*. Finally, we got 377 hidden PMU events available for meltdown detection, 530 for spectre_v1 detection, 4230 for spectre_v2, 1586 for spectre_v4, 1823 for zombieload_v1, and 2014 for zombieload_v2.

1	zero_pmu();
2	<pre>if (xbegin() == (~0u)) {</pre>
3	asm volatile(
4	"cmp (%0), %1"
5	"jz equal"
6	"nop"
7	"jmp end"
8	"equal:"
9	" insl (eg. movq (%%rax),%%rax)"
10	"end:"
11	" ins2 "
12	:
13	:"r"(The address of Secret),
14	"r"(Controllable Variable)
15	:
16);
17	xend();
18	}
19	read_pmu();

Figure 4: Encoding Secret into PMU Side-Channel.

5 Application 2:Implementing the Side Channel Attacks

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

5.1 Encoding The Secret Data into PMU

Qiu et al.[18] found that some instructions executed in transient windows also affect the PMU count. Based on this principle they designed an instruction gadget that encodes secret data into the PMU side channel, as in List1. First, the side channel state is cleared, as in the first line of Fig.4.3. Then, in the fourth line, we compare the secret data with a controllable variable V, which triggers a transient execution. If the secret data is equal to V, the path of command execution changes, i.e. ins1 is executed. And ins1 is bound to the PMU event we set, which allows us to infer whether ins1 is executed or not from the change in the PMU count, and thus secret data from the controllable variable V.

5.2 The Experiment Setup

The victim device we chose is also an Intel i7-6700 (Skylake) processor with 32 KiB, 8-way L1 data Cache, on Ubuntu 16.04 with kernel version 4.15.0. We successfully reproduced two transient execution attacks, Meltdown and Spectre_v2, using the above instruction gadget. We also tried the same for Spectre_v1, but it did not work. A brief analysis of the reason for this may be the existence of branching instructions in our gadget, which we suspect may affect branch mistraining.

Theoretically, the best way is to perform a combined traversal of instruction and PMU Events. However, even if we set ins2 to nop, this would require $5492 * 20599 \approx 1.13 * 10^8$ iterations with an average time of 0.4s per iteration, which would take about a year, and this is not acceptable. In addition, we try to record the corresponding instructions that trigger a PMU in the Collector. Then we only traverse the combination of these, which can reduce the iteration space to about 11 million iterations, but this also takes more than 30 days.

So, we set insl to be a single access instruction, because according to the results in Collector, the access instruction can trigger the most PMUs. Then only 20,599 PMU Events are traversed, which eventually reduces the traversal time to about one hour. Although this loses some precision, it also demonstrates the potential security threat of these hidden PMU events.

5.3 Experiment Analysis

Throughput rate and error rate are two important measures to evaluate side-channel attacks. The throughput rate is mainly determined by the instruction gadget execution time, the number of iterations, the exception handling time, or the branch training time. On our experimental device (i7-6700), the instruction gadget was iterated 10 times to recover the secret data. For the meltdown attack, the average throughput rate can reach 789.86 Bps if Intel TSX exception suppression is used. if the exception signal processing function is used, the throughput rate drops to 497.49 Bps. while for spectre_v2, the average throughput rate is around 148.68 Bps because the branch training takes longer than the exception processing.

Another important metric is the error rate (or accuracy). Unlike the throughput rate, the accuracy of an attack is determined by the individual PMU events. Different PMU events have different Accuracy, so we iterate through 20,000 hidden PMU events and filter out the event numbers with *Accuracy* \geq 80, and then calculate their average accuracy. For demonstration purposes, we randomly sampled 100 samples and show them in Fig. 5. It can be seen that for the meltdown attack, the average error rate(1 – *Accuracy*) is 0.9% for 10 iterations and 9.08% for spectre_v2. Finally, we obtain 357 hidden PMU events that can be used to construct a side channel to recover the secret data leaked for the meltdown attack with *Accuracy* \geq 80, and 1094 for spectre_v2.



Figure 5: Accuracy of Hidden PMU Side-Channel Attack

6 Discussion & Future Work

6.1 Limitations

The first is that this paper only tries to explore hidden PMU events in three microarchitectures of Intel CPUs, so whether there are also hidden events in other microarchitectures or there are also undocumented PMU events in other processor vendors' CPUs. Second, this paper does not work reverse for all the hidden events, i.e., find their respective corresponding microarchitectural behaviors. Then, only two transient execution attacks are successfully reproduced using the hidden PMU side channels, and whether other transient execution attacks can also recover private data using hidden PMUs as side channels. Or are there any other security threats for these hidden PMUs other than as side channels? Also, whether these hidden PMU events correspond to some unknown hardware components in the microarchitecture. We leave these questions to be explored in the future.

6.2 Future Extensions

Reverse-Engineering: We have found a large number of hidden PMU events and associated these events with various microarchitecture attacks. Although we collected the events by associating the hidden events with the instructions that triggered them, we did not do further analysis of the specific behavior of the instructions. However, because of the enormous number of events, we were unable to match the event codes to the microarchitecture behaviors as Intel officially disclosed events. As we introduced before, a PMU event consists of 16 bits. For the high 8 bits (Umask) most of the possible values appear in our collection of events, while for the low 8 bits (Event Select) it is limited to a fixed range, which also determines the general class of events. Therefore, we believe that it is feasible and necessary to reverse the hidden Event Select Code in future work.

Specific bit decision Umask: As we mentioned above, Umask appears in most of the possible values of the events we collected. However, we do not believe that each Umask represents an event condition, so we suspect that the CPU only focuses on the specified bit of Umask when checking the event number. In this paper, although the pattern of Umask distribution was initially analyzed in Section 3.4, it was not further explored. Therefore, it is necessary to explore the principle of Umask selection in future work.

7 Conclusion

PMU were originally designed for software performance optimization, but because of their granularity of monitoring, they have been widely used in various scenarios. In this paper, we found a large number of undocumented PMU events in microarchitecture CPUs such as Skylake, Alderlake, etc. Based on this discovery, we use these hidden PMU events to accomplish the detection of various transient execution attacks, as well as the leakage by encoding private data into PMU events during transient execution to build side channels.

Our experiments show that these hidden PMU events do exist and can be used for positive or negative work. Also at the end of the paper, we analyze the limitations and possible extensions of this study. Future work on hidden PMU events is worth exploring.

References

 Andreas Abel and Jan 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.

- [2] Alejandro Cabrera Aldaya et al. "Port contention for fun and profit". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 870–887.
- [3] Sanjeev Das et al. "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.
- [4] John Demme et al. "On the feasibility of online malware detection with performance counters". In: ACM SIGARCH computer architecture news 41.3 (2013), pp. 559–570.
- [5] Ben Gras et al. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks." In: USENIX Security Symposium. Vol. 216. 2018.
- [6] Nick Gregory et al. "Using Undocumented Hardware Performance Counters to Detect Spectre-Style Attacks". In: (2021).
- [7] Jann Horn. "Speculative execution, variant 4: Speculative store bypass, 2018". In: URI: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 (2018).
- [8] Intel. Intel Perfmon. https://github.com/intel/ perfmon. Oct 2022.
- [9] Intel. Intel VTune Profiler. https://www.intel. com/content/www/us/en/developer/tools/ oneapi/vtune-profiler.html. 2023.
- [10] 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.
- [11] Paul Kocher et al. "Spectre attacks: Exploiting speculative execution". In: *Communications of the ACM* 63.7 (2020), pp. 93–101.
- [12] Congmiao Li and Jean-Luc 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.
- [13] Congmiao Li and Jean-Luc Gaudiot. "Detecting spectre attacks using hardware performance counters". In: *IEEE Transactions on Computers* 71.6 (2021), pp. 1320–1331.
- [14] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: 27th USENIX Security Symposium (USENIX Security 18). 2018.
- [15] Fangfei Liu et al. "Last-level cache side-channel attacks are practical". In: 2015 IEEE symposium on security and privacy. IEEE. 2015, pp. 605–622.

- [16] Clémentine Maurice et al. "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.
- [17] Philip J Mucci et al. "PAPI: A portable interface to hardware performance counters". In: *Proceedings of* the department of defense HPCMP users group conference. Vol. 710. 1999.
- [18] Pengfei Qiu et al. "PMUSpill: The Counters in Performance Monitor Unit that Leak SGX-Protected Secrets". In: *arXiv preprint arXiv:2207.11689* (2022).
- [19] Michael Schwarz et al. "ZombieLoad: Cross-privilegeboundary data sampling". In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019, pp. 753–768.
- [20] Srinivas Devadas Victor Costan. Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf. 2016.
- [21] Xueyang Wang and Ramesh 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.
- [22] Vincent M Weaver. "Linux perf_event features and overhead". In: *The 2nd international workshop on performance analysis of workload optimized systems, Fast-Path.* Vol. 13. 2013, p. 5.
- [23] Vincent M Weaver, Dan Terpstra, and Shirley 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.
- [24] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: 23rd USENIX Security Symposium. 2014. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/ conference / usenixsecurity14 / technical sessions/presentation/yarom.
- [25] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: a timing attack on OpenSSL constanttime RSA". In: *Journal of Cryptographic Engineering* 7 (2017), pp. 99–112.
- [26] Zirui Neil Zhao et al. "Binoculars:{Contention-Based}{Side-Channel} Attacks Exploiting the Page Walker". In: 31st USENIX Security Symposium (USENIX Security 22). 2022, pp. 699–716.