Real-Time Edge Processing Detection of Malicious Attacks Using Machine Learning and Processor Core Events

Rob Oshana
Vice President, Software R&D
NXP Semiconductors
Austin, Texas, USA
robert.oshana@nxp.com

Mitchell A. Thornton
Darwin Deason Institute for Cybersecurity
Southern Methodist University
Dallas, Texas, USA
mitch@smu.edu

Eric C. Larson
Darwin Deason Institute for Cybersecurity
Southern Methodist University
Dallas, Texas, USA
eclarson@smu.edu

Xavier Roumegue
Senior Engineer
NXP Semiconductors
Austin, Texas, USA
xavier.roumegue@nxp.com

Abstract*—A method for the detection of the malicious events such as the SPECTRE exploit is proposed and evaluated using machine learning and processor core events. In this work, we use machine learning to implement a system based on hardware event counters to detect malicious exploits such as SPECTRE running in a process on a Linux based system. Our approach is designed to use existing on-chip hardware to detect a SPECTRE-based exploitation in real time. Prototype architectures in both x86 and ARM-based SoC's representing an embedded system with a corresponding real-time Edge-based classifier is designed and implemented to validate the approach. This exploit detection architecture uses software agents and requires no additional hardware. In particular, a software agent periodically accesses the event counter register file during runtime. At each observation time, a feature vector is formulated consisting of a particular subset of event counter data. The event counter data used in the detection technique includes cache and branch prediction counts. Various different machine learning classifiers are implemented with a goal of predicting either the presence of the malicious exploit or something other than the malicious exploit. Thus, the classifier outputs binary states of "malicious exploit present" versus "normal operation." Many classifiers resulted in true positive rates in excess of 98% with corresponding false positive rates less than 1%. In many cases, a 0% false positive rate is achieved. These predictive approaches are compared for training complexity and performance.

Keywords*—Edge Processing, malicious event, SPECTRE, embedded system, SoC, event counter, performance counter, machine learning

I. INTRODUCTION

SPECTRE is a security vulnerability that exploits speculative execution and indirect branch prediction circuitry that is common and present in most modern CPU cores. The exploit allows access to unauthorized information by implementing side channel analysis of information in the data cache of the system [1]. SPECTRE is documented in the Common Vulnerabilities and Exposures (CVE) database as CVE-2017-5717 and CVE-2017-5753. The general idea behind the attack is that the attacker uses the performance enhancement features of the processor, namely the cache and branch predictor plus speculative execution circuitry, to read higher privileged data.

In this work, we use machine learning to implement a system based on hardware event counters to detect the SPECTRE exploit running in a process on Linux system. Our approach is designed to use existing on-chip hardware to detect a SPECTRE-based exploitation in real time [2][3]. We inform the design of our machine learning models and evaluate performance by creating a dataset of 16,000 samples from an x86 as well as ARM-based system running Linux. We investigate a number of machine learning algorithms, and find that a support vector machine classifier achieves perfect performance on our collected dataset (100% detection of the SPECTRE exploit without any false positives).

We are motivated to consider the SPECTRE exploit detection problem for embedded systems and thus devise an approach that is real-time, requires no new hardware, and minimizes overall system performance degradation. For these reasons, our architecture design utilizes a classifier that is present in the cloud and that interacts with the embedded system via the use of software agents that run on the embedded system. The use of a cloud-based classifier is a design choice and is not required for our methodology to be implemented in other systems and environments as described in [9].

A diagram representing the overall architecture of our prototype implementation is shown in Fig. 1. Event counter samples are collected from the system that represent the runtime profiles of the active processes on the embedded system. In our prototype system, these samples are communicated to a laptop representing a cloud-resident processor and referred to as the "firehouse." The firehouse predicts whether or not the SPECTRE exploit, denoted as the "ghost," is present. If the firehouse predicts that a ghost is present, the process is killed in real time by another software agent running on the embedded system referred to as the "ghostbuster."

We analyze the sampled event counter data for its significance and effectiveness in the classification task. Likewise, we evaluate a variety of different machine learning-based classifiers using the sampled event counter data. Comparisons of the performance of these classifiers is carried out. We include the results of this comparison that indicates the support vector machine (SVM) classifier is the most suitable choice for implementation in the prototype. The firehouse in our prototype implements the SVM classifier although we also provide results of our comparison of different classifiers.

We evaluate the performance of this prototype real-time system. Out of 267 total actual exploits we observe that the prototype system misses only one instance of the SPECTRE exploit without any false positives. We conclude that using machine learning for detection of the SPECTRE exploit is a viable method in terms of performance for real time embedded systems.

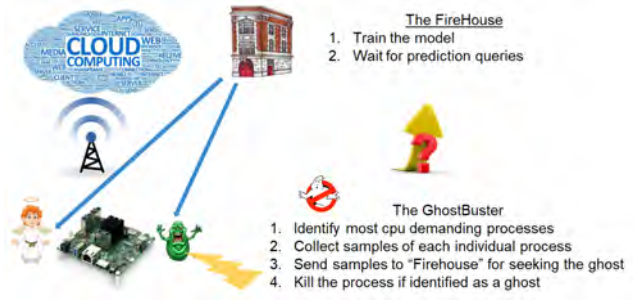

Fig 1: iMX8QM Embedded SoC used as the Exploitation Victim

II. RELATED RESEARCH

The use of hardware performance counters for the detection of malware has been investigated by other researchers. Surveys of the use of performance counters such as the event counters we use are provided in [4][5][29]. One of the first investigations of the use of performance counters resulted in the *Eunomia* prototype where malware including *code-injection*, *return-to-libc*, and *return-oriented programming* attacks were considered although machine learning classifiers were not used [6]. Later work did incorporate performance counters and machine learning classifiers to detect Android ARM malware and Intel rootkits [7]. Research that utilizes performance counters in combination with the inclusion of specialized hardware support for malware detection includes that of [2][3][8].

The use of performance counters for malware detection in terms of the required computational overhead is addressed in [9] where a "sample-locally-analyze-remotely" and "compressive sensing" approach was proposed. This allowed counter performance data to be collected on the target machine, compressed, and processed remotely resulting in decreased computational overhead for malware detection using counter data. We use similar approaches to reduce computational overhead in our approach.

A method for the detection of the Heartbleed vulnerability used support vector machines (SVM) to detect the Heartbleed vulnerability [10]. In that approach a SVM was used as a binary classifier to detect between regular and abnormal behavior and reported a 92% accuracy. This work also concluded that data-oriented attacks were more difficult to detect than control-data exploits based on their focus on buffer over-reads. An SVM is also used in [11] and a methodology is proposed for a generalized side-channel attack detection system by correlating its execution trace with a secret encryption key. As described below, we also use SVM in our prototype after considering a variety of candidate classifiers.

Some recent publications have used performance counters to detect the SPECTRE exploit [12][13][14][15] in conjunction with performance management unit (PMU) generated interrupts, however, they did not employ machine learning for classification as we do.

III. SPECTRE EXPLOITATION IN ARM BASED SYSTEMS

In this section, we provide details regarding how the SPECTRE exploit is implemented within the environment of an ARM-based system. We use these concepts to motivate the selection of processor core events that are useful in revealing the presence of the SPECTRE exploit when a machine learning algorithm is implemented as the exploit detection classifier.

On ARM-based platforms, the exploit success rate is dependent on the victim function implementation. Since the caches are relatively small (16KB for Arm/iMX and 32KB for x86 i6950), the evict rate is important and it has been proven to be preferable to avoid the load instruction in the speculative secret leak path. We designed an exploit to take advantage of the architectures of both the x86 and the iMX8 SoC. A key goal is to avoid memory accesses in the speculated path.

In order to leak a user-space secret to both the attacker and the victim, there must be processes in user space. The approach is to leak a kernel secret through a kernel victim. The attacker runs in unprivileged mode, interacting with the kernel through legal mechanisms such as syscalls. The victim runs in kernel space as illustrated in Fig. 2. We developed a dedicated and vulnerable kernel module named *"SPECTRE_victim"* for our proof-of-concept implementation. The "victim" function implementation follows the original SPECTRE publication [20], reproduced here in pseudocode form as:

```
void victim_function(size_t x) {
   if (x < *array1_size)
         {
            temp &=
           array2[array1[x] * ARRAY2_CHAR_SIZE];
         }
}
```

In this example, as long as the variable *x* is smaller than *array*1_*size*, nothing out of the ordinary happens. The code checks to make sure that *x* does not go beyond the end of *array*1, which is generally good defensive programming.

However, this type of check does not take into consideration the behavior of branch prediction and

speculative execution. Many embedded processors monitor and record how often a branch is taken and use this information to predict future behavior. In this example, if the prediction is that *x* is smaller than *array*1 *size*, then the embedded processor will speculatively execute the instruction before the condition has been evaluated.

If *array*1 *size* is not in the cache, the time to evaluate the condition will be relatively high compared to the time it takes to speculatively execute the next instruction. It is also possible to "train" the branch predictor with many valid examples of the test and array access values.

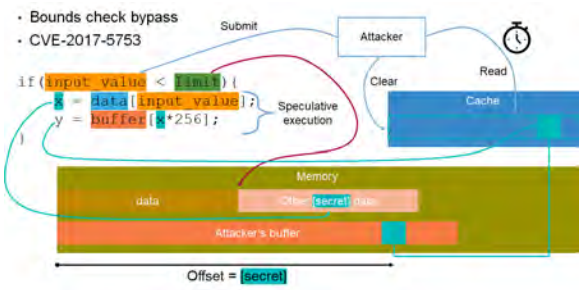

Fig 2: Diagram of the SPECTRE attack

When the condition that "is *x* is actually greater than *array*1_*size*" is evaluated, the processor will determine that it made a mistake and attempt to throw away the pre-computed computation. At this point, the content of *array*1[*x*] is located in memory beyond the end of the array and must be used to look up an element of *array*2 which is now located in the cache. The contents of *array*1[*x*] are not in the cache. However by finding which element of *array*2 is in the cache its easy to deduce *array*1[*x*]. To perform this deduction, all that is needed is to access each element of *array*2 and record the corresponding access times. In this manner, observing a faster access allows one to deduce that it is obviously the one that is present in the cache.

IV. SELECTING RELEVANT CPU EVENTS FOR DETECTION

We hypothesize that it is possible to detect the SPECTRE exploit by monitoring the CPU events on the ARM and x86 processors that represent the runtime profiles of processes that may or may not be instrumented to contain the SPECTRE exploit. In this section, we explain the types of events available for monitoring on x86 and ARM based systems and provide an explanation to justify which events are more applicable for use in detecting the SPECTRE exploit.

The ARM A72 processor contains a subsystem referred to as the Performance Monitor Unit (PMU). The purpose of the PMU is to gather various statistics characterizing the runtime profile or operation of the processor and memory system. These PMU events provide information about the behavior of the processor during runtime that can be used for purposes such as debugging and profiling software [18]. The event counter register values are accessible through system calls since it is anticipated that some system and application software may use these data for other purposes such as dynamic performance tuning. The PMU in the ARM A72 comprises six counters. Each of these six counters can count any of 84 different specified events within the processor.

For our application, we narrow this exhaustive list down to events that are associated broadly with timing, the L1 data cache, and speculative load/store executions. Six of the events were selected to train our classifier. The events were chosen based on their ability to yield appropriate side channel information that is indicative of the presence of the SPECTRE exploit, or the presence of something other than the SPECTRE exploit. For instance, since the SPECTRE attack attempts to "massively" clean the cache many times in order to fool the branch predictor and thus force the malicious code to be executed in a speculative path, we select various event counters that monitor branch prediction and cache access. The six events we selected are:

1. Event 0x11: CPU_cycles. We collect this event data in order to normalize other events by the number of active CPU cycles.
2. Event 0x12: Predictive branch speculatively executed. We hypothesize this event could yield information regarding when branch predictions are occurring and thus, when the system is vulnerable.
3. Event 0x10: Mis-predicted or not predicted branch speculatively executed. Like the previous event, we hypothesize this event could indicate when the system is vulnerable.
4. Event 0x42: L1 data cache refill read. We hypothesize that monitoring this event may reveal when the data cache is being traversed by the SPECTRE exploit.
5. Event 0x48: L1 data cache invalidate. Similar to the previous event, this event may indicate cache checking by the SPECTRE exploit.
6. Event 0x72: Operation speculatively executed: load/store. This event may indicate vulnerability of the system because of speculatively executed operations involved in branch prediction.

We collect these six hardware event counter data over time as described in the next section and analyze each of them with respect to their ability to discern the presence of the SPECTRE exploit, or the presence of something other than the SPECTRE exploit.

For the x86 architecture, we used a similar approach. The key x86 event counters used are;

1. Event 0x41: Not taken speculative and retired mis predicted macro conditional branches
2. Event 0x81: Taken speculative and retired macro-conditional branches
3. Event 0x82: Taken speculative and retired macro-conditional branch instructions excluding calls and indirects
4. Event 0x84: Taken speculative and retired indirect branches excluding calls and returns
5. Event 0x88: Taken speculative and retired indirect branches with return mnemonic
6. Event 0x90: Taken speculative and retired direct near calls

V. DATA COLLECTION

In order to inform the design of our detection algorithm, we collect data from the identified event counters during the execution of various applications. We perform data collection using a dual-core embedded SoC architecture.

A. Embedded SoC Architectures

The SoC used for this experiment is the iMX8QM from NXP Semiconductors as shown in Fig. 3. This SoC is a dual A72 core-based SoC with additional A53 cores, graphics, and video acceleration processors [19]. This device is used in many automotive infotainment applications. This device has two A72 cores and four A53 cores. For this experiment we used the two A72 cores. The two A72 cores both use a 1MByte shared instruction cache.

The event measurements are taken continuously with measurements lasting about 2.5 seconds per measurement. This is long enough to eliminate any pipeline effects when collecting the data. Each of these event counter measurements are normalized per active CPU cycle by computing the ratio of the number of events per number of active cycles.

We performed a similar experiment on the Intel i6950x processor in a Dell laptop.

B. Selected Applications for Testing

To build and evaluate our model, nine different application types were run in the user space in a Linux environment on two of the four A72 cores present within the iMX8QM in order to simplify the proof-of-concept experiment. Eight of these applications were chosen to be as representative of a broad range of common tasks as possible. The ninth application type was a process that implemented a SPECTRE attack. These applications are the system idle task, I/O operations on the network file system using **`iozone`** which is a filesystem benchmark tool that generates and measures different types of file operations, I/O operation on a sdcard using **`iozone`**, graphics operations using **`glmark2`**, an OpenGL benchmark maintained by the Linaro Graphics Working group, a TCP/IP communication using **`iperf`**, a tool for measuring bandwidth on IP networks, I/O operation on **`nfs`** using **`fio`**, a tool that spawns a number of threads that performs user specified I/O actions, **`Openssl`** crypto

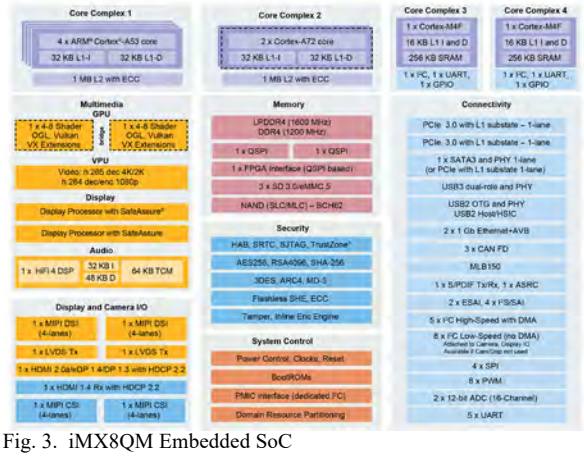

Fig. 3. iMX8QM Embedded SoC

operations that secure communications using the SSL and TLS protocols, a Linux benchmark application using **`lmbench`**, and the standard proof of concept SPECTRE attack.

The same application use cases were also run in the same use scenarios on the x86 i6950 core. Data from the event counters is collected for the nine application types mentioned above. The measurements are all started with a random delay so that each measurement sampling is different. The Linux "**`perf`**" tool is used to collect the event counter samples during runtime. Operational deployment of our method would likely use an interrupt-driven background process rather than a polling approach.

Each application is chosen at random for running on the test system and the selected application is logged along with its runtime on the system. The log enables each process to be labeled so that the supervised learning model can be trained. The system is allowed to run for approximately 12 hours in duration, resulting in 16,000 labeled measurements. A script is used to post process the 16,000 labeled samples of raw data to build the dataset for later processing.

C. Test Environment

An embedded processor is infected with the SPECTRE exploitation code along with instantiating the other application types. Internal SoC event data is collected and used to train the model on the laptop acting as the firehouse in a cloud-based system. When running the system, prediction queries from the cores (iMX8 ARM based SoC and x86 i6960) are collected. Data are collected showing the most demanding CPU processes running on the cores. Samples are collected from each of the individual processes running on the two cores of the SoC.

VI. ANALYSIS OF COLLECTED DATA

A. Preprocessing and Cross Validation

In the preliminary portion of the investigation reported here, prediction models were created using a variety of machine learning techniques. Models are trained using a large amount of data gathered and processed from an experimental environment. It was hypothesized that the processor event data, such as that provided in Table 1, could be used to form a feature vector that differentiates between the binary machine states of "normal operation" versus "SPECTRE" exploitation."

We preprocess the event counter data by applying normalization to the dataset by making each feature has zero mean and unit standard deviation. This standardization is common for machine learning algorithms that employ linear models and can help to speed convergence as well as prevent one feature from dominating the decision boundary [20].

The data collected for this experiment was used to create a function that maps an input to an output based on example input-output pairings. This is a classic case of a supervised learning approach, since it is possible to annotate the collected dataset with responses. The supervised machine learning approach is attempting to determine a classification of "SPECTRE attack present" or "something other than SPECTRE attack present," so a "classification" approach is used in this experiment. Each classifier assigns new examples (core events) to one category or the other (SPECTRE or no SPECTRE).

In order to determine which type of classification model to performs the best, we analyzed several machine learning models using a process called "forward chaining" cross validation. Forward chaining cross validation is a statistical technique for estimating the robustness of machine learning model performance with training and testing that are appropriately selected for time series data. As shown in Fig. 4, this method selects data for training a machine learning model from a contiguous time block of samples in the dataset. A smaller testing set is chosen from a contiguous block of data occurring after the training data. This process is repeated multiple times using a longer block of contiguous training data and a test set that occurs even further in time. This cross validation ensures that the analysis does not violate any time boundaries. For example, it would not be proper to evaluate the algorithm on test data when training data was sampled both before and after the testing data blocks. Likewise, it would not be proper to select testing data and training data that occur temporally close in time.

B. Selected Evaluation Criteria

Once the event counter data was collected, forward chaining cross validation (with four splits of the dataset as shown in Fig. 4) was used to assess performance of several machine learning algorithms for the SPECTRE event counter data. We chose to evaluate each model using "recall," "precision," and "F1" (which is a metric that combines the information given by recall and precision). The formulas for these metrics are given in (1) through (3).

$$Recall = \frac{TP}{\left(TP + FN\right)} \tag{1}$$

$$Precision = \frac{TP}{\left(TP + FP\right)} \tag{2}$$

$$F1 = \frac{2\left(Precision \times Recall\right)}{\left(Precision + Recall\right)} \tag{3}$$

Recall or *Sensitivity*, measures the proportion of times the classifier predicted the presence of the SPECTRE exploit with respect to the number of times it was truly present. Thus, a perfect Recall would be 100% and would occur when all the exploits were correctly predicted and when no predictions occurred indicating the exploit was present when in fact it was not. In a more general sense, Recall provides a quantitative answer to the question "How many relevant items are selected?"

Precision is a measure of the proportion of times that the classifier correctly predicted the presence of the SPECTRE exploit with respect to the total number of times it predicted the exploit was present whether they were correct or not. Thus, a perfect accuracy of 100% would result if the classifier always predicted when the exploit was truly present and never predicted that the exploit was present when in fact, it was not present. In general, *Precision* answers the question: "How many selected items are relevant?"

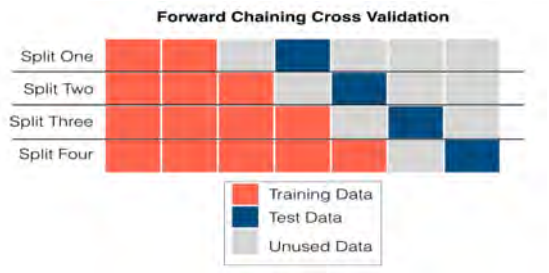

Fig 4: Forward chaining cross validation example showing training and test data separation for four splits.

*F*1 is an overall measurement of a classifier's effectiveness and considers both *Recall* and *Precision*. At a *F*1 score of 1, the system is performing ideally with respect to *Recall* and *Precision*.

C. Performance Across Machine Learning Algorithms

We chose a variety of different machine learning algorithms that could be implemented in the prototype system. Fig. 5 shows the cross-validation results of several machine learning classifiers including Decision Trees (DT) using **`gini`** index and no pruning applied, Gaussian Naïve Bayes (GaussNB), Random Forests (RF) with 100 trees, *K*-nearest neighbors (*K*NN) with *K*=3 and Euclidean distance, Support Vector Machines (SVM) with radial basis function kernel, *C*=1, and gamma=0.001 and Multilayer Perceptron, with one hidden layer of ten neurons (MLP-10) and sigmoid activation functions.

All classifiers were implemented using the open-source python machine learning toolkit, **`scikit-learn`** [24]. The Support Vector Classifier (SVC) was trained using **`LIBSVM`**, an open-source library for SVMs. Unless stated otherwise, all other hyper-parameters of the machine learning algorithms were chosen to be default parameters.

The bar chart in Fig. 5 shows the mean *recall*, *precision* and *F*1-score averaged across all testing folds. Error bars are also shown that indicate the "95% prediction interval" of all the folds in the dataset. The prediction interval is defined as $\mu \pm 1.96\sigma$ for each metric, calculated from the four train/test separations. Notice that all machine learning methods perform well, as the vertical axis of Fig. 5 is zoomed, ranging from 0.98 up to a perfect score of 1.00. Two methods perform perfectly on the training and test sets: *K*NN and SVM.

We prefer the SVM as compared to *K*NN because of the time it takes to predict when the model is deployed. *K*NN involves an exhaustive search over the training data for every prediction, which can be time consuming for a deployed machine learning algorithm with real time constraints. In accordance with these results, SVM is selected as the machine learning model to use for further analysis. A SVM trained with a radial basis function is optimized by maximizing a decision boundary margin in a large dimensional space [22] [23] [24]. Because we use a radial basis function kernel, the theoretical dimensional space is infinite. This means the SVM

can achieve arbitrary decision boundaries based on the feature data.

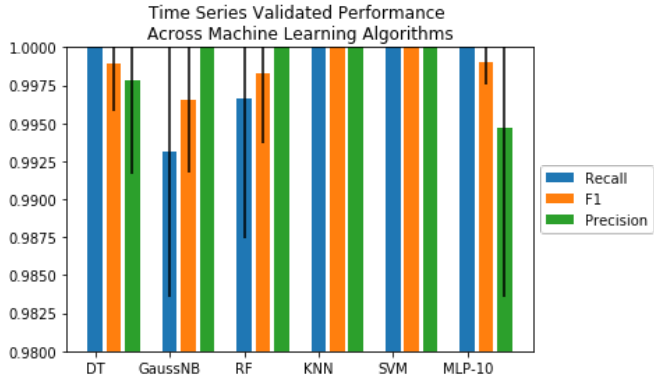

Fig. 5; Time series validated performance of machine learning models based on *K*-fold cross validation

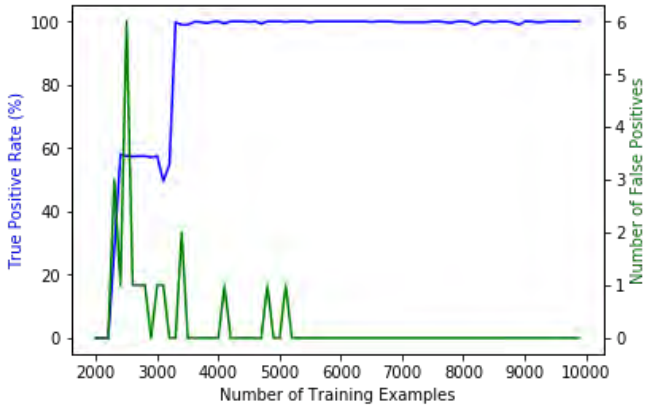

Fig. 6: False positives and true positives versus test sample size for SVM classifier.

D. Amount of Training Data Analysis

While an SVM is able to provide superior performance using cross validation, we wanted to understand the performance of the SVM as a function of the amount of training data provided. We analyzed the event counter data using SVM with increasing amounts of training data from our 16,000 instance dataset.

We used contiguous train and test separation options and performed an analysis on the amount of training examples required to achieve perfect true positive rate without any false positives (Fig. 6). Fig. 6 shows a split vertical axis with the percentage of positive SPECTRE exploits found on the left axis and the total number of false positives on the right axis. The horizontal axis shows the amount of data points in the training set, ranging from 2,000 up to 10,000 instances. As shown, using about 6000 data points for training achieves excellent true positive rate, without any false positives.

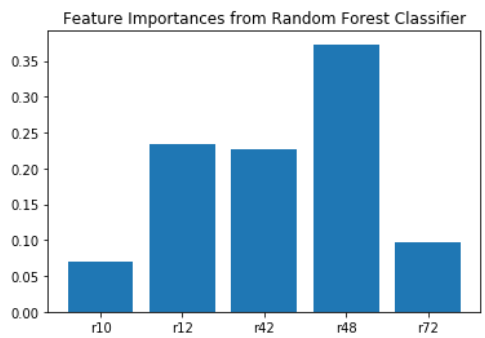

Fig. 8: Feature Importance from the Random Forest Classifier

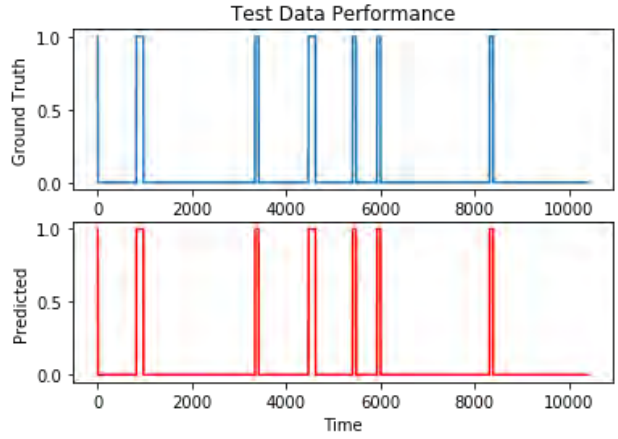

Fig. 7 Ground truth versus predicted test data comparison

Fig. 7 shows the results of the analysis on test data for the SVM when the model is trained with 6,000 data points. The output of the classifier is shown over approximately 7.5 hours, along with ground truth for when the SPECTRE exploit was active. As shown in Fig. 7, the predictions correlate perfectly with ground truth. Each of the seven times that the SPECTRE exploit is active, the SVM almost immediately detects it.

E. Relevant Feature Analysis

We also performed an analysis of the feature importance in the training data. For this analysis, we chose to analyze the feature importance as identified using a random forest (RF) classifier. We chose the RF classifier for feature importance analysis because it has been shown to be highly consistent and has less bias than other methods [25]. To investigate importance, we use the feature permutation method as described in [25]. Intuitively, one can think of this method as measuring importance by randomly permuting each feature into the random forest and observing the degradation in performance from this permutation. Highly relevant features

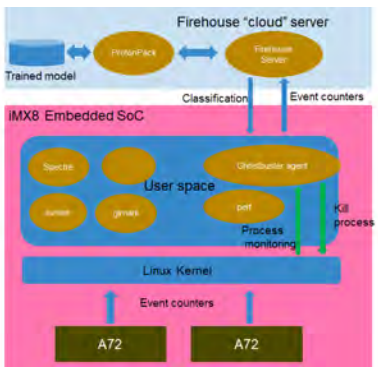

Fig. 9: Software Architecture for SPECTRE proof of concept

result in large performance degradations.

Fig. 8 shows the relative importance of features for this method (higher indicated more importance). In this case, the features are the normalized event counter data. As observed in Fig. 8, the "data cache invalidate" (r48) event counter is a dominant feature in this experiment, as well as "data cache refill read "(r12) and "Predictive branch speculatively executed " (r42). We note that none of the features have an importance near zero indicating that all features contribute to the performance of the classifier. Therefore all features are used in further analysis.

IIX. REAL-TIME SYSTEM ARCHITECTURE

Given the performance of the SVM on our collected dataset, we implement a prototype real time SPECTRE exploit detection system using this model. The software architecture for the real-time system is shown in Fig. 9. The perf process is used to collect the hardware event counter data. These samples are sent to the laptop (Firehouse) to identify the SPECTRE exploit (ghost). If identified, the process is killed in real time. We therefore refer to the process as Ghostbuster. We developed the Firehouse to run on a cloud machine and the GhostBuster process was responsible for detecting the SPECTRE attacks on the embedded iMX8 device. GhostBuster collects samples from the applications running on iMX and sends them to the Firehouse server requesting a classification. If Firehouse predicts that there is a SPECTRE attack running within a process, it will kill or suspend the process.

`ProtonPack` is the server class responsible for machine learning training and classification. This class can be easily inherited for further experimentation. In order to demonstrate the SPECTRE attack, a patch was added to route the PMU counters to the ARM A72 cores instead of the A53 cores. A SPECTRE victim kernel module, which is the SPECTRE gadget module designed to have a vulnerable driver, is implemented in the kernel and is used to demonstrate the SPECTRE attack against the kernel space. Additional kernel modifications were made to grant access to the PMU counters from user space since the SPECTRE proof of concept uses the PMU counters as a timing measurement method.

IX. EXPERIMENTAL RESULTS

Testing was conducted on two machines, a laptop running Linux 4.14 and an iMX8QM running Linux 4.14 built as an embedded **`Yocto`** distribution. Specifically, the Linux laptop machine is a Dell Latitude with a 1.3GHz Intel® i5 processor and 4GB of main memory and the iMX8 SoC is a dual ARM A72 processor running at 1.6 GHz with 1Mbyte of shared I2 memory.

`ProtonPack` is used to train the model and determine model performance. A test suite is created that consists of a total of 4,105 trials running for about 2.5 hours. During the 2.5 hour runtime, the SPECTRE exploit was randomly made active 227 times. Other applications were also run during the 2.5 hour runtime. In many circumstances, another application was running when the SPECTRE exploit was activated in an attempt to provide a realistic environment.

A confusion matrix is computed as one of the means to determine the performance of our classification model. The confusion matrix, or error matrix, allows a determination of the performance of a supervised learning algorithm such as ours that predicts a binary classification and is provided in Table I. Values in Table I are in the format "percentage (absolute number)." The confusion matrix contains four key values; the number of times a predicted presence of the SPECTRE attack was correct or not, and the number of times a predicted non-presence of the SPECTRE attack was present or not. In Table I, the topmost sub-table is a key for interpreting the measured values in the bottommost sub-table. Data is shown for both Arm and x86 ISA's.

TABLE I. TABLE I. CONFUSION MATRIX FOR SPECTRE EXPERIMENT

	P (predicted)	**N (predicted)**
P (actual)	*TP*	*FN*
N (actual)	*FP*	*TN*

Arm	**P (predicted)**	**N (predicted)**
P (actual)	94.47% (3878)	0.024% (1)
N (actual)	0	5.51% (226)

X86	**P (predicted)**	**N (predicted)**
P (actual)	86.16% (31784)	0.01% (6)
N (actual)	0.01% (6)	13.80% (5093)

In the topmost part of Table II, the legend for the bottommost part, we use *TP* to denote "true positive," the number of observations where the SPECTRE attack was present and the SVM prediction was correct. *TN* denotes "true negative" and is the number of times the SVM classified the exploit as being present when it actually was not present. *FP* denotes "false positive," and is the number of times the SVM predicted that the SPECTRE exploit was present when it actually was not present. Finally, *FN* denotes "false negative," the number of times the classifier did not predict the exploit was present when it actually was present. In the entire experiment, *FN* occurred only once out of the 227 SPECTRE exploits experimented.

While the confusion matrix is a fundamental measure of the performance of a binary prediction or classification implementation in an experimental environment, we also report the *recall*, *precision*, and *F*1 score for the real time experiment in Table II.

TABLE II. TABLE II. RECALL, PRECISION, AND F1 FOR 4105 TRIALS

METRIC	**VALUE**
Recall	99.97%
Precision	100%
F1	99.98%

Because our experiments were conducted under the framework of a supervised machine learning model, the choice of the training data is a crucial aspect of the method. It is important to craft a learning phase that is capable of characterizing SPECTRE payload behavior even when the actual exploit may be in the form of a zero-day exploit.

X. CONCLUSIONS

A method for the detection of SPECTRE is devised and experimentally verified. The technique is based upon monitoring on-board, hardware event counters rather than characteristics of the targeted data. The technique requires a minimal amount of modification to hosting computer systems since it uses pre-existing event counters and supporting circuitry and associated system software assets with no additional hardware required. The disclosed SPECTRE detection technique has been experimentally shown to be effective for Linux based operating systems in a real time embedded system based on our prototype implementation and associated experimental results.

Our experimental results use a polling or sampling method where the operational phase of the detection method would periodically query the event counters to obtain recent values. This approach may suffer from potential aliasing problems if the SPECTRE exploit were to be implemented in a manner such that it occurred between adjacent event counter polling observation times. In the future, we intend to investigate the use of an interrupt-driven technique whereby event counter readings are sampled on demand. Additionally, the interrupt-driven approach should also reduce the computational overhead.

REFERENCES

[1] Thomas M. Conte, Erik P. DeBenedictis, Avi Mendelson, and Dejan Milojičić, "Computers to Avoid SPECTRE and Meltdown," *IEEE Computer* , vol. 51, iss. 4, April 2018.

[2] Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S., "On the Feasibility of Online Malware Detection with Performance Counters," in proc. *40th Annual Int. Symposium on Comp. Arch,* pp. 559-570, June 2013.

[3] Tang, A., Sethumadhavan, S., and Stolfo, S., "Unsupervised Anomaly-based Malware Detection using Hardware Features," in proc. *Int. Symposium on Research in Attacks, Intrusions, and Defenses,* pp. 109-129. Sept. 2014.

[4] Foreman, J.C., "A Survey of Cyber Security Countermeasures Using Hardware Performance Counters," *arXiv:1807.10868v1* [cs.CR], July 28, 2018.

[5] Das, S., Werner, J., Antonakakis, M., Polychronakis, M., and Monrose, F., "SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security, in proc. *IEEE Symp. on Security and Privacy*, 2019.

[6] Yuan, L., Xing, W., Chen, H., and Zang, B., "Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks using Performance Counters" in proc. *ACM SIGOPS Asia-Pacific Workshop on Systems* (APSys), pp. 6.1 – 6.5, July 2011.

[7] Ozsoy, M., Donovick, C., Gorelick, I., Abu-Ghazaleh, N., and Ponomov, D., "Malware-Aware Processors: A Framework for Efficient Online Malware Detection," in proc. *IEEE Int. Symp. on High Performance Computer Architecture* (HPCA), pp. 651 - 661, 2015.

[8] Kazdagli, M., Reddi, V.J., and Tiwari, M., "Quantifying and Improving the Efficiency of Hardware-based Mobile Malware Detectors," in proc. *IEEE/ACM Int. Symp. on Microarchitecture* (MICRO), pp. 1 - 13, 2016.

[9] Wang, X., Chai, S., Isnardi, M., Lim, S., and Karri, R., "Hardware Performance Counter-Based Malware Identification and Detection with Adaptive Compressive Sensing," *ACM Trans. on Architecture and Code Optimization*, vol. 13, no. 1, art. 3, March 2016.

[10] Torres, G. and Liu, C., "Can Data-Only Exploits be Detected at Runtime Using Hardware Events?: A Case Study of the Heartbleed Vulnerability," in proc. *Hardware and Architectural Support for Security and Privacy* (HASP), Article 2, 2016.

[11] Alam, M., Bhattacharya, S., Mukhopadhyay, D., and Bhattacharya, S., "Performance Counters to Rescue: A Machine Learning based safeguard against Micro-architectural Side-Channel Attacks," in proc. *Cryptology ePrint Archive*, Report 2017/564, 2017.

[12] Pierce, C., "Detecting Spectre and Meltdown using Hardware Performance Counters," online *EndGame BLOG*, January 2018.

[13] Herath, N. and Fogh, A., "These are Not Your Grand Daddy's CPU Performance Counters," in proc. *BlackHat Conf.*, 2015.

[14] Pierce, C., Spisak, M., and Fitch, K., "Capturing 0day Exploits with PERFectly Placed Hardware Traps," in proc. *BlackHat Conf.*, 7 pages, 2016.

[15] Depoix, J. and Altmeyer, P., "Detecting Spectre Attacks by Identifying Cache Side-Channel Attacks using Machine Learning," in proc. 4th *Wiesbaden Workshop on Advanced Microkernel Operating Systems* (WAMOS), pp. 75 - 86, August 2018.

[16] Prakhar Kaushik and Rana Majumdar, "Timing attack analysis on AES on modern processors," in proc. *2017 6th International Conference on Reliability, Infocom Technologies and Optimization* (ICRITO), Sept. 20 – 22, 2017.

[17] Conte, T. M., DeBenedictis, E. P., Mendelson, A., and Milojičić, D., "Computers to Avoid SPECTRE and Meltdown," *IEEE Computer* , vol. 51, iss. 4, April 2018.

[18] ARM Ltd, Technical Reference Manual ARM® Cortex®-A72 MPCore Processor, Revision: r0p3,chapter 11.

[19] iMX 8 Applications Processors Family Fact Sheet, *NXP Semiconductors*, September 2016.

[20] Hao, Z., Liu, B.,Yang, X.-W., A Comparison of Multiclass Support Vector Machine Algorithms, in proc. *Int. Conf. on Machine Learning and Cybernetics*, 2006, Pages 4221-4226.

[21] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. and Vanderplas, J., "Scikit-learn: Machine learning in Python," *Journal of machine learning research*, Oct. 12, pp. 2825 - 2830, 2011.

[22] Kajale, R., Das, S., and Medhekar, P., "Supervised machine learning in intelligent character recognition of handwritten and printed nameplate," in proc. *2017 Int. Conf. on Advances in Computing, Communication and Control* (ICAC3), December 2017.

[23] Saha, S.S., Siraj, M.S., and Habib, W.B., "FoodAlytics: A formal detection system incorporating a supervised learning approach," in proc. *2017 IEEE Region 10 Humanitarian Technology Conference* (R10-HTC), December 2017.

[24] Ma, S. and Ji, C. "Performance and efficiency: recent advances in supervised learning," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1519-1535, 1999.

[25] Breiman, L., 2001. Random forests. *Machine learning*, *45*(1), pp.5-3