Using Undocumented Hardware Performance Counters to Detect Spectre-Style Attacks

Nick Gregory*[1] and Harini Kannan†[1]

[1]Sophos Ltd.

Abstract

In recent years, exploits like Spectre, Meltdown, Rowhammer, and Return Oriented Programming (ROP) have been detected using Hardware Performance Counters. But to date, only relatively simple and well-understood counters have been used, representing just a tiny fraction of the information we can glean from the system. What's worse, using only well-known counters as detectors for these attacks has a huge disadvantage - an attacker can easily bypass known counter-based detection techniques with minimal changes to existing sample exploit code. Uncovering the treasure trove of overlooked and undocumented counters is necessary if we are to both build defenses against these attacks and anticipate how an adversary could bypass our defenses.

In this paper, we'll first introduce our version of Spectre variant 4 with evasive changes that can bypass any detections using conventional cache miss, branch miss, and branch misprediction counters. We'll then show how our model using select undocumented counters is able to detect this new edited variant, and how it is also able to detect a novel Spectre implementation submitted to Virus Total.

1 Introduction

Performance Monitoring Counters (PMCs - also sometimes referred to as Hardware Performance Counters) are devices built-in to modern processors that can count the number of times specific events happen while a processor is executing with nearly zero overhead. On Linux systems, these counters can be made available to userland through the perf subsystem and CLI. This is very useful because properly programming the counters can require a lot of advanced knowledge of hardware internals and very low-level, microarchitecture-specific code. These have obvious potential in the security space and have been used in prior research to detect various attacks and exploit techniques[11], e.g., return oriented programming (ROP)[13] and speculative execution exploits[3].

However, prior work has focused on using only select well-documented, well-understood PMCs. In an effort to make a better detection, we approached this completely as an ML problem, creating models using any counters made available by the processor. We introduce two exercises as part of model creation:

In the first exercise, we collect all available performance counters (both documented and undocumented) when running a baseline program (scikit-learn benchmarks) which was the most common program to cause false positives among the models using well-known documented counters as features in our prior work[9]. We then select the best features based on similarity between the Spectre variants and the difference between the Spectre variants and the baseline program.

*Nick.Gregory@sophos.com

†Harini.Kannan@sophos.com

Proceedings of the Conference on Applied Machine Learning for Information Security, 2021

The second exercise involves a set of ML experiments which will show these selected counters being used as features in interpretable ML models, trained on existing Spectre and meltdown exploits and other benign programs and detecting all the Spectre and meltdown variants. The trained model successfully detects our Spectre variant with evasive changes and the Spectre attack seen by Virus Total as part of our holdout dataset.

The results seen here will emphasize the need for documenting these counters, which were highly significant in our models for attack detection.

2 Background

2.1 PMCs

Most (if not all) modern processors include subsystems (usually called Performance Monitoring Units or PMUs) for monitoring CPU performance by counting certain events within the CPU. These are primarily used for low-level software performance monitoring/tuning. The available counters usually include "simple" events, such as a single instruction executing, but may also contain highly specialized events emitted by microarchitecture-specific, implementation-specific subsystems.

On Intel CPUs, individual counters are identified by two 8-bit fields - an event identifier (also known as the event selector), and a unit mask (umask)[7]. In general, the event identifier selects a specific type of event, and the umask selects a filter to further narrow the counter. For example, on Intel Haswell-EP processors, event id 0xC4 denotes all `BR_INST_RETIRED` (branch instructions retired) events, and the 7 umask values filter the type of branch instructions that are counted (conditional branches, branches not taken, CALLs, etc.)[6]. However, these event identifiers and unit masks are not guaranteed to be the same across microarchitectures, even from the same CPU vendor. Documented counters will be often referred to by their name or type which is translated to the architecture-specific counter for the CPU being run on. Since we collect all counters regardless of whether they have a documented name however, the events and umasks that were used in our models may not behave the same on other CPU microarchitectures.

For our research, all counters measured were on the "core" PMU, however there are other PMUs (e.g. the "uncore" PMU on Intel CPUs) that could have been sampled from. These typically require further filtering or setup (by way of Model Specific Registers or MSRs), which is why we did not sample them.

2.2 Cache-based Side-channel Attacks

Since the first variants of Spectre were originally published in 2018[10], a significant amount of research has gone into finding new side-channels in modern, high-performance processors. A comprehensive overview of cache-based side-channel attacks is outside the realm of this paper, however to give some background, in its simplest form, the idea behind cache-based side-channel attacks is for an attacker to setup the cache lines of a processor corresponding to some secret data in some "known" state, let the target/victim program run which will modify the cache, then measure the time it takes to do some operation on the cache to determine if the target accessed a specific region of memory.

There are three common techniques: Flush+Reload, Flush+Flush, and Prime+Probe.

Flush+Reload is arguably the most common technique. It involves the attacker flushing the cache (usually via the CLFLUSH instruction on x86 processors), then reading back the memory in question after the victim has run measuring the time it takes to read each region. If the time is small enough, that implies the victim must have accessed the data (implicitly loading it into cache), and the attacker can use this knowledge to potentially leak some private state of the program.

Flush+Flush is a variant of this, but instead of reading the data back directly, it attempts to flush each cache line again. The time taken for the flush to succeed is then used as an oracle to determine if the data was present in cache.

Lastly, Prime+Probe fills the entire cache (instead of flushing it) and uses timing information to determine which (if any) cache line(s) was/were flushed/replaced when the victim ran, thereby potentially leaking what region of memory the victim was accessing.

3 Bypassing Existing Detections

Prior work to detect cache side channel attacks using PMCs has focused around hand-picking a specific set of counters (usually some count of cache misses and some count of program "complexity") and checking if the number of cache misses (or equivalent) per branch/instruction seems abnormal[3, 9, 4, 2]. This comparison could be a simple threshold, or a full-fledged ML model tracking the ratio(s) over time. This method has limitations, however. As noted by others[3], this technique can't detect Flush+Flush attacks, as no cache misses are ever directly incurred. More importantly, these techniques can be trivially bypassed by simply slowing down the side channel exploit, and interleaving "cache friendly" code in between exploit attempts.

Spectre style attacks typically require tens to hundreds of iterations to be run to leak a single byte reliably. The hundreds to thousands of total flushes, cache probing, etc. are what make existing detections work. Simply interleaving a few hundred thousand intentional cache reads between iterations is enough to defeat these detections, as this dramatically reduces the cache miss rate per instruction/branch.

To give a concrete example, applying the following modifications to a publicly available Spectre v4 proof-of-concept[5] makes the program's cache miss ratio nearly identical to "normal" programs:

- Create a large global array (we used `unsigned long long stuff[0x1000]`)
- In between Spectre iterations, apply the following (or any other code which does repeated cache reads/writes):

```
for (register int round = 0; round < 2000000; round++) {
    register unsigned long long *p = &stuff[round % sizeof(stuff)];
    ctr += *p;
    *p = ctr;
}
```

The above successfully bypasses the first detections we created when Spectre and Meltdown were initially revealed in early 2019 and should also bypass all detections based on measuring cache misses. In fact, by adjusting the round count above, an arbitrarily low cache miss ratio can be achieved with little runtime overhead - the exploit still completes in under a second. In the rest of this paper, we'll refer to this modified proof-of-concept with these modifications as the "ghosting" Spectre v4, as it's invisible to existing detections.

4 Experimental Setup

Data was collected on two computers powered by Intel processors. The first was an Intel Xeon E5-2667 v3 (Haswell-EP) based workstation. The second was a slightly older Intel Core i5-3210M (Ivy Bridge) based laptop. Both computers were running Ubuntu Linux. We chose to use two slightly different processors (one "generation" apart) to see if the counters identified were somewhat stable between microarchitectures.

On each machine, we collected the raw counter value of every possible (event id, umask) pair 10 times for each program under test. While Intel only documents hundreds to just over a thousand counters depending on the microarchitecture, we collected all 65536 possible counters.

The programs we collected counters for were:

- Scikit-learn benchmarks - to provide a CPU-heavy workload which had notably caused false positives in our prior Spectre detections
- Phoronix nginx tests - to provide a "real-world" workload
- Linux defconfig compile - to provide a kernel-heavy workload
- LibJIT unit tests - to provide a very cache-unfriendly workload
- Various public Spectre proof-of-concepts - to provide a baseline for what a cache side channel attack "looks" like
  - Meltdown (aka Spectre v3 - rogue data cache load)
  - Spectre v1 (bounds check bypass)
  - Spectre v2 (branch target injection)
  - Spectre v4 (speculative store bypass)
- Ghosting Spectre v4 proof-of-concept - to provide a comparison with the unmodified proof-of-concept to identify counters that were measuring the exploit itself
- "Spectre in the wild" - a working Spectre implementation for Linux found on Virus Total (see Section 9)

5 Initial Data Analysis and Results

Before training any models, we first performed some basic filtering on the datasets. All counters which were always zero were removed, simply to speed up further processing. Counters which exhibited an insignificant difference per instruction (<90% difference) between the compute-heavy scikit benchmark and the Spectre proof-of-concept were removed, as these would also necessarily be unhelpful in model creation. Lastly, counters which differed significantly (>5%) between the Spectre proof-of-concept and the modified proof-of-concept were removed, as these counters are antithetical to the goal of finding counters that are detecting Spectre itself. We were left with a total of 81 counters which fulfilled these criteria. Interestingly, none of them were officially documented.

6 Data Collection

The Intel CPUs being used support up to four arbitrary PMCs being collected at the same time. We selected two sets of three counters each for initial analysis. The rest of this paper only references results for the first dataset.

Dataset 1:

- `event_id=0xef,umask=0xf4`
- `event_id=0x4d,umask=0xe3`
- `event_id=0x36,umask=0x98`

Dataset 2:

- `event_id=0xef,umask=0xf4`
- `event_id=0x4d,umask=0xb1`
- `event_id=0xd5,umask=0xa6`

Data was collected for the programs mentioned in Section 4, with counters being measured every 100ms and each program run 5 times:

7 Dataset Summary

The data composition looked as follows (in %):

scikit benchmarks	72.07
phoronix nginx	24.46
libjit	0.032
meltdown	0.019
spectre1	0.33
spectre2	0.0063
spectre4	0.84

The train/test split was kept at 67/33 (%). The holdout dataset composition looked as follows (note that none of the programs in holdout dataset was used for training):

Linux defconfig compile	96.35
Ghosting spectre 4	1.01
Spectre in the wild	2.64

8 Models Performance and Metrics

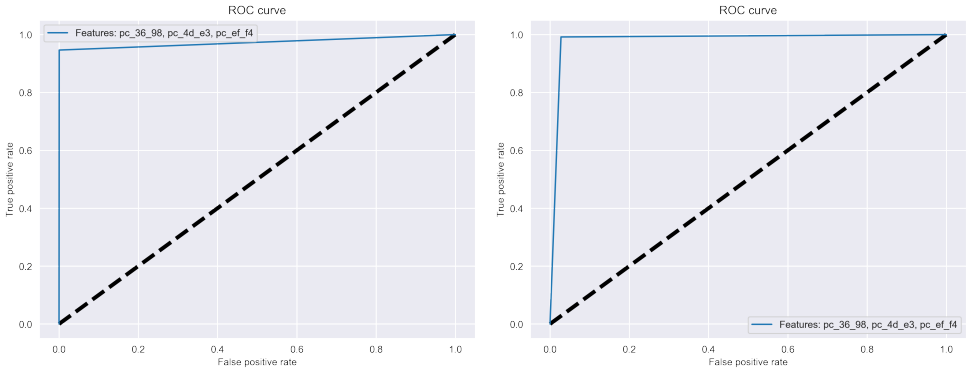
We trained 4 different models - Support Vector Machine with RBF kernel, Random Forest, XGBoost and Hist Gradient Boosting Classifier. For both datasets, XGBoost and Hist Gradient Boosting Classifier showed promising results. XGBoost showed 0.973 Area Under the Curve, with 0.05% False Positive Rate and 5.3% False Negative Rate for the test dataset. The holdout dataset had 0.982 AUC, 2.77% False Positive Rate and 0.82% False Negative Rate. The ROC curve and confusion matrices for the XGBoost model are shown in Figure 1 and Figure 2 respectively. Here, we are considering a data point as False Negative even if it's just a part of the exploit that's missed and the other parts are successfully detected. We have confirmed that all exploits tested here (including Ghosting Spectre v4 and the Spectre attack seen by Virus Total) are detected successfully.

Statistics from different models created for first set of features are shown in Table 1, and detection results for the models are shown in Table 2. Model statistic results for the second set of features are shown in Table 3, and detection results are shown in Table 4.

9 Detecting Spectre in the Wild

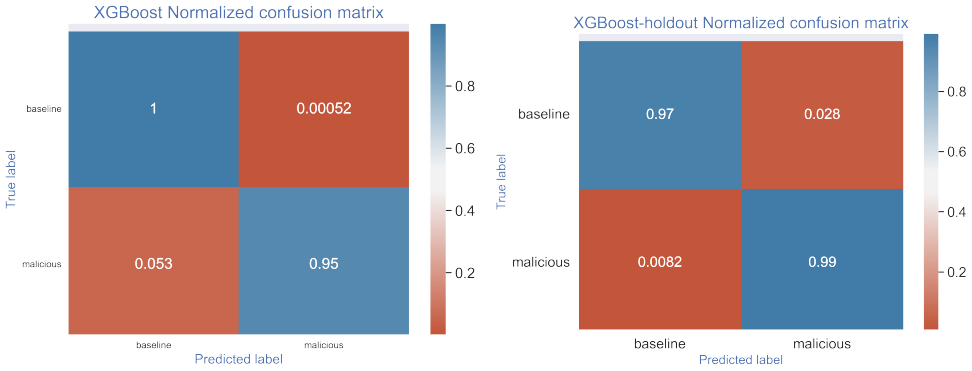
One of the (if not the) first instances of a Spectre-style attack being publicly available as part of an exploit was leaked as a module in Immunity Inc.'s CANVAS toolkit[1]. We performed some simple reverse-engineering of this in-the-wild exploit and found it had an apparent test flag, which could be provided an address to leak.

This exploit was discovered after our initial research was completed, and we tested our best model (XGBoost) blind against this attack. It was able to successfully classify the novel program as malicious. Even though this was done after our initial research, we include this program as part of our holdout dataset to calculate the final model metrics.



(a) Test Dataset (b) Hold-out Dataset

Figure 1: ROC curves for the XGBoost model



(a) Test Dataset (b) Hold-out Dataset

Figure 2: Normalized confusion matrices for the XGBoost model

10 Feature Interpretation

Given that all the best PMCs identified (and therefore all models created) were reliant on PMCs that are not documented by Intel, we are limited in how much we can interpret and explain the models created. With that caveat in mind, we did find some limited references for the 0x36 and 0xEF event ids, which may give some explanation for what these models are measuring.

10.1 0x36

While not present in the list of "core" PMCs, Haswell-EP documentation refers to the 0x36 event id in the uncore PMC listing as `UNC_C_TOR_OCCUPANCY` (Uncore Table of Requests Occupancy)[6]. However, the 0x98 umask is not documented, and the other counters in the 0x36 event selector all refer to a separate filter MSR (Model Specific Register) being set to further select the desired data. Given that the uncore is responsible for last level cache (LLC) coherence, and as such, would need to be informed of cache line flush requests, we believe we may be observing this uncore counter on the core PMU device, perhaps as a side-effect of the PMU implementation.

10.2 0xEF

Intel names the 0xEF event id as `CORE_SNOOP_RESPONSE` in Skylake-X and newer documentation, but the 0xF4 umask is not mentioned (and Skylake-X is also years newer than Haswell and Ivy Bridge, the microarchitectures of the chips data was collected on)[8]. However, others[12] have noted that this event id appears to be related to evictions processed by L1 and L2 caches, which

Arch	Model	Precision	Recall	FPR	FNR	AUC	Acc
ivybridge	SVM	1	0.85	0	0.3	0.85	0.99
ivybridge	XGBoost	0.98	0.94	0.0004	0.12	0.94	0.99
ivybridge	RF	1	0.86	0	0.28	0.86	0.99
ivybridge	HGBoost	0.98	0.94	0.0004	0.112	0.94	0.99
haswell	SVM	0.98	0.93	0.0005	0.13	0.94	0.99
haswell	XGBoost	0.99	0.98	0.0004	0.04	0.98	0.99
haswell	RF	1	0.97	0.0001	0.06	0.97	0.99
haswell	HGBoost	0.98	0.98	0.0008	0.04	0.98	0.99

Table 1: Model Results 36-98, 4D-E3, EF-F4

Arch	Model	Meltdown	Spectre1	Spectre2	Spectre4	Spectre4_New
ivybridge	SVM	N	N	N	Y	Y
ivybridge	XGBoost	Y	Y	Y	Y	Y
ivybridge	RF	Y	N	N	Y	Y
ivybridge	HGBoost	Y	Y	N	Y	Y
haswell	SVM	Y	N	N	Y	Y
haswell	XGBoost	Y	Y	Y	Y	Y
haswell	RF	Y	N	N	Y	Y
haswell	HGBoost	Y	Y	Y	Y	Y

Table 2: Detection Results 36-98, 4D-E3, EF-F4

of course would include the cache flush instructions that the Spectre exploit was using. Due to this, we believe this event id may have existed undocumented in microarchitectures prior to Skylake-X and believe that cache line snooping is the general event that the model is picking up on.

11 Conclusion

By approaching cache side channel exploit detection as a ML problem instead of an expert knowledge problem, we've shown that it's possible to create highly accurate models using un- or partially documented Performance Monitoring Counters (PMCs), none of which are things prior work have incorporated into detections. These improved models are resilient to trivial exploit modifications that break existing detections. While further data collection would be necessary to productionize these models (as the PMCs used may not exist or may have changed in any given processor generation), we have shown that it is possible to use machine learning to determine PMCs which may be a better signal of attacks than conventional wisdom/expert knowledge alone.

There is also still plenty more to explore. For instance, data from the "uncore" Performance Monitoring Units (PMU) on Intel processors could be collected and may provide even stronger signals to detect cache side channel attacks, due to the uncore being a key component of Intel processors' caches. With the recent rise in ARM processor usage in the datacenter, research is also necessary into what PMCs ARM CPUs offer. Due to manufacturer-specific changes, however, there may not be as much "consistency" in the quality of detection obtainable on ARM CPUs as there is on x86.

Arch	Model	Precision	Recall	FPR	FNR	AUC	Acc
ivybridge	SVM	0.99	0.81	0.0002	0.37	0.81	0.99
ivybridge	XGBoost	0.98	0.88	0.0005	0.25	0.88	0.99
ivybridge	RF	0.99	0.86	0.0002	0.28	0.86	0.99
ivybridge	HGBoost	0.98	0.87	0.0006	0.26	0.87	0.99
haswell	SVM	1	0.93	0.0001	0.13	0.93	0.99
haswell	XGBoost	0.99	0.97	0.0003	0.06	0.97	0.99
haswell	RF	0.99	0.95	0.0002	0.1	0.95	0.99
haswell	HGBoost	0.99	0.97	0.0002	0.056	0.97	0.99

Table 3: Model Results 4D-B1, D5-A6, EF-F4

Arch	Model	Meltdown	Spectre1	Spectre2	Spectre4	Spectre4_New
ivybridge	SVM	Y	N	N	Y	Y
ivybridge	XGBoost	Y	Y	Y	Y	Y
ivybridge	RF	Y	Y	N	Y	Y
ivybridge	HGBoost	Y	Y	Y	Y	Y
haswell	SVM	Y	N	N	Y	Y
haswell	XGBoost	Y	Y	N	Y	Y
haswell	RF	Y	N	N	Y	Y
haswell	HGBoost	Y	Y	Y	Y	Y

Table 4: Detection Results 4D-B1, D5-A6, EF-F4

References

[1] 2021. URL `https://www.virustotal.com/gui/file/6461d0988c835e91eb534757a9fa3ab35afe010bec7d5406d4dfb30ea767a62c/detection`.

[2] M.-M. Bazm, T. Sautereau, M. Lacoste, M. Sudholt, and J.-M. Menaud. Cache-based side-channel attacks detection through intel cache monitoring technology and hardware performance counters. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 7–12, 2018. doi: 10.1109/FMEC.2018.8364038.

[3] J. Cho, T. Kim, T. Kim, and Y. Shin. Real-time detection on cache side channel attacks using performance counter monitor. In *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 175–177, 2019. doi: 10.1109/ICTC46691.2019.8939797.

[4] D. Fiser and W. Gamazo Sanchez. Detecting attacks that exploit meltdown and spectre, 2018. URL `https://www.trendmicro.com/en_us/research/18/c/detecting-attacks-that-exploit-meltdown-and-spectre-with-performance-counters.html`.

[5] J. Horn. speculative execution, variant 4: speculative store bypass, 2018. URL `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`.

[6] Intel. Events for intel microarchitecture code name haswell x, . URL `https://perfmon-events.intel.com/haswell_server.html`.

[7] *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, 3C, and 3D: System Programming Guide*. Intel.

[8] Intel. Events for intel microarchitecture code name skylake-x, . URL `https://perfmon-events.intel.com/skylake_server.html`.

[9] H. Kannan and G. Williams. Spectre and meltdown | the data science approach, 2018. URL `https://info.capsule8.com/spectre-and-meltdown-the-data-science-approach`.

[10] 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 *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019. doi: 10.1109/SP.2019.00002.

[11] 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)*, volume 1, pages 588–597, 2019. doi: 10.1109/COMPSAC.2019.00090.

[12] J. D. McCalpin. Hpl and dgemm performance variability on the xeon platinum 8160 processor. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 225–237, 2018. doi: 10.1109/SC.2018.00021.

[13] X. Wang and J. Backer. Sigdrop: Signature-based rop detection using hardware performance counters, 2016.