Detecting Exploits with Novel Hardware Performance Counters and ML Magic

Nick Gregory
Harini Kannan

Black Hat USA 2020

Who Are We?

Nick "ghost" Gregory

- Research Scientist @ Capsule8
- Background in binary exploitation and low-level systems
- Grad of NYU Tandon / OSIRIS Lab
- Now "Hacker in Residence" at the lab

Email: ghost-spam@capsule8.com (but you know I don't like spam)

Twitter: @kallsyms

Harini Kannan

- Data Scientist @ Capsule8
- Background in Business Statistics
- Currently on:
  - System user behavior profiling
  - Text analytics
  - Interpretable ML
  - MLOps

Twitter: @jarvision_

Website: https://harini.blog/

Introduction

What Are We Covering?

1. Hardware Performance Counters - what and why
2. Prior work - using counters to detect Spectre/Meltdown
3. This work
    a. Exploring undocumented counters
    b. Training models on undocumented counters
    c. Detection capabilities with trained models
    d. Interpretation of results
4. Future work

Main Question

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

Hardware Performance Counters

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.

Hardware Performance Counters

- We'll be focusing on the "CPU" PMU today
- Most Intel CPUs let you pick a few of these counters to monitor at once (per core)
- Specified as event_id, umask
  - event_id: broad category of event (cache, branches, etc.)
  - umask: specific counter (level 1 cache misses filled by level 2)

Hardware Performance Counters

- On Linux, interact with counters through the "perf" subsystem (and CLI)
- For example:
  - `perf stat -e cache-misses -- /bin/ls`
  - `perf stat -e "cpu/event=0xef,umask=0xf4/" -- /bin/ls`
- Multiple sampling methods
  - Time/Ticker
  - Count threshold
  - Entire program run

A Couple of Years Ago...

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!

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];

}
...
```

Flush+Reload Hypothetical Example

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

Flush+Reload Hypothetical Example

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

Flush+Reload Hypothetical Example

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

Flush+Reload Hypothetical Example

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

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;
    }
}
```

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

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

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**

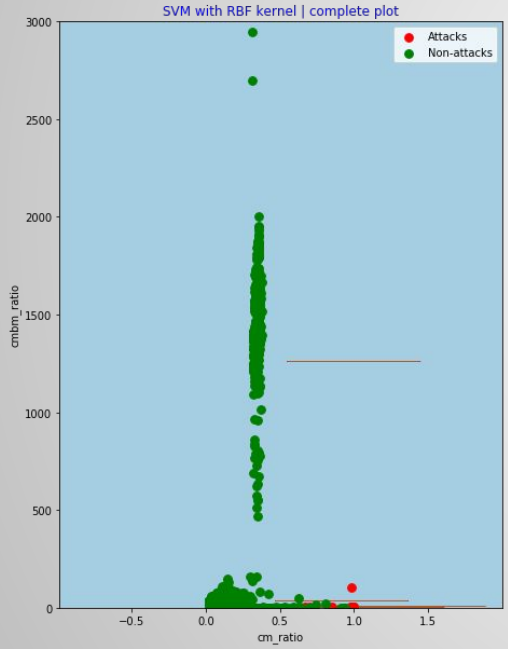
Spectre and Meltdown Detections

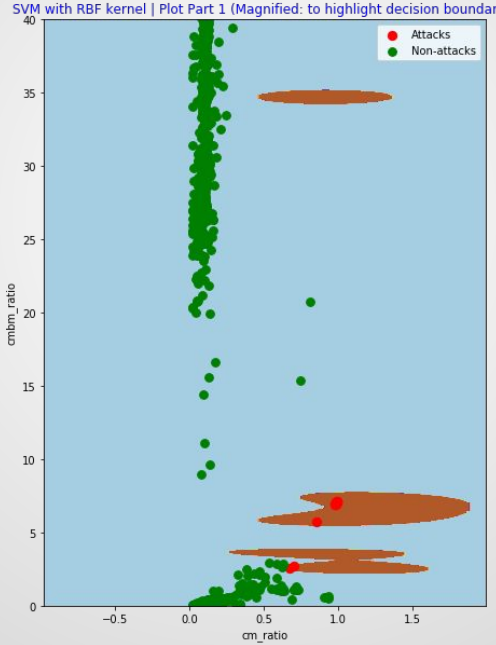
- Developed detections shortly after public announcement of the bugs (early 2018)
- Used 3 perf counters as features
  - Cache misses
  - Cache references
  - Branch misses
- First two form "cache miss ratio"
- Third normalizes to the complexity of the program
- Sampled on a 100ms ticker
- Successfully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

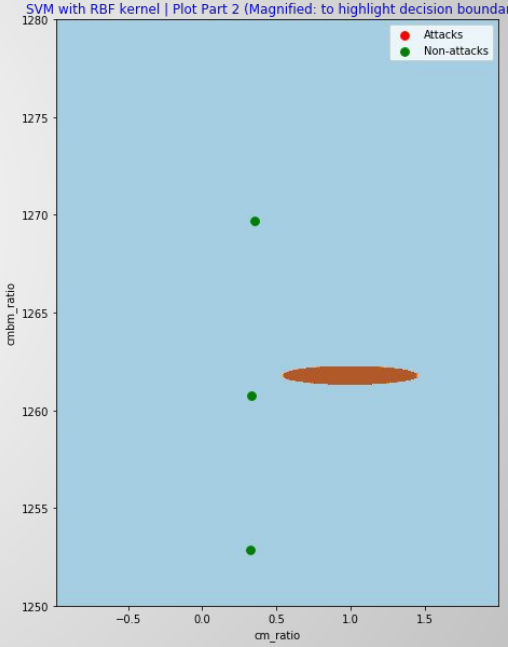
Spectre and Meltdown

Support Vector Machine - Decision Function visualized

Test Data | SVM-RBF kernel | Features: Cachemiss, Cachemiss-Branchmiss



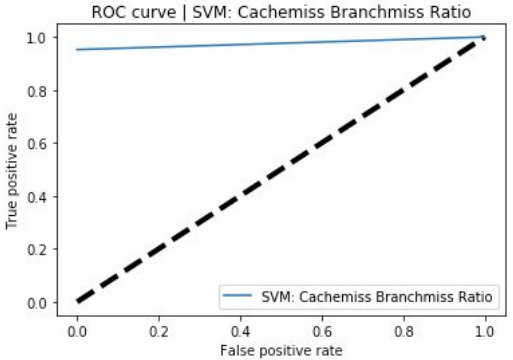




Support Vector Machine

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
```



Model	accuracy (%)	fpr	fnr	sensitivity (%)	specificity (%)
1_Capsule8_deterministic	99.8618148	0	0.142857	85.7142857	100
2_SVM	99.9539383	0	0.047619	95.2380952	100
5_Ensemble_cap8det_svm	100	0	0	100	100

Spectre and Meltdown

- This detection can be easily defeated though!
- Mix-in cache friendly code into the proof-of-concept

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*

Counter Selection (cont'd)

- All tests run on
  - Intel Xeon E5-2667 v3 (Haswell)
  - Intel Core i5-3210M (Ivybridge)
- Results will differ on other microarchitectures

Counters of Interest

- Dataset 1:
  - `event_id=0xef,umask=0xf4`
  - `event_id=0x4d,umask=0xe3`
  - `event_id=0x36,umask=0x98`

- Dataset 2 (not covering due to time constraints):
  - `event_id=0xef,umask=0xf4`
  - `event_id=0x4d,umask=0x`**`b1`**
  - `event_id=0x`**`d5`**`,umask=0x`**`a6`**

Over to Harini

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)
- Return-Oriented Programming (ROP)

Data Collection

- Using Linux perf counters
- Along with the exploits mentioned before, collected data for the following baseline programs:
  - LibJIT unit tests
  - Scikit-learn benchmark tests
  - Phoronix-nginx test suite
  - Linux defconfig compile
- Selected counters were measured every 100ms
- Each program was run five times

Model Metrics Calculated

1. Precision
2. Recall
3. F1-score
4. False Positive Rate (FPR)
5. False Negative Rate (FNR)
6. Area Under the Curve (AUC)
7. Test Accuracy
8. Confusion Matrix

What Do These Mean?

PRECISION

Precision is the ability a classifier to not label a true negative observation as positive.

$$\frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

ChrisAlbon

RECALL

"Recall is about the real positives"

$$\frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Recall is the ability of the classifier to find positive examples. If we wanted to be certain to find all positive examples, we could maximize recall.

Chris Albon

F1 SCORE

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

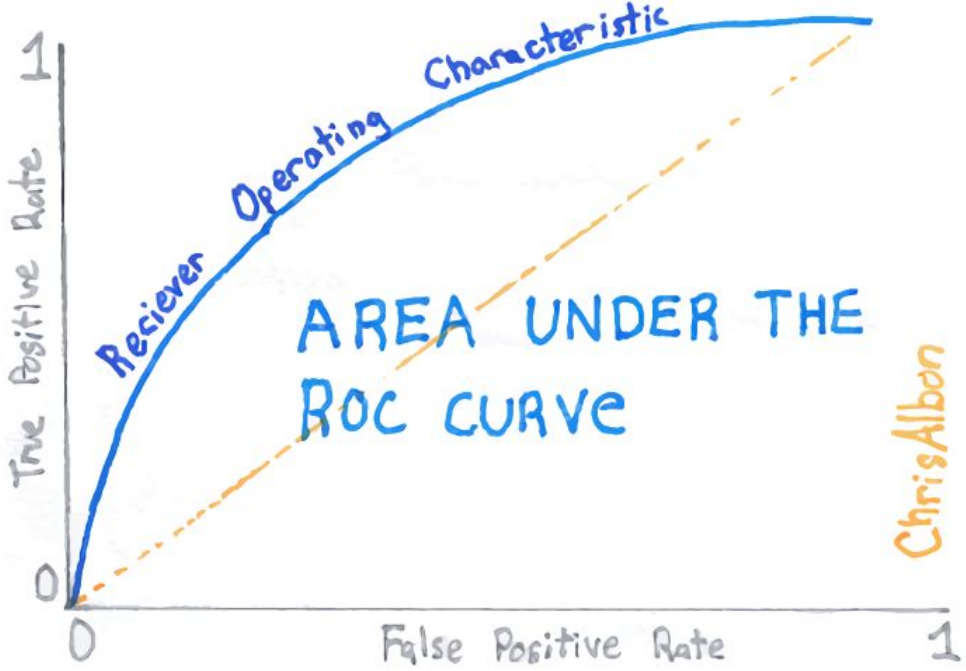
F1 score is the harmonic mean of precision and recall. Values range from 0 (bad) to 1 (good).

Chris Albon

AUC

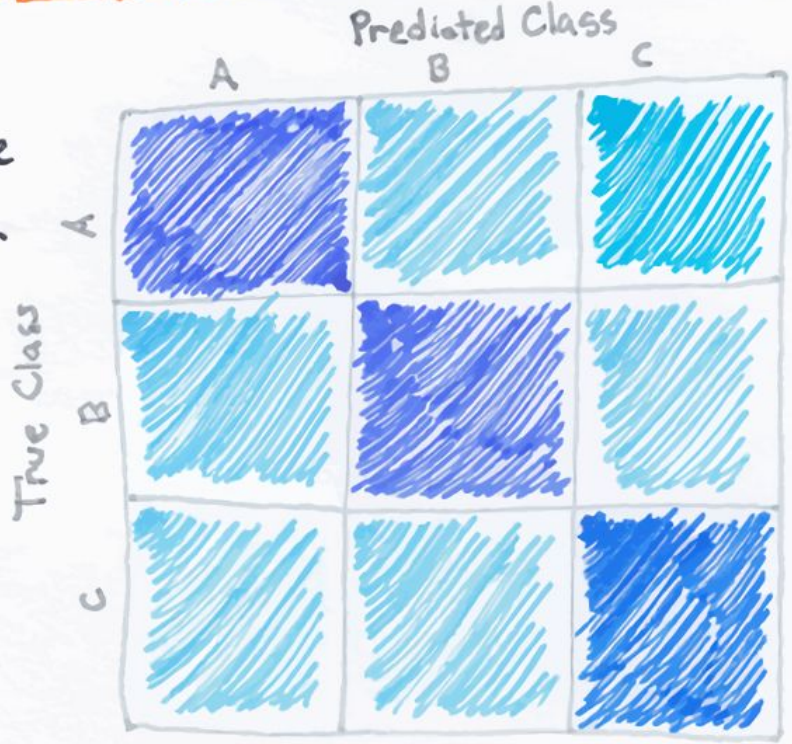
AREA UNDER THE CURVE

The ROC curve represents the true positive rate and false positive rate for all probability thresholds of a binary classifier. The AUC evaluates the overall quality of the model. More AUC, the better.



CONFUSION MATRIX

Confusion matrices visualize the accuracy of a classifier by comparing the true and predicted classes. Off diagonal squares are incorrect predictions.

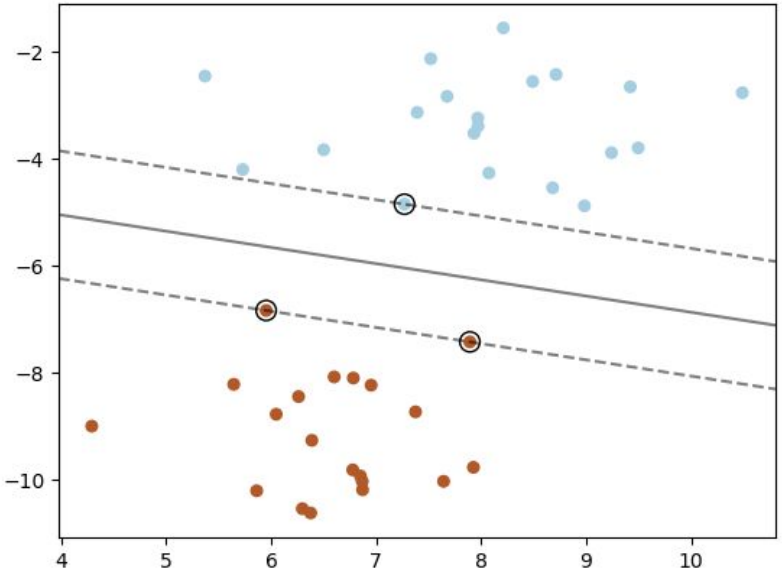


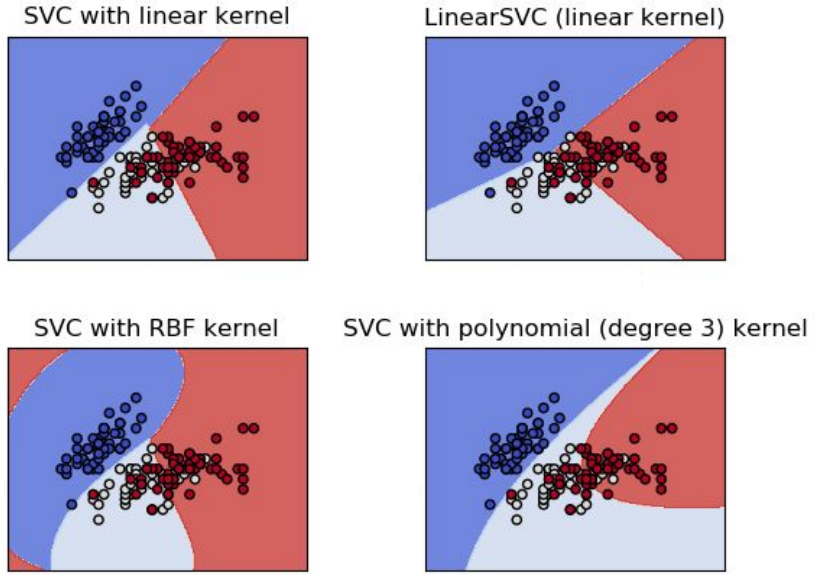
Chris Albon

Algorithms used

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

Support Vector Machine

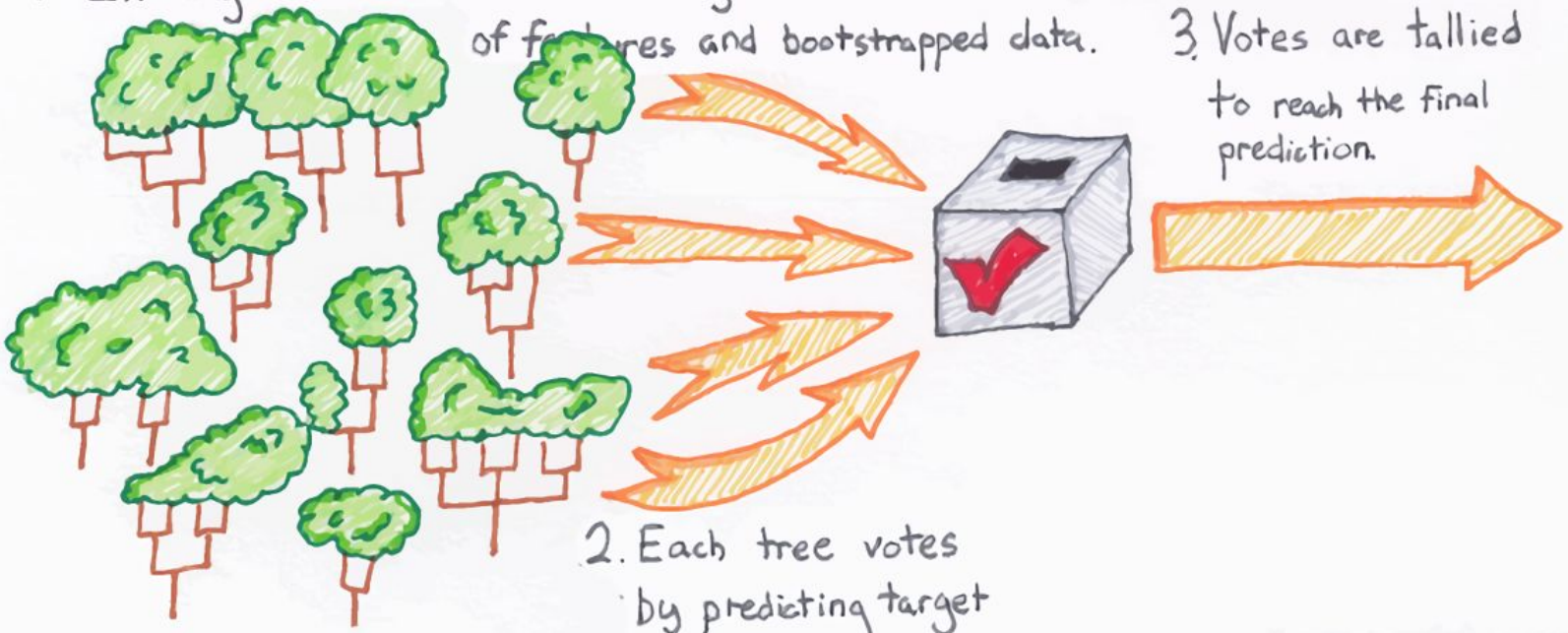




RANDOM FOREST

CLASSIFICATION

1) 1. Many trees are created using random subsets of features and bootstrapped data.

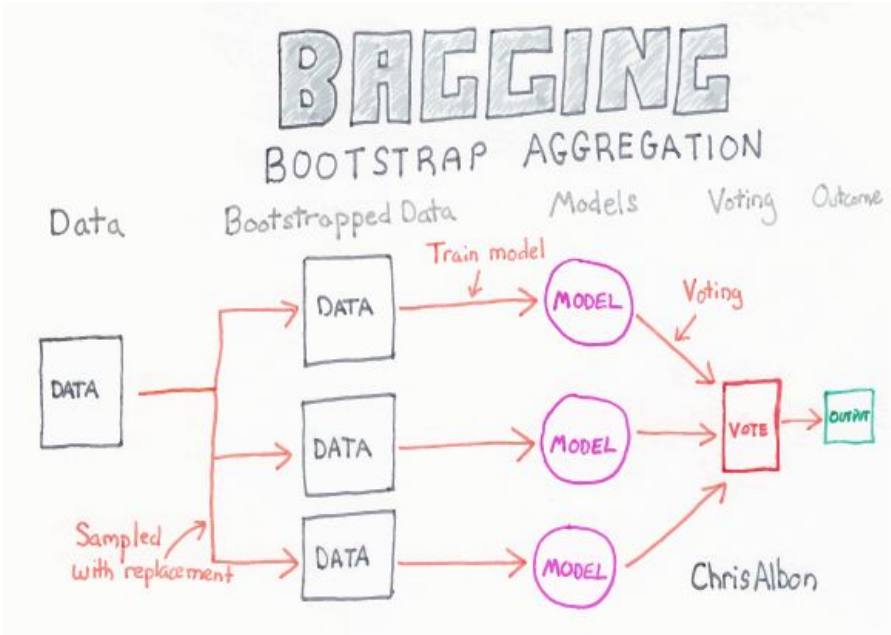


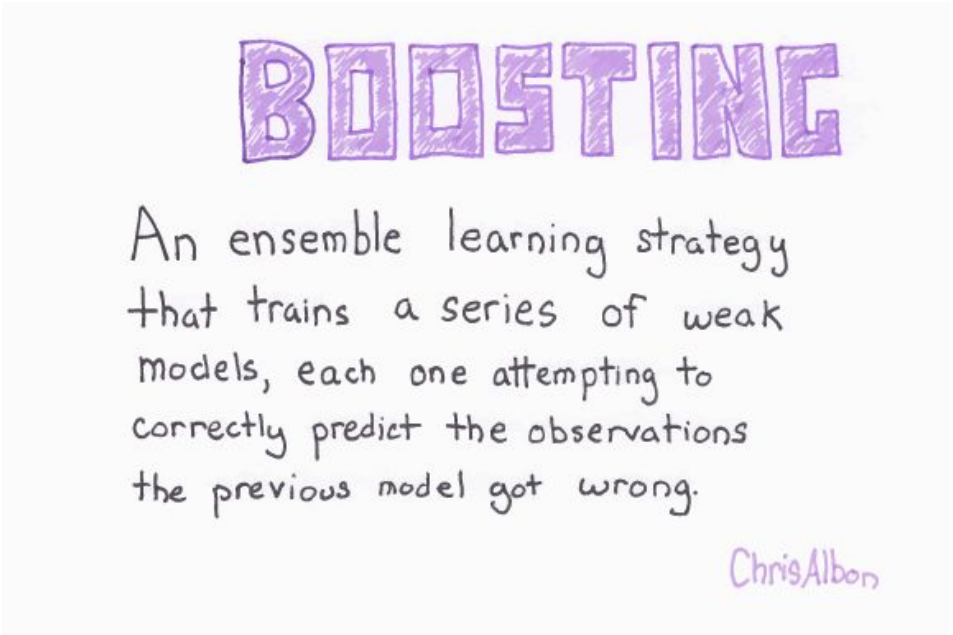
2. Each tree votes by predicting target class.

3. Votes are tallied to reach the final prediction.

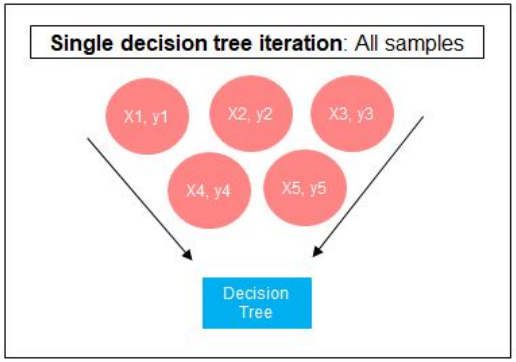
CHRIS ALBON

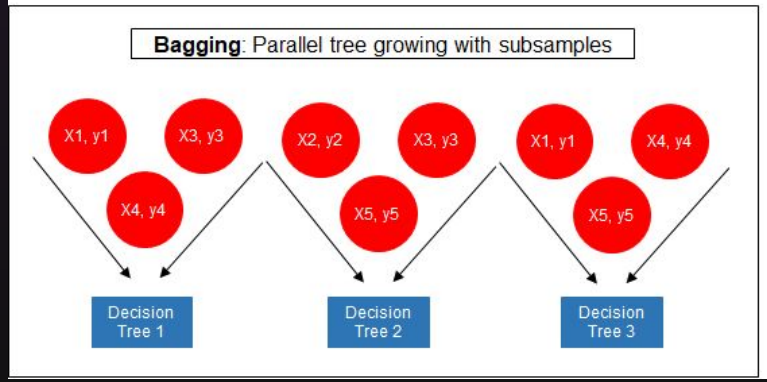
Bagging Vs Boosting

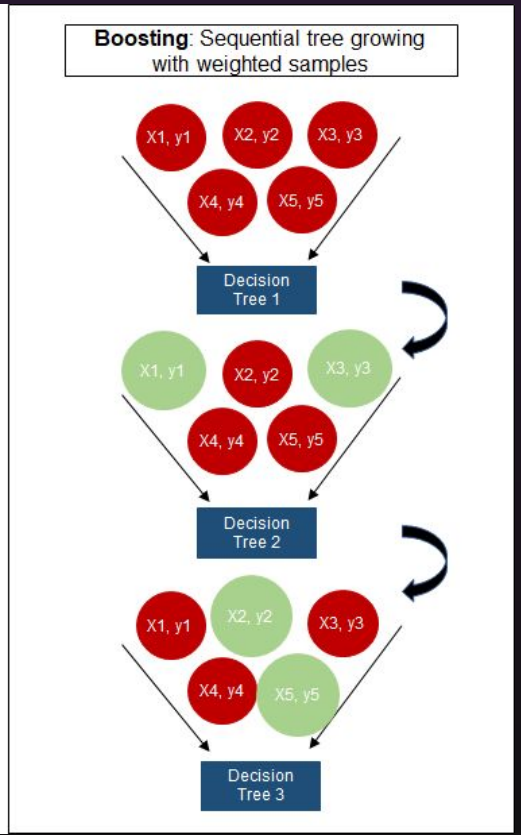




eXtreme Gradient Boosting (uses Boosting)







- Builds on weak classifiers (high bias, low variance)
- Add a classifier (tree) at a time, so that next classifier is trained to improve the already trained ensemble

Histogram based Gradient Boosting

- A faster implementation gradient boosting classifier when no. of samples is higher
- It bins input samples into integer-valued bins (typically 256 bins) which reduces the no. of splitting points to consider
- Allows the algorithm to leverage integer-based data structures (histograms) instead of relying on sorted continuous values when building the trees

Detecting Spectre (Again)

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

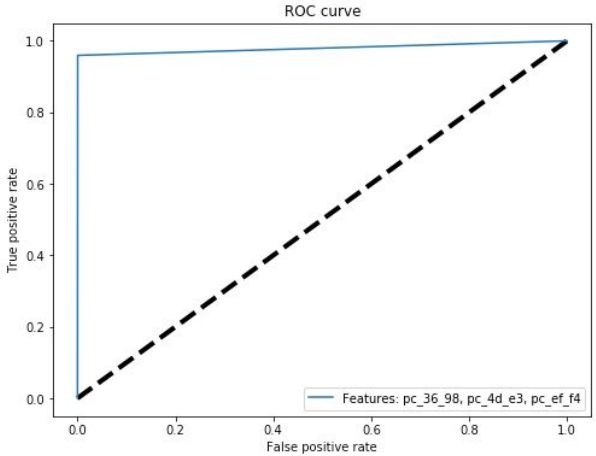
F1	F2	F3	intel_arch	model	precision	recall	fpr	fnr	auc	acc	meltdown	spectre1	spectre2	spectre4	spectre4_new
36_98	4d_e3	ef_f4	ivybridge	SVM	1	0.85	0	0.3	0.85	0.99	no	no	no	yes	yes
36_98	4d_e3	ef_f4	ivybridge	XGBoost	0.98	0.94	0.0004	0.12	0.94	0.99	yes	yes	yes	yes	yes
36_98	4d_e3	ef_f4	ivybridge	RF	1	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

Best feature set and model

- Dataset 1 with with features 36-98, 4d-e3, ef-f4 perform the best
- XGBoost is the best model so far
  - 99% precision
  - 98% recall
  - 0.04% FPR
  - 4% FNR
  - 98% AUC
- Note: Here the FNR denotes the part of exploit(s) that’s missed, the model itself caught most parts of all exploits

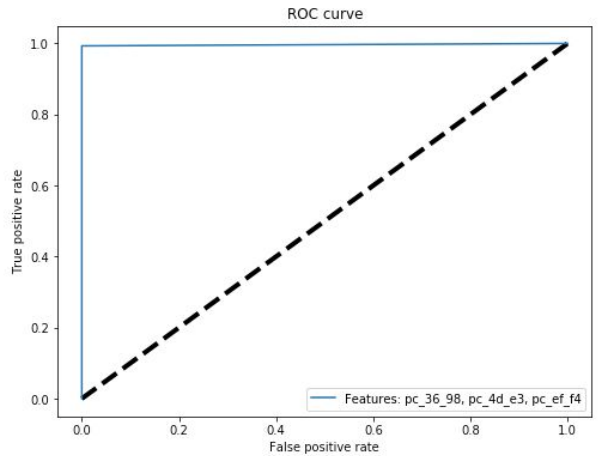
XGBoost AUC for test and hold-out dataset

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



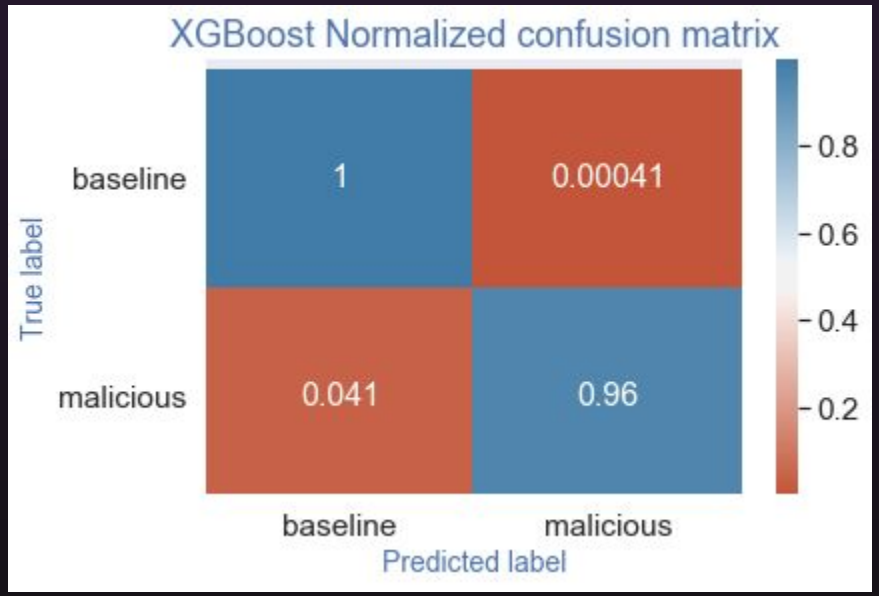
	precision	recall	f1-score	support
0	1.00	1.00	1.00	14566
1	0.98	0.96	0.97	271
accuracy			1.00	14837
macro avg	0.99	0.98	0.98	14837
weighted avg	1.00	1.00	1.00	14837

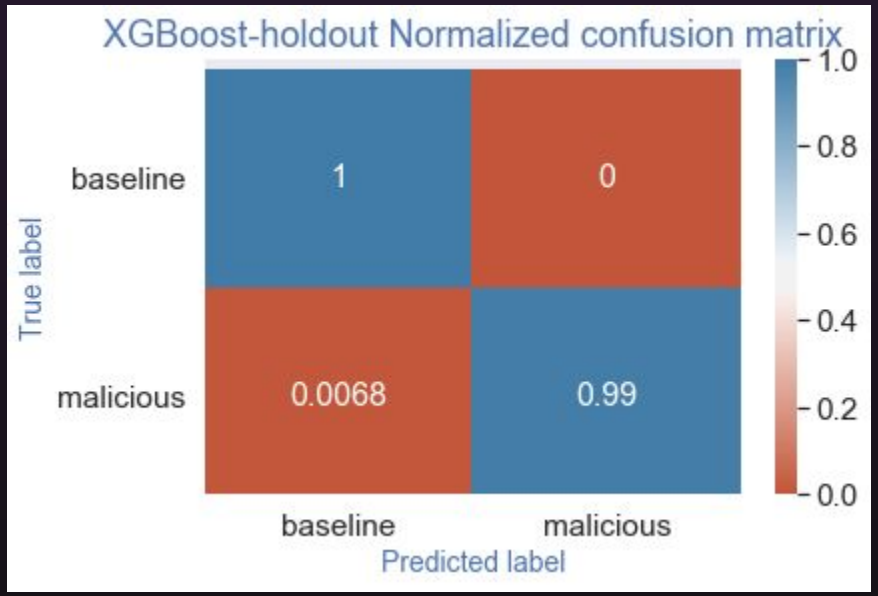
```
Train accuracy: 0.9998672022841207
Test accuracy: 0.9999321435841759
AUC: 0.9965928449744463
False Positive Rate: 0.0
False Negative Rate: 0.0068143100511073255
```



	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

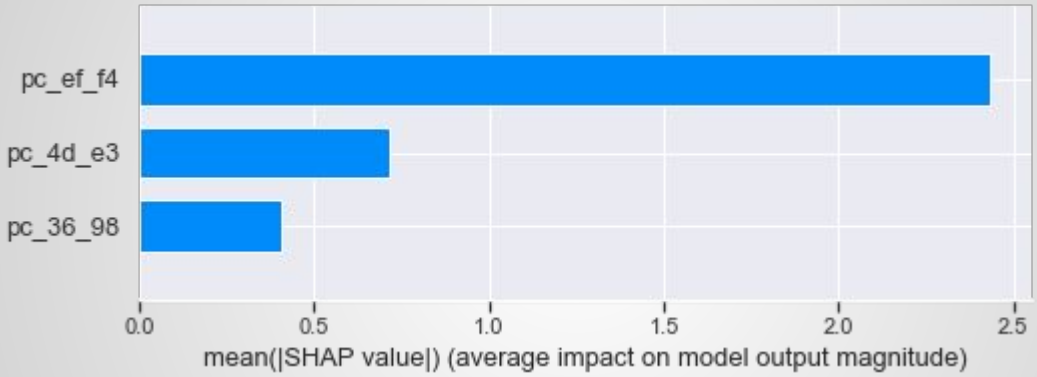




SHAP model interpretation

- 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

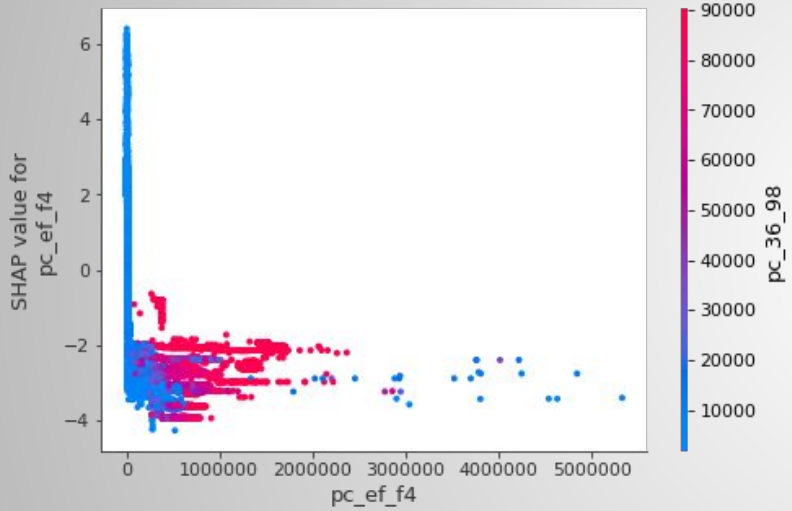
XGBoost Feature Importance



XGBoost Partial Dependence Plot

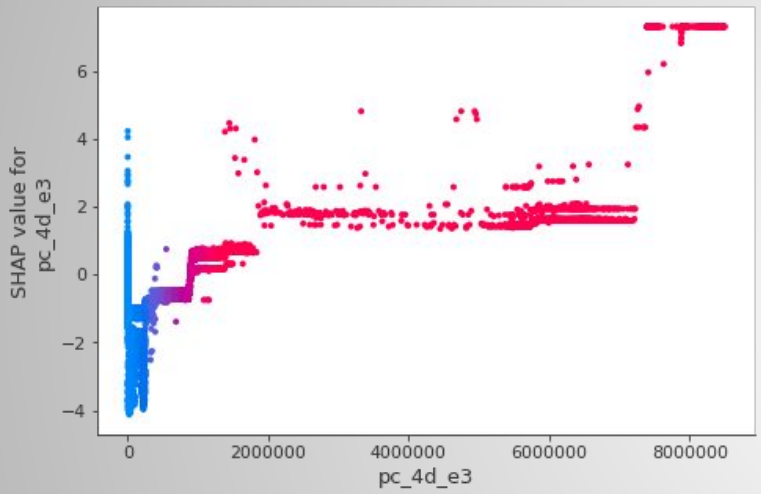
- Shows the marginal effect that one or two variables have on the predicted outcome.
- Whether the relationship between the target and the variable is linear, monotonic, or more complex
- Let’s see the partial dependence plots for each of the three features

XGBoost Partial Dependence Plot (cont'd)



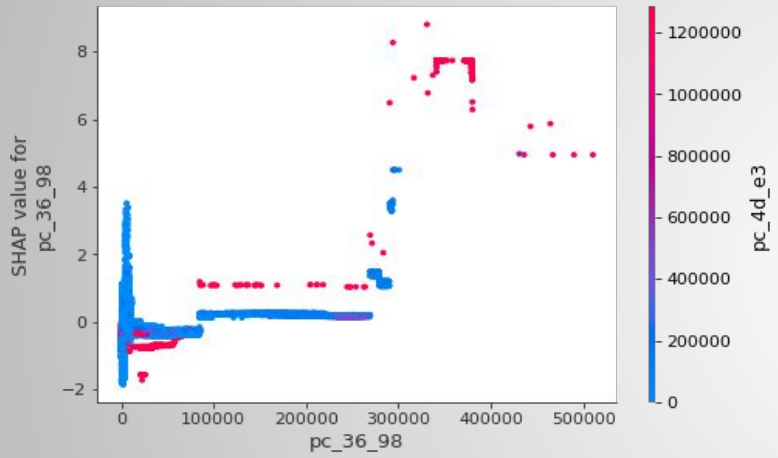
- Plot shows SHAP values for ef-f4 clearly influencing extremely negatively, helping the model classify the baseline data correctly.
- There is some interaction with feature 36-98 where it's values are between 10k-30k

XGBoost Partial Dependence Plot (cont'd)



- Partial dependence plot for feature 4d-e3 shows there is an approximately linear and positive trend between 4d-e3 and the target variable
- It clearly doesn't react with any other feature

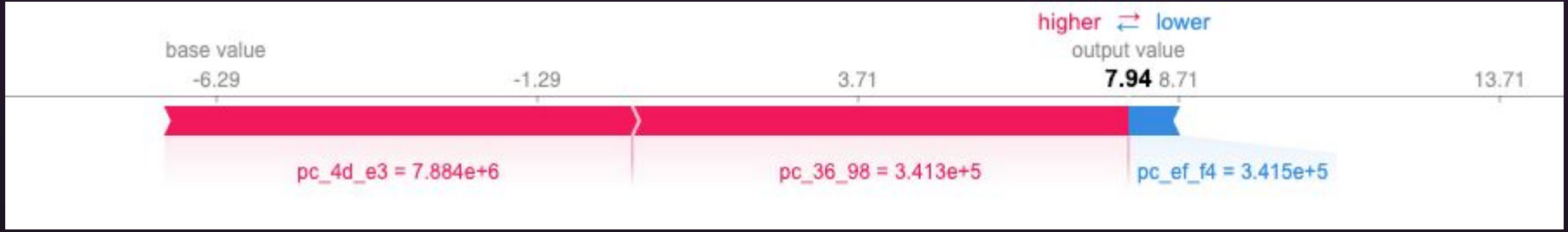
XGBoost Partial Dependence Plot (cont'd)

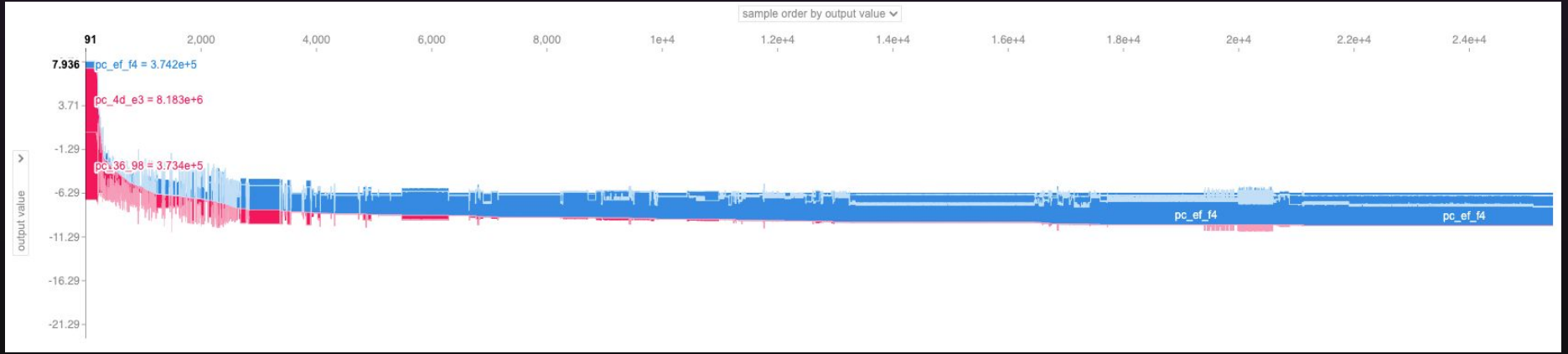


- Plot 2 shows SHAP values for 36-98, significant impact can be seen for the highest and the lowest values of the feature.
- There is some interaction with feature 4d-e3 for the values around 75k-300k

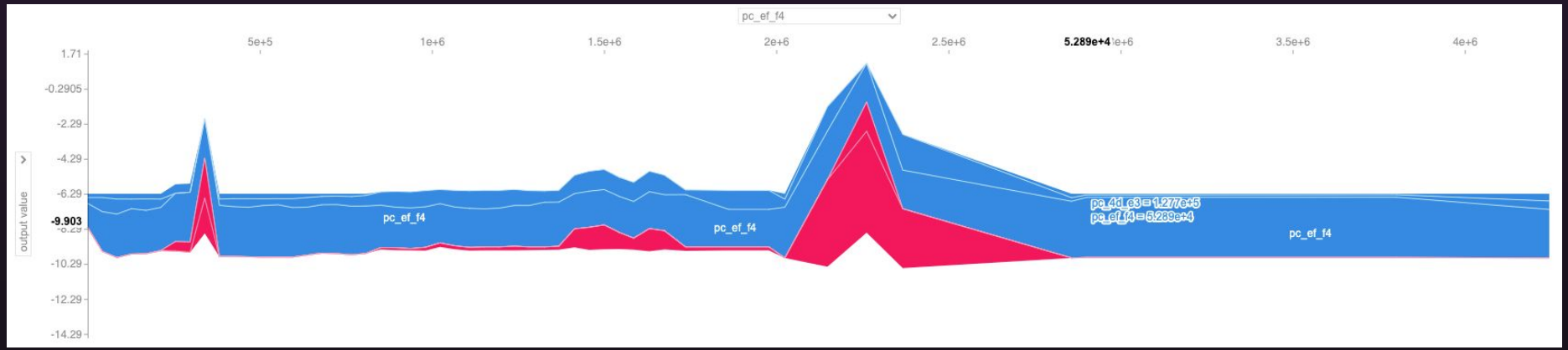
SHAP Force plots

How each feature pushes the prediction to 1/0

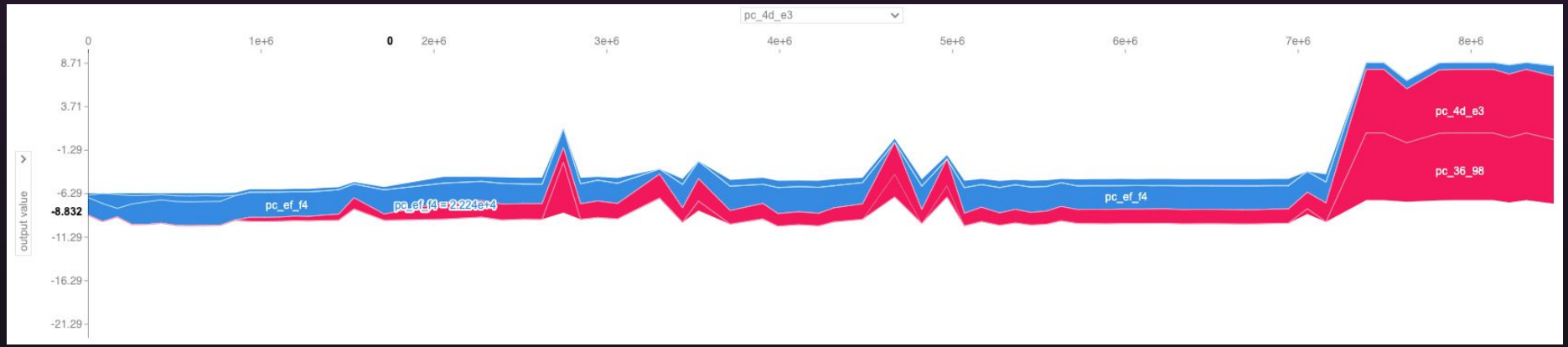




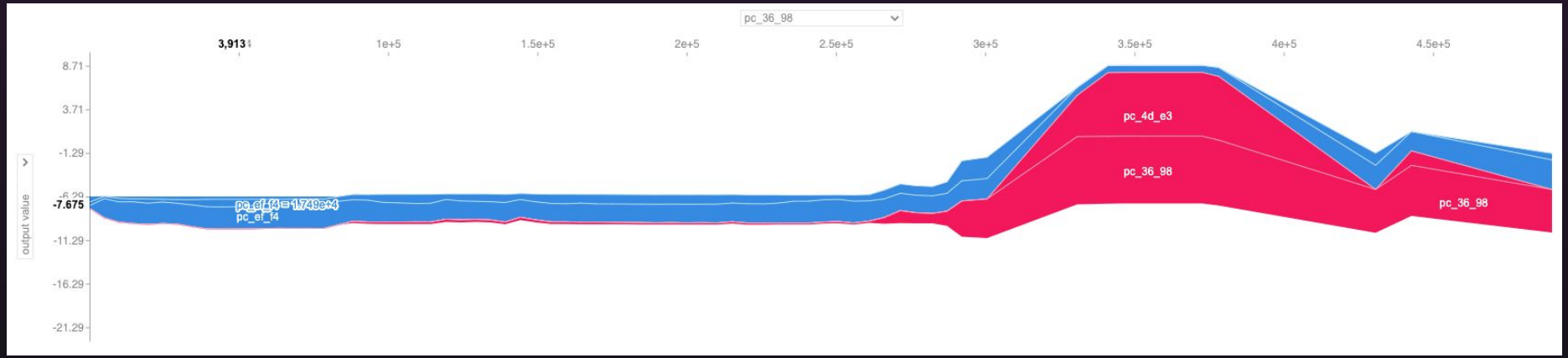
SHAP Force plot for ef-f4



SHAP Force plot for 4d-e3



SHAP Force plot for 36-98



Over to Ghost

Detecting Other Exploits

Detecting ROP

- Prior work
  - Last Branch Records (LBR) / Processor Trace (PT)
    - Sampling throughput/overhead
  - Branch mispredicts
    - ROP chains make the processor's return stack buffer useless
    - Problem: ROP chains are short
      - 50-100 gadgets at most
      - Gives a weak signal

Data Collection

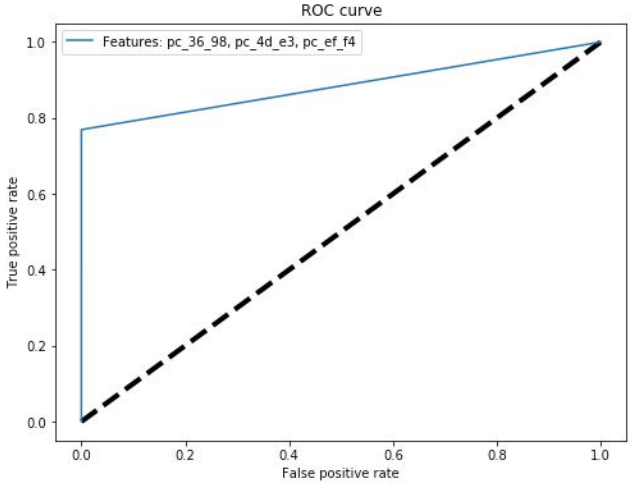
- Ran the ROP exploit 100x in our experiments to maximize signal
- Added a new baseline program: exec-only
  - Executes the same shell as the ROP exploit, but without ROP
  - Used to ensure that we're picking up the ROP itself, not a side effect of the shell creation

Detecting ROP (cont'd)

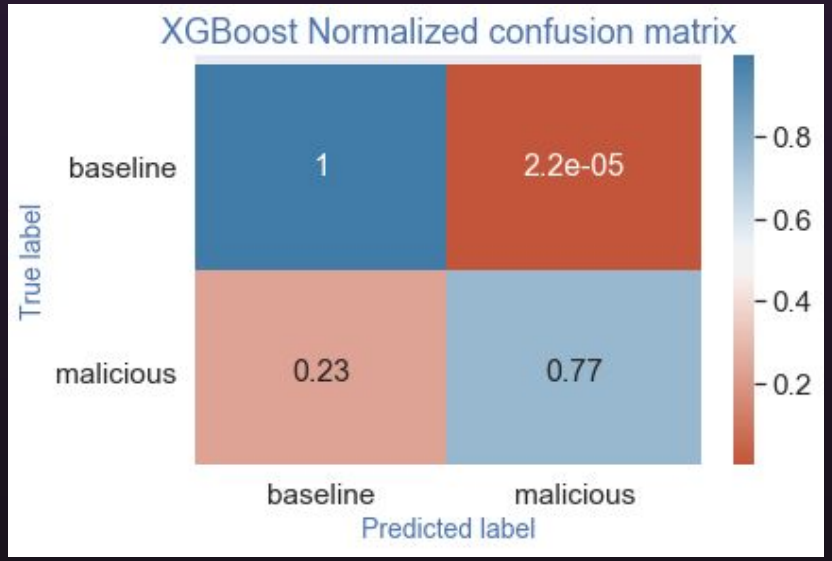
- Same counters work?!?

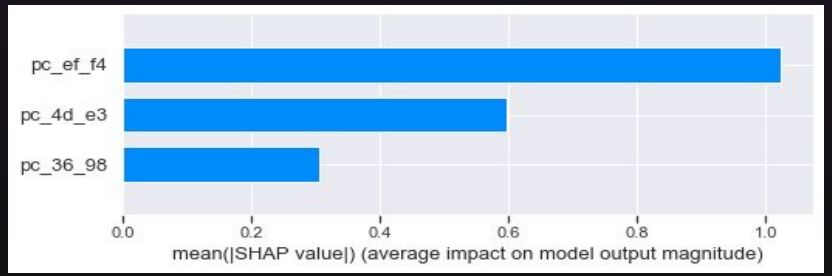
HGBoost AUC and confusion matrix

```
Train accuracy: 0.9999046448550601
Test accuracy: 0.9998494235071417
AUC: 0.8846046231328827
False Positive Rate: 2.1522965003658902e-05
False Negative Rate: 0.23076923076923078
```



	precision	recall	f1-score	support
0	1.00	1.00	1.00	46462
1	0.95	0.77	0.85	26
accuracy			1.00	46488
macro avg	0.98	0.88	0.93	46488
weighted avg	1.00	1.00	1.00	46488





Interpretation
Warning: *speculation* ahead

Interpretation - Spectre & co.

- 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...
- 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

Interpretation - ROP

- Very unsure.
- Detecting the embedded stack pivot?
  - Invalidation/flushing of store buffers for the stack?
- Indirectly detecting the RSB mispredicts?
  - Caches loading based on RSB but all returns don't go to expected location

Future Work

Future Work

- Generalizing/automating data collection
  - Collecting data on a broader set of microarchitectures and analyzing differences

- Other PMUs (uncore counters on Intel chips could be promising)
- Non-Intel x86 (AMD)
- ARM
  - Potentially interesting vendor-specific internals?

Closing Remarks

- Due to the nature of things being undocumented, we don't know what the counters in this talk actually measure
- Please let us know if you have any ideas/knowledge/experiments that could help determine those
- Or the chip manufacturers could release more documentation :)

Q&A

Resources

Resources

- https://www.blackhat.com/docs/us-15/materials/us-15-Herath-These-Are-Not-Your-Grand-Daddys-CPU-Performance-Counters-CPU-Hardware-Performance-Counters-For-Security.pdf

References

References

- 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
- Graphics
  - https://techcommunity.microsoft.com/t5/windows-kernel-internals/understanding-hardware-enforced-stack-protection/ba-p/1247815?lightbox-message-images-1247815=178977iD35F1A3BB0C30431
  - https://machinelearningflashcards.com/
- Model Interpretation:
  - https://www.nature.com/articles/s42256-019-0138-9
  - https://github.com/slundberg/shap

Appendix

Model results
Features: 4d-b1, d5-a6, ef-f4

F1	F2	F3	intel_arch	model	precision	recall	fpr	fnr	auc	acc	meltdown	spectre1	spectre2	spectre4	spectre4_new
4d_b1	d5_a6	ef_f4	ivybridge	SVM	0.99	0.81	0.0002	0.37	0.81	0.99	yes	no	no	yes	yes
4d_b1	d5_a6	ef_f4	ivybridge	XGBoost	0.98	0.88	0.0005	0.25	0.88	0.99	yes	yes	yes	yes	yes
4d_b1	d5_a6	ef_f4	ivybridge	RF	0.99	0.86	0.0002	0.28	0.86	0.99	yes	yes	no	yes	yes
4d_b1	d5_a6	ef_f4	ivybridge	HGBoost	0.98	0.87	0.0006	0.26	0.87	0.99	yes	yes	yes	yes	yes
4d_b1	d5_a6	ef_f4	haswell	SVM	1	0.93	0.0001	0.13	0.93	0.99	yes	no	no	yes	yes
4d_b1	d5_a6	ef_f4	haswell	XGBoost	0.99	0.97	0.0003	0.06	0.97	0.99	yes	yes	no	yes	yes
4d_b1	d5_a6	ef_f4	haswell	RF	0.99	0.95	0.0002	0.1	0.95	0.99	yes	no	no	yes	yes
4d_b1	d5_a6	ef_f4	haswell	HGBoost	0.99	0.97	0.0002	0.056	0.97	0.99	yes	yes	yes	yes	yes