Using Undocumented Hardware Performance Counters to Detect Spectre-Style Attacks

Nick Gregory

Harini Kannan

CAMLIS 2021

SOPHOS

Who We Are

SOPHOS

Nick Gregory

- Research Engineer @ Sophos
- Background in binary exploitation and low-level systems

Email: nick.gregory@sophos.com

Twitter: @kallsyms

Website: https://www.nickgregory.me

Harini Kannan

- Data Scientist @ Sophos
- Background in Business Statistics
- Currently area of interests:
  - System user behavior profiling
  - Interpretable ML
  - Command line language modeling
  - MLOps

Twitter: @jarvision__

Website: https://harini.blog/

Introduction

Can we detect exploits using undocumented hardware performance counters on Intel CPUs?

SOPHOS

Hardware Performance Counters

- A.k.a. Performance Monitoring Counters
- Hardware devices that count specific events across different Performance Monitoring Units (PMUs)
- Usually used to debug program/system slowness
  - Measuring things like cache misses, branch mispredicts, port usage, etc.

A Couple of Years Ago...

SOPHOS

Background: Spectre and Meltdown

- CPU-level vulnerabilities that (ab)use processor speculation
    - Processor guesses what code should be run before it knows for sure
- Many ways to "do bad things"
    - Speculate over a bounds check (Spectre v1)
    - Speculate through a bad return address (Spectre RSB)
    - Speculation reading a disabled FPU (LazyFP)
    - And more!

SOPHOS

Background: Flush+Reload

- One possible technique for exfiltrating data inside speculative execution
- Consistent, easy (with asm access)
- Basic idea:
    - (CL)FLUSH each line in a "timing" array
    - Have speculative execution load one of the lines
    - Subsequent attacker loads will find one line faster than the others

Flush+Reload Hypothetical Example

```
...
if (slow_to_load_usually_true) {

    a = out[secret_number & 0x3];

}
...
```

INACTIVE
INACTIVE
INACTIVE
INACTIVE

Flush+Reload Hypothetical Example

```
...
if (slow_to_load_usually_true) {
    a = out[secret_number & 0x3];
}
...
```

INACTIVE
INACTIVE
INACTIVE
INACTIVE

SOPHOS

Flush+Reload Hypothetical Example

```
…
if (slow_to_load_usually_true) {

    a = out[secret_number & 0x3];

}
…
```

INACTIVE
INACTIVE
INACTIVE
INACTIVE

SOPHOS

Flush+Reload Hypothetical Example

```
...
if (slow_to_load_usually_true) {
    a = out[secret_number & 0x3];
}
...
```

INACTIVE
INACTIVE
ACTIVE
INACTIVE

Flush+Reload Hypothetical Example

```
…
if (slow_to_load_usually_true) {

    a = out[secret_number & 0x3];

}
…
```

INACTIVE
INACTIVE
ACTIVE
INACTIVE

Flush+Reload Hypothetical Example

```
for (int i = 0; i < 4; i++) {
    uint64_t start = rdtsc();
    int a = cache[i];
    uint64_t end = rdtsc();
    if (end-start < threshold) {
        secret = i;
    }
}
```

INACTIVE
INACTIVE
ACTIVE
INACTIVE

Flush+Reload Hypothetical Example

```
for (int i = 0; i < 4; i++) {
    uint64_t start = rdtsc();
    int a = cache[i];
    uint64_t end = rdtsc();
    if (end-start < threshold) {
        secret = i;
    }
}
```

i=0 SLOW

INACTIVE
INACTIVE
ACTIVE
INACTIVE

SOPHOS

Flush+Reload Hypothetical Example

```
for (int i = 0; i < 4; i++) {
    uint64_t start = rdtsc();
    int a = cache[i];
    uint64_t end = rdtsc();
    if (end-start < threshold) {
        secret = i;
    }
}
```

i=1 SLOW

ACTIVE
INACTIVE
ACTIVE
INACTIVE

Flush+Reload Hypothetical Example

```
for (int i = 0; i < 4; i++) {
    uint64_t start = rdtsc();
    int a = cache[i];
    uint64_t end = rdtsc();
    if (end-start < threshold) {
        secret = i;
    }
}
```

i=2 **FAST**

ACTIVE
ACTIVE
ACTIVE
INACTIVE

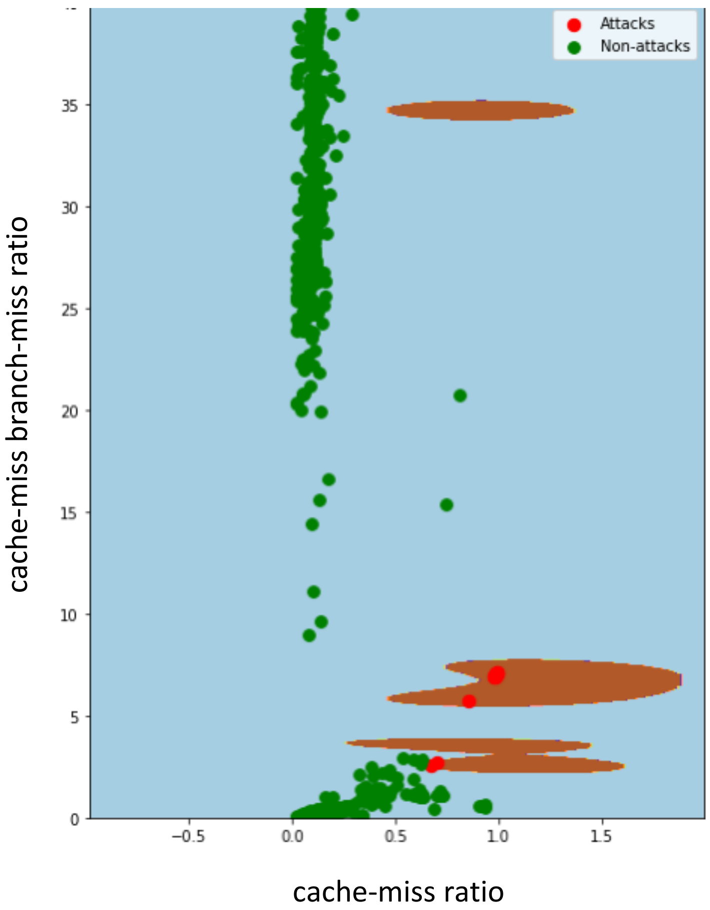
Spectre and Meltdown Detections

- Developed detections shortly after public announcement of the bugs (early 2018)
- Used 3 perf counters as features
  - Cache misses
  - Cache references
  - Branch misses
- First two form "cache miss ratio"
- Third normalizes to the complexity of the program
- Sampled on a 100ms ticker
- Successfully detects all public proof-of-concepts we’ve tried

Spectre and Meltdown

Support Vector Machine - Decision Function visualized

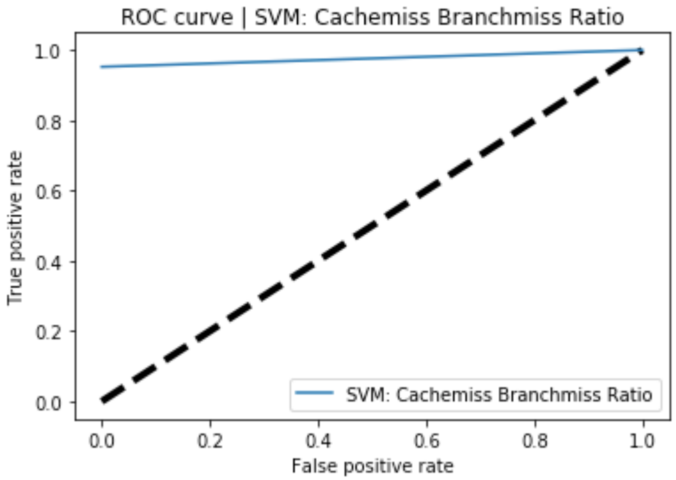
- Plot shows a part of the decision boundary learnt by the SVM model
- Blue shaded region represents benign surface
- Rust shaded region represents malicious surface
- Superimposing the test data points as a scatter plot over this decision boundary where green data points represent baseline data and red data points represent spectre/ meltdown variants



Support Vector Machine

Features: Cache miss ratio, Cache miss - Branch miss ratio

```
SVM: Cachemiss  Branchmiss Ratio | Train accuracy: 0.9997730882686635
SVM: Cachemiss Branchmiss Ratio | Test accuracy: 0.9995393827729157
SVM: Cachemiss Branchmiss Ratio | AUC: 0.9761904761904762
```



Spectre and Meltdown

- This detection can be easily defeated though!
- Mix-in cache friendly code into the proof-of-concept
- Bypasses existing cache-miss-ratio-based detections
  - Lets us achieve an arbitrarily low cache-miss ratio
  - Little runtime overhead (since it's trying to be extremely cache friendly)

Spectre and Meltdown in Hiding

```
// stuff that will be read in a cache-friendly way to evade detection
unsigned long long stuff[65536];

...
// do some stuff that's really cache-nice to throw off detection
register unsigned long long ctr = 0;
for (register int round = 0; round < 80000000; round++) {
    register unsigned long long *p = &stuff[round % (sizeof(stuff) / sizeof(stuff[0]))];
    ctr += *p;
    *p = ctr;
}
...
```

Our Research

Hardware Performance Counters

- Space for 256*256 counters
- Number of documented counters (and what they count) varies per microarchitecture
    - Only a few hundred documented on most microarchitectures
- What if we read *all* of them (even the undocumented ones)?
- **Turns exploit detection into a blackbox ML problem**

Counter Selection

- Ran four programs and sequentially gathered all counters 10 times
  - Optimized/minified _exit(0);
  - Scikit benchmark
  - Spectre v4
  - Spectre v4 in Hiding

Counter Selection (cont’d)

- Removed always zero counters
- Removed counters that had a difference between scikit benchmark and spectre v4 less than 95%
- Removed counters that differed more than 5% between spectre v4 and spectre v4 "in hiding"

- Left with 81 counters
- Interestingly *no documented counters*

Using Undocumented Counters

Exploits of Interest

- Meltdown (aka Spectre v3 - rogue data cache load)
- Spectre v1 (bounds check bypass)
- Spectre v2 (branch target injection)
- Spectre v4 (speculative store bypass)
- Ghosting_spectrev4 (speculative store with evasive changes)

Data Collection

- Used Linux perf tool
- Along with the exploits mentioned before, collected data for the following baseline programs:
  - LibJIT unit tests
  - Scikit-learn benchmark tests
  - Phoronix test suite
  - Linux defconfig compile
  - Sort function
  - Mibench benchmarks
- Counters were measured every 100ms
- Each program was run five times

Algorithms used

- Support Vector Machine
- Random Forest
- eXtreme Gradient Boosting (XGBoost)
- Histogram based Gradient Boosting (HGBoost)

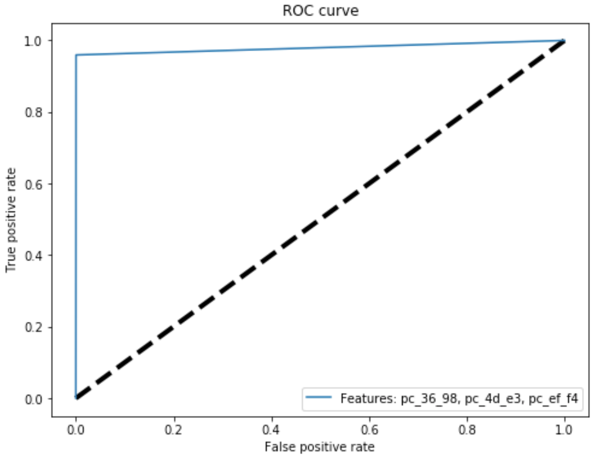
Detecting Spectre (Again)

Model results
Features: 36-98, 4d-e3, ef-f4

F1	F2	F3	intel_arch	model	precision	recall	fpr	fnr	auc	acc	meltdown	spectre1	spectre2	spectre4	spectre4_new
36_98	4d_e3	ef_f4	ivybridge	SVM	1	0.85	0	0.3	0.85	0.99	no	no	no	yes	yes
36_98	4d_e3	ef_f4	ivybridge	XGBoost	0.98	0.94	0.0004	0.12	0.94	0.99	yes	yes	yes	yes	yes
36_98	4d_e3	ef_f4	ivybridge	RF	1	0.86	0	0.28	0.86	0.99	yes	no	no	yes	yes
36_98	4d_e3	ef_f4	ivybridge	HGBoost	0.98	0.94	0.0004	0.112	0.94	0.99	yes	yes	no	yes	yes
36_98	4d_e3	ef_f4	haswell	SVM	0.98	0.93	0.0005	0.13	0.94	0.99	yes	no	no	yes	yes
36_98	4d_e3	ef_f4	haswell	XGBoost	0.99	0.98	0.0004	0.04	0.98	0.99	yes	yes	yes	yes	yes
36_98	4d_e3	ef_f4	haswell	RF	1	0.97	0.0001	0.06	0.97	0.99	yes	no	no	yes	yes
36_98	4d_e3	ef_f4	haswell	HGBoost	0.98	0.98	0.0008	0.04	0.98	0.99	yes	yes	yes	yes	yes

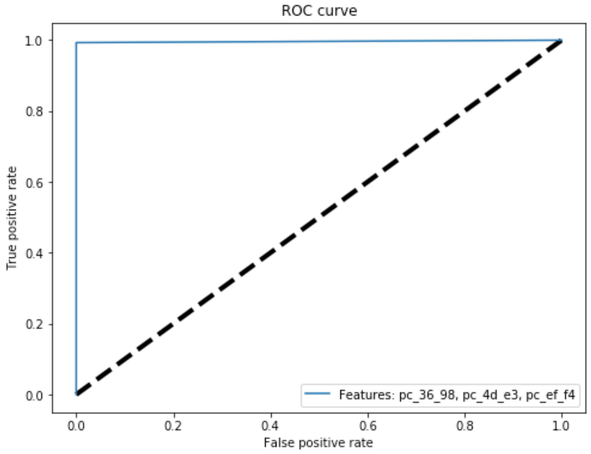
XGBoost AUC for Test and Hold-out Dataset

```
Train accuracy: 0.9998672022841207
Test accuracy: 0.9988542158118218
AUC: 0.9794988379651749
False Positive Rate: 0.00041191816559110257
False Negative Rate: 0.04059040590405904
```



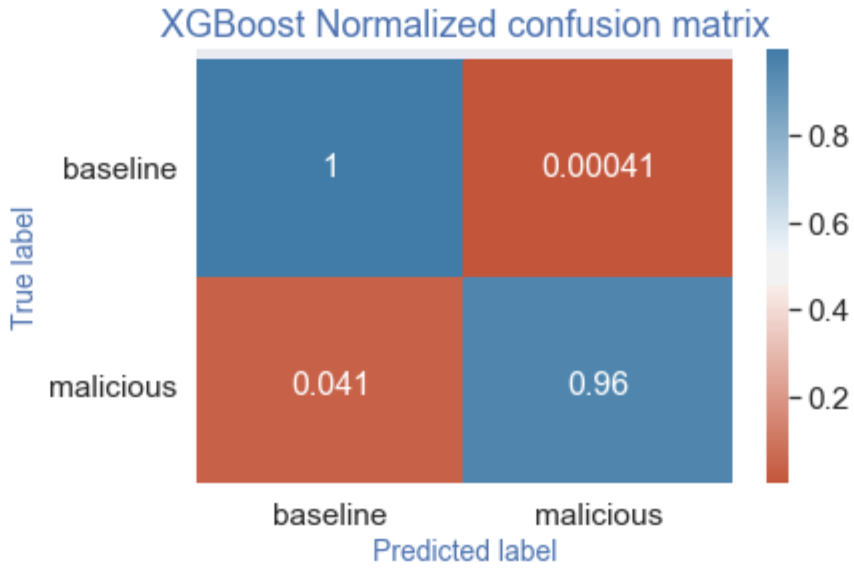
	precision	recall	f1-score	support
0	1.00	1.00	1.00	14566
1	0.98	0.96	0.97	271
accuracy			1.00	14837
macro avg	0.99	0.98	0.98	14837
weighted avg	1.00	1.00	1.00	14837

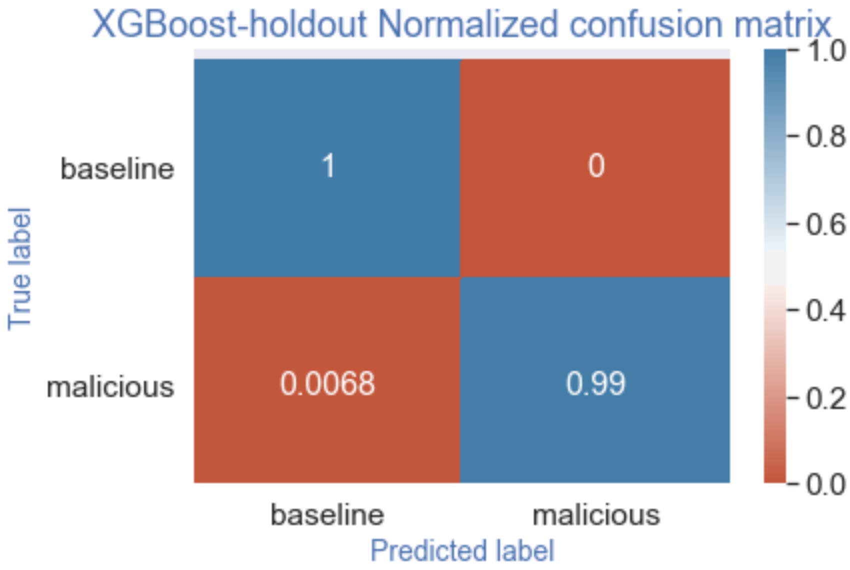
```
Train accuracy: 0.9998672022841207
Test accuracy: 0.9999321435841759
AUC: 0.9965928449744463
False Positive Rate: 0.0
False Negative Rate: 0.0068143100511073255
```



	precision	recall	f1-score	support
0	1.00	1.00	1.00	58361
1	1.00	0.99	1.00	587
accuracy			1.00	58948
macro avg	1.00	1.00	1.00	58948
weighted avg	1.00	1.00	1.00	58948

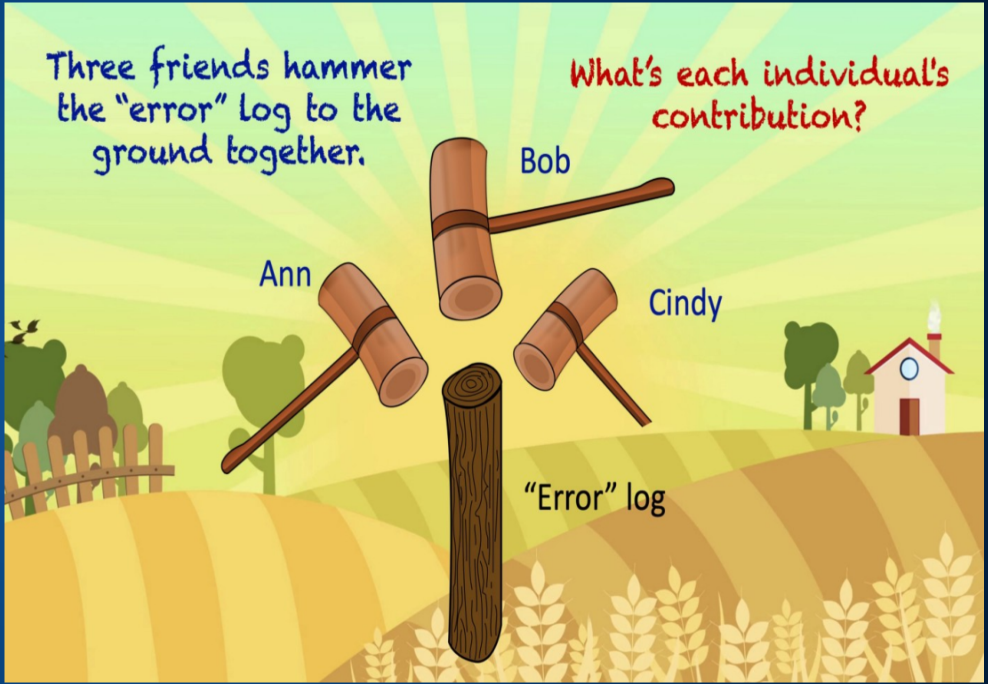
XGBoost Normalized Confusion Matrices



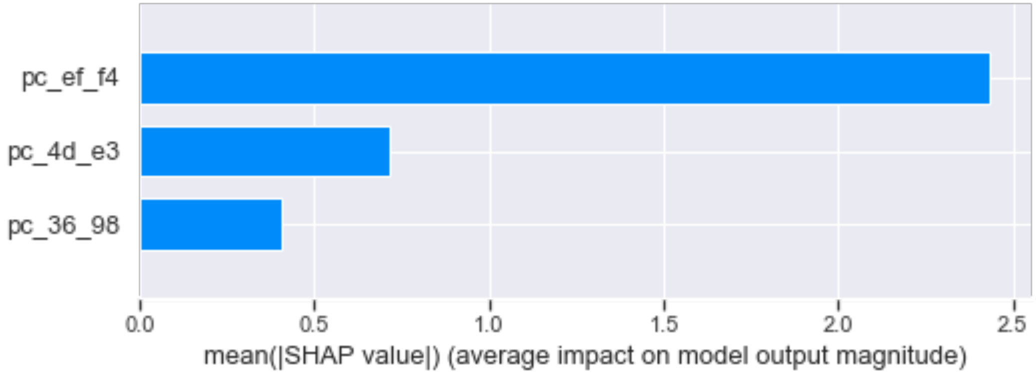


SHAP Model Interpretation

- SHapley Additive exPlanation (Lundberg, et al)
- Based on Shapely values, a technique used in game theory to determine how much each player in a collaborative game has contributed to its success
- Each SHAP value measures how much each feature in our model contributes to the prediction, either positively or negatively



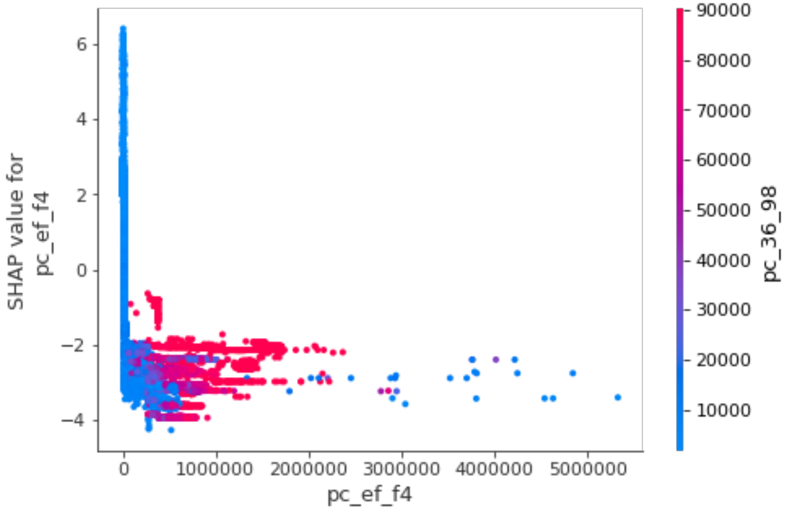
XGBoost Feature Importance



XGBoost Partial Dependence Plot

- Shows the marginal effect that one or two variables have on the predicted outcome.
- Whether the relationship between the target and the variable is linear, monotonic, or more complex
- Let’s see the partial dependence plots for each of the three features

XGBoost Partial Dependence Plot (cont'd)

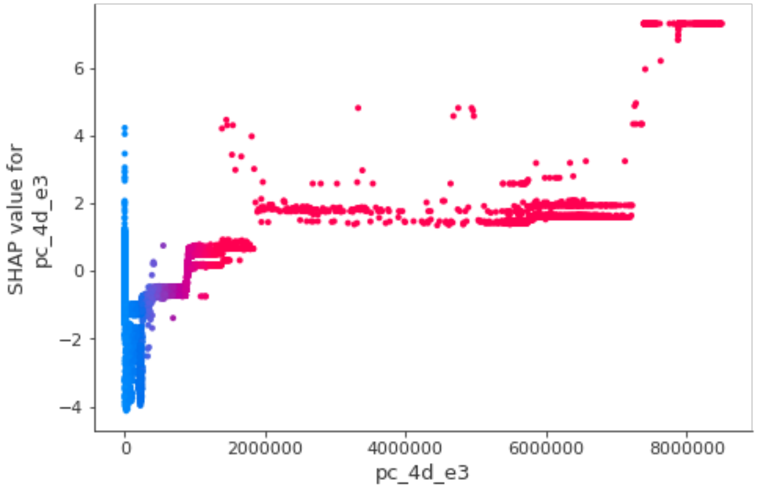


Feature: ef-f4

- High SHAP value, low counter value -> Benign
- Low SHAP value, high counter value -> Malicious

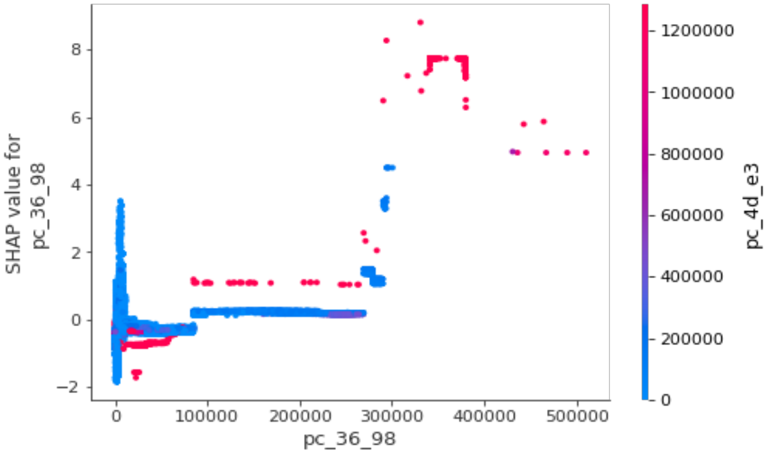
XGBoost Partial Dependence Plot (cont'd)

Feature: 4d-e3



- Low SHAP value, low counter value -> Benign
- High SHAP value, high counter value -> Malicious

XGBoost Partial Dependence Plot (cont'd)

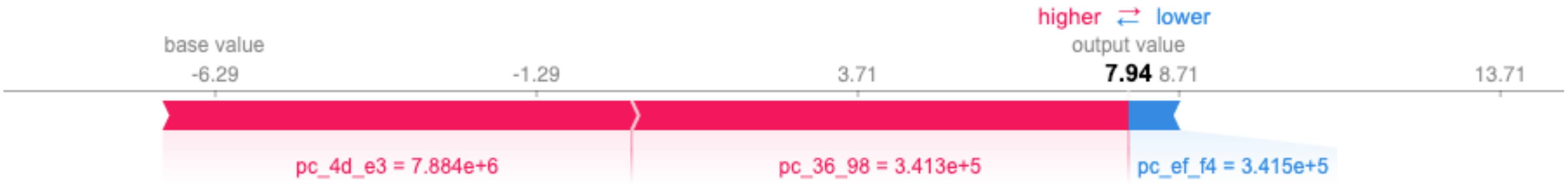


Feature: 36-98

- Low SHAP value, low counter value -> Benign
- High SHAP value, high counter value -> Malicious

SHAP Force Plots

How each feature pushes the prediction to 1/0



A Surprise Confirmation

SOPHOS

Some Time Later...

- Widely publicized leak of Immunity Inc.'s CANVAS
    - Exploit toolkit
- Included a Spectre-style exploit, with a helpful test flag!
- Ran the "in-the-wild" exploit, and our model was able to detect it

Interpretation

Warning: *speculation* ahead

SOPHOS

Possible Interpretation of Counters: ef-f4

- A *single* support file in Intel VTune names the 0xEF event_id as "CORE_SNOOP_RESPONSE"
  - Description: "tbd" - thanks Intel
  - Supposedly only for SKL-X and Cascade Lake…
  - 0xf4 umask not documented
- Hypothesis: counter is detecting the responses from other cores when CLFLUSH invalidates cache lines
- Counters showed "malicious" even when the cache sampling was broken
  - Supports the theory that this is measuring cache evictions instead of sampling

Possible Interpretation of Counters: 36-98

- Haswell-EP documentation names the **uncore** PMC 0x36 as "UNC_C_TOR_OCCUPANCY"
  - 0x98 umask not documented
  - Other umasks refer to a separate MSR being used to filter/select data
- Uncore is responsible for LLC coherence though...
- Maybe "seeing through" to the uncore PMU because of an implementation detail?

Q&A

SOPHOS

References

References

- Counter Interpretation:
  - https://dl.acm.org/doi/pdf/10.1109/SC.2018.00021
  - https://software.intel.com/content/www/us/en/develop/download/intel-xeon-processor-scalable-memory-family-uncore-performance-monitoring-reference-manual.html
- Model Interpretation:
  - https://www.nature.com/articles/s42256-019-0138-9
  - https://github.com/slundberg/shap
  - https://towardsdatascience.com/explain-your-model-with-the-shap-values-bc36aac4de3d
  - https://towardsdatascience.com/shap-explain-any-machine-learning-model-in-python-24207127cad7