Detecting and Preventing ROP Attacks using Machine Learning on ARM

Gebrehiwet B. Welearegai
Faculty of Informatics and Mathematics, University of Passau
Passau, Germany
gebrehiwet.welearegai@uni-passau.de

Chenpo Hu
Munich, Germany
chenpo.charlotte.hu@gmail.com

Christian Hammer
Faculty of Informatics and Mathematics, University of Passau
Passau, Germany
Christian.Hammer@uni-passau.de

***Abstract*—As the ARM processor is receiving increased attention due to the fast growth of mobile technologies and the internet-of-things (IoT), it is simultaneously becoming the target of several control flow attacks such as return-oriented programming (ROP), which uses code present in the software system in order to exploit memory bugs. While some research can detect control flow attacks on architectures like x86, the ARM architecture has been neglected. In this paper, we investigate whether ROP attack detection and prevention based on hardware performance counters (HPC) and machine learning can be effectively transferred to the ARM architecture. Given the observation that ROP attacks exhibit different micro-architectural events compared to benign executions of a software, we evaluate whether and which HPCs, which track these hardware events, are indicative on ARM to detect control flow attacks. We collect data exploiting real-world vulnerable applications running on ARM-based Raspberry Pi machines. The collected data then serves as training data for different machine learning techniques. We also implement an online monitor consisting of a modified program loader, kernel module and a classifier, which labels a program's execution as benign or under attack, and stops its execution once the latter is detected. An evaluation of our approach provides detection accuracy of 92% for the offline training and 75% for the online monitoring, which demonstrates that variations in the HPCs are indicative of attacks on ARM architectures. The performance overhead of online monitoring evaluated on 8 real-world vulnerable applications exhibits a moderate 6.2% slowdown on average. The result of our evaluation indicates that the behavioral changes in micro-architectural events of the ARM platform can play a vital role in detecting memory attacks.**

***Index Terms*—ROP Detection, ARM, HPC, Machine Learning, Online Monitor**

I. Introduction

The Advanced RISC Machine (ARM) processor is a modern processor being widely used in many everyday devices such as smartphones, thermostats, refrigerators, and smartwatches. ARM claims that more than 200 billion ARM processors have been shipped by 2021[1]. Due to the fast growth of mobile technologies and the internet-of-things (IoT) [31], ARM is becoming a more appealing target for control flow attacks aiming to acquire the capability to control a system. A popular method to that end is code injection, where an attacker exploits memory bugs as to maliciously altering the program's behavior or even taking full control over a system. Memory exploitation can be done by writing new machine code into the vulnerable program's memory or by reusing existing code. The latter is imperative when a protection technique known as W $\oplus$ X [1] is applied, which stipulates that memory is either writable or executable (but not both). Return-into-libc (RILC) [33] is a relatively simple code-reuse attack where a call stack is manipulated such that control is transferred to the beginning of an existing libc function, such as *system()*. For maximum expressiveness [15], return-oriented programming (ROP) [6] was introduced, which exploits a software vulnerability by chaining existing *gadgets* (small snippets of code ending in a return opcode) together in arbitrary ways. Moreover, Checkoway et. al. [8] show that ROP attacks can be mounted even without using return instructions, on both the x86 and ARM architectures.

To detect and protect against code-reuse attacks on ARM (e.g., ROP and its sibling *jump oriented programming* (JOP) [4]) some techniques have been proposed that try to enforce control flow integrity (CFI) via dynamic binary instrumentation (DBI) [17], [28] or the ARM CoreSight debugger [21], [22], [23], [25], [24], supplemented with *meta-data* collected by static analysis. Most of them rely on a *shadow call stack (SCS)* [27] for stateful backward edge protection (i.e, to detect ROP), while using a range of different static forward-edge policies such as *branch-table* (generated from CFG) and *branch regulation* (BR) [19] (i.e., to detect JOP). However, the techniques that use dynamic instrumentation suffer from high performance overhead while those that use the ARM CoreSight debugger suffer from high storage overhead. Besides, the hardware monitor that uses the hardware debugger could drop traces given a sufficiently high branch rate since the monitor requires more time to process a trace than the rate at which branches occur on the target processor [12]. Another limitation of using debugger traces to detect CRA attacks is that the hardware debugger can be used by an attacker to circumvent the security of the system. If the attacker can access the debug interface, he could use it to tamper with code and data memory, or even disable the hardware monitor by tampering with the tracing mechanism [12]. Therefore investigating whether another line of defense can detect ROP attacks on ARM accurately and precisely with low performance and storage overhead is advisable.

[1]https://www.arm.com/blogs/blueprint/200bn-arm-chips

The ARM processor provides hundreds of hardware events related to instructions that can be monitored during process execution using hardware performance counters (HPC) [10]. On x86 there exist research that tries to detect ROP attacks by investigating the characteristics of hardware events during an attack using HPCs and machine learning [29], [13]. However, to the best of our knowledge there is no corresponding research for ARM. Hence it is important to investigate whether such a technique can detect code reuse attacks, such as ROP and JOP, on ARM-based applications.

In this paper, we evaluate the suitability of a combination of HPC and machine learning techniques to detect and prevent ROP and JOP attacks on the ARM platform. Note that the myriad of HPC events on ARM differ from x86, as well as the execution model (e.g., there is no dedicated return opcode on ARM). The HPCs count the occurrence of certain hardware events on the ARM processor when executing a program, but it has not been investigated whether the events for normal and ROP attack executions differ significantly enough to enable automatic detection. In this paper, we create a machine learning model of the behavior on ARM-based Raspberry Pi machines to address this question empirically.

Our machine learning approach computes models for *runtime monitoring*. The *offline training* examines several machine learning techniques and generates a set of classification models from HPC training data collected during benign executions and attacks. To that end, we developed a novel tracer that commences recording of HPC events in executions with ROP attacks only when this attack actually starts (i.e., the first gadget of the exploit executes), which improves the classifier's accuracy by 12% over recording the program's complete execution as previous work. The *runtime monitor* contains a *modified program loader*, a *kernel module* and a *classifier*. The program loader configures the CPU using the tool *perf* as to track the set of HPCs required for the trained classifier, the kernel module computes the delta of these HPCs each time an interrupt occurs and feeds these values to the machine learning-based classifier, which labels the recent program execution as an attack or benign.

To obtain an optimal classification model our approach trains models using multiple machine learning approaches. Of eight machine learning techniques examined, the optimal classification model trained – SVM with a RBF kernel – displays a 92% and 91% accuracy for Raspberry Pi 4 and Pi 3, respectively. Leveraging this optimal classifier we evaluate ROP attack detection via runtime monitoring on Raspberry Pi using 15 exploits (based on four ROP attack variants) of real-world vulnerable applications. The detection of these attacks at runtime provides 75% accuracy, and we will elaborate on possible technical reasons for this difference.

Finally, we compared the detection accuracy of ROP vs JOP as well as Raspberry Pi 3 vs. Pi 4 (using dedicated models for each processor), which only yield insignificant differences when using dedicated models for each type. The latter is in sync with x86, where even switching inside the same processor family resulted in a significant decline of the detection rate [29].

In summary, the major contributions of this paper revolve around investigating how well a promising line of defense against code-reuse attacks on the x86 platform transfers to the ARM platform:

- In order to evaluate how well control-flow attacks can be detected on the ARM platform using HPCs and machine learning techniques we implemented a runtime monitor containing a modified program loader, kernel module and a classifier implemented in the kernel space to synchronize with the unmaskable kernel interrupts that trigger HPC reading.
- A novel debugger (tracer) that selectively records the actual attack section of a program subject to a control flow attack, in order to improve the classification model during the offline training.
- Compilation of a benchmark of 15 exploits (of four ROP variants) for the ARM platform (i.e., Raspberry Pi) from 8 real-world vulnerable applications. This benchmark is leveraged for offline training and online monitoring. Given existing ROP exploits are predominantly for x86 processors generating exploits for ARM processors is a complex, mostly manual task.
- A comparison of eight machine learning techniques to identify the optimal classification model.
- An evaluation of the ROP attack detection's accuracy and performance overhead considering various evaluation criteria.

II. Background

A. Return-Oriented Programming on ARM

A ROP[2] attack is a code-reuse attack in which an adversary generally leverages a buffer overflow to overwrite parts of the stack in order to divert the program's control flow to existing executable code sections of the program. The core idea of ROP on ARM is to exploit the presence of gadgets (small instruction sequences) that induce some well-defined behavior, such as returning using POP or branching using BLX instructions [11]. Figure 1 presents two gadgets, one ending in POP and another ending in BLX. In the first gadget the first instruction moves the value in register *r4* to register *r0* (which is used as the first argument of a function call), subsequently the values on the top of the stack are popped to registers *r4* and *pc*, which alters control flow depending on the value loaded to *pc*. Similarly, the instruction *blx r3* in the second gadget changes the flow of the program to the address in *r3*. Selecting such gadgets (e.g., with the help of gadget discovery tools such as ROPgadget[3]) and chaining them together properly, an adversary can build complex exploits to induce arbitrary behavior in the target program to a malicious end.

[2]In the sequel we will use the term ROP for all gadget-based code-reuse attacks.

[3]https://github.com/JonathanSalwan/ROPgadget

```
mov r0, r4;          mov r0, r7 ;
pop {r4, pc};        blx r3;
```

(a) pop based gadget (b) branch based gadget

Fig. 1: Gadget examples on the ARM platform

1) ROP variants on ARM: According to the differences in the (control-flow manipulating) last instruction, ROP attacks can be classified into *ret-based* ROP and *jmp-based* ROP or jump-oriented programming (JOP). Since ARM does not provide a *ret* opcode, the POP instruction, which moves a return address from the stack into the pc is used instead, i.e., gadgets ending in *pop(...,pc)* can be used to perform a ROP attack. Conversely, JOP uses gadgets ending in a *blx r* instruction, where *r* represents a general-purpose register that stores a gadget's address. Figure 1 contains POP-based and branch (BLX) based gadgets for ROP and JOP attack variants, respectively. The comparison of ROP and JOP attacks on the x86 platform was initially presented by Bletsch et al. [5]. For the ARM platform, we have designed our own model, which is presented in Section III-A

B. Hardware Performance Counters (HPCs)

HPCs, which have been available in modern processors (such as ARM, AMD, and Intel) for more than a decade [10], monitor and measure events that occur at the CPU level during process execution related to instructions. In order to obtain HPC information, initially the HPCs must be configured according to the events of interest. Then, polling or sampling can be used to read the HPC values at runtime [10]. When polling is used the HPCs can be read at any instant whereas for event-based sampling the occurrence of events triggers reading HPCs.

Though the initial purpose of HPCs was for debugging, they have also been used in several other applications, such as vulnerability research [29]. Profiling tools, such as Linux's perf[4], allow HPC data to be obtained using several methods, but that flexibility comes at the expense of yielding different counter values for the same application due to the multi-process environment and the non-determinism of HPCs [10].

C. Machine Learning

Machine learning is a collection of methods to automate data-driven model building and programming through a systematic discovery of statistically significant patterns in available data [2]. Machine learning algorithms build a mathematical model based on sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to do so. Machine learning algorithms are used in a wide variety of applications, such as email filtering, malware detection and other similar tasks that cannot be easily solved using conventional algorithms. A selection of these algorithms is presented in Appendix A

[4]https://en.wikipedia.org/wiki/Perf_(Linux)

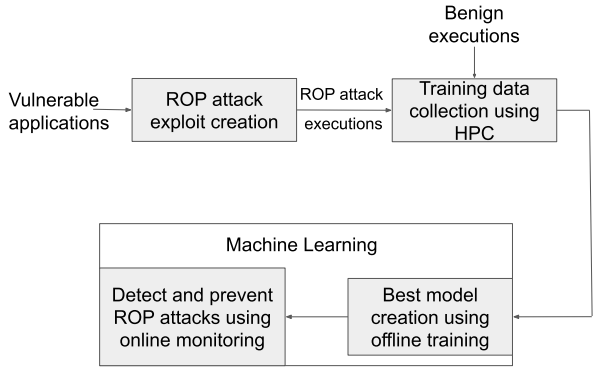


Fig. 2: Our approach to detect and prevent ROP attacks

III. Approach

The approach we use to detect and prevent ROP attacks on ARM platforms is based on learning the behavior of micro-architectural events in the CPU, combining HPC readings and machine learning techniques. Figure 2 shows our approach to detect and prevent ROP attacks. It starts with creating exploits for different real-world vulnerable applications on the ARM platform. Subsequently, we collect and pre-process the required training data by profiling both ROP attack and benign executions and reading the HPC values using the kernel's *perf* event profiler. Next, we apply machine learning techniques to train a set of models that detect and prevent ROP attacks in terms of binary classification of the programs' executions into *benign* or *under attack/ROP* based on the data collected from the HPCs. The offline training phase determines a combination of events that characterize ROP and, benign executions. After finding the model that provides the best combination of HPC events for attack detection, we use our online monitoring technique to detect and prevent the ROP attacks in a real-time execution setting.

At a high level we re-evaluate the approach of HadROP [29] and EigenROP [13] in the sense that we also leverage HPCs and machine learning to detect ROP attacks. However, we are investigating this approach's applicability to the ARM platform (with its growing prevalence) due to its peculiar instruction set (no explicit *ret* instruction) and dedicated set of HPC events. Moreover, we are training several machine learning classifiers and choosing the optimal algorithm based on its performance. In addition, we selectively record the actual micro-architectural events of the ROP attack section only, rather than of the entire ROP attack execution, which increases our classifier's accuracy from 80% to 92%. Finally, unlike EigenROP our machine learning approach follows supervised learning and the online monitoring section also differs from HadROP's approach in its program loader and classier implementation. All the steps of our approach are explained in detail below.

A. ROP exploit creation on ARM

In this section, we present how to create ROP exploits for vulnerable *real-world applications* on the ARM platform, as a sufficient set of ROP-affected malicious executions is required to train a stable classifier. Although ROP attacks on ARM are not a new idea, exploits of real-world vulnerable applications are hard to find. The first challenge entails installing and compiling vulnerable applications originally developed for x86 on the ARM platform and reproducing exploits known from x86, i.e. finding proper ARM gadgets and chaining these gadgets together into an ARM-specific exploit. The existing ROP detection approaches [21], [22], [23], [25], [24] for ARM use not more than five (3 ROP and 2 JOP) simple example attacks based on shellcodes provided by shell-storm[5], which exploit programs that are by no means real-world.

In general, the exploit creation process for ARM is challenging as we need to customize an available ROP attack for a different platform. Hence, we first need to locate the vulnerable code in the program, i.e., the size of the buffer vulnerable to buffer overflow which determines the position of the return address on the stack succeeding the buffer. Then, we use ROPgadget to identify gadgets that can be chained to perform the ROP attack types we need. Finally, we create the ROP exploit (payload) by exhausting the buffer with random values (e.g., "AAA...") until we reach the return address and overwriting the return address with the address of the first gadget. The subsequent values (usually return addresses of further gadgets) must be carefully selected such that the gadgets are chained in the order given by ROPgadget to accomplish the intended attack.

Moreover, to increase the diversity of the training and testing data set, we implemented several ROP attack variants (Ret2ZP [18], JOP [4], Ret2mP [35] and Stack pivoting [30]), which, based on the types of gadgets used to create gadgets chains, can be generalized into ROP (gadget chains ending in POP) and JOP (gadget chains ending in BLX) attacks. More generally, Figure 3 shows how we modeled the ROP and JOP attack exploits creation on ARM platforms using *pop-based* and *blx-based* gadgets. In ROP, gadget addresses are loaded into the program counter (PC) register using POP. In JOP, control flow (CF) is driven using a special dispatcher gadget that executes the gadget chain. A register that points into the gadget address list is used as the virtual program counter. In both variants, to provide arguments to a function, the contents of function argument registers (i.e., r0-r3) must be assigned before CF is redirected to the desired function. For instance, if we want to open the system's shell the register r0 must point to the address of *"/bin/sh"* before CF is directed to the address of the *system* function. Overall, we developed 15 exploits (8 ROP and 7 JOP) attacking 8 real-world vulnerable applications, which we have made publicly available for research on Github[6].

[5]http://www.shell-storm.org/

[6]https://github.com/ghiwet/ARM-ROP-exploits-benchmark

B. Data Collection on Arm using HPCs

Data collection on the ARM processor is a most challenging process since there is no tool available that can directly and continuously collect and store the relevant data separately from the program to be executed. So we had to modify existing profiling tools such as *perf* and the Linux kernel module on ARM to enable recording of HPC data. Furthermore, we developed a program that traces the ROP program and records only the actual ROP section, i.e., starts just before the execution of the first ROP gadget.

To record HPC data via *perf* the interrupt handlers of the *performance monitoring unit* (PMU) in the ARM processor must be modified. The interrupt handlers then regularly poll the HPC counters, which contain the frequencies of the hardware events since the previous interrupt, as attributes for model training. We leverage the kernel message log (*printk()*) to store that data for offline training.

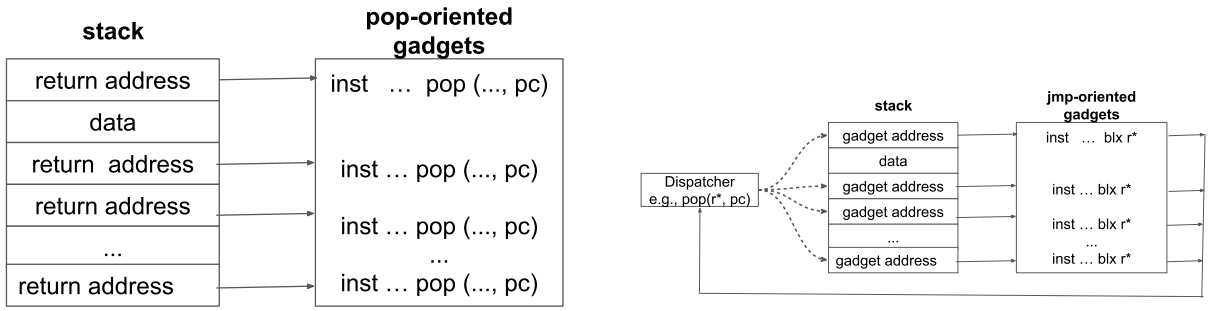
1) ***Tracer tool to record only the ROP part****:* During a ROP attack, the stack is overwritten by the adversary with a chain of gadgets pointing to existing executable binary code. However, the program execution exhibits regular behavior until the first gadget is invoked, as the actual ROP attack, which manipulates the program's control flow, starts at that time. To gather HPC data only for the actual ROP process, we implemented a tracer tool that reads configuration data from a file, including the first gadget's address (to set a breakpoint), the path to the vulnerable application and the ROP attack's payload.

The tracer acts like a debugger, injecting a trap instruction via the *ptrace* API, which suspends the target process at the beginning of a ROP chain. In summary, the tracer performs the following steps to exclusively record the ROP behavior: First, it suspends the program and replaces it with a forked child process. Then, it injects the trap (i.e., a synchronous interrupt caused by an exceptional condition, in our case a breakpoint) and runs the vulnerable program until the breakpoint is reached, indicating that the actual ROP execution is about to begin. Thus *perf record* is triggered to record the HPC data for the remainder of the program's execution.

In order to virtualize the HPC readings to each process and thus remove noise from concurrently executing processes we apply the *-p* option to *perf record*. However since only a certain number of HPC events can be recorded at any time, we sample them in smaller batches and train the machine learning algorithms with the combined data in order to select the most relevant HPC event types (features) for ROP detection.

C. Offline learning using HPC data

The offline model learning phase on a high level follows the approach of HadROP [29]. Apart from considering ARM instead of x86, we are also evaluating a number of machine learning techniques beyond a SVM. Training data is gathered by recording the HPC features based on a given set of ROP attacks and benign program executions. Using feature selection techniques we derive the best classification model for online ROP attack detection via the learned classifiers. To determine the models, we use several machine learning (ML) techniques



(a) ROP model

(b) JOP model

Fig. 3: ROP vs JOP

that classify the feature vectors into malicious (ROP attack) and benign. The feature vectors comprise the deltas of the HPC counts per sampling period of a particular program run. The sampling period needs to be fixed before collection of the training data starts and cannot be changed subsequently.

The model generated using the offline training is expected to contain a small subset of the large number of available HPC events by applying feature selection techniques that select the most meaningful HPC events for ROP detection. The problem with feature selection is that the measurements of HPC events during the data collection phase are noisy for several reasons: The complexity of the CPU and non-determinism of the HPC specification renders reproducing HPC events across multiple sub-sampling runs difficult. Moreover, context switching might trigger additional events since HPCs are saved to the process control block.

Despite this problem, the feature selection technique helps determine a small subset of HPC event types supported by Raspberry Pi that best matches the expected results from an in principle large number of events (in our case 51). Note that feature selection is imperative as the selected number of event types must also adhere to the constraints of the target CPU. For instance, on Raspberry Pi, *perf* cannot sample more than 8 event types simultaneously. To resolve this issue we sample using subset batches of the events and combine them (synchronized according to time) for offline learning and feature selection in order to determine the most suitable combination of HPC events.

Model Selection Methods: We train eight ML classifiers by providing two sets of feature vectors, collected from ROP and benign program runs. Moreover, parameters like an error penalty C, which allows more or fewer mis-classifications, are given as input parameters. Choosing the appropriate parameters that result in an optimal model is not trivial. Hence, we select the optimal parameters through dynamic oscillating search [34].

Our model selection approach uses *k-fold* cross-validation to optimize the selection. Cross-validation is a statistical method re-sampling procedure to evaluate and compare machine learning algorithms by splitting data into two segments: one segment for training the machine learning model, and the other segment to validate it. Typically, the training and validation data sets must cross-over in successive rounds such that each data point can be validated against. These evaluation results guide the dynamic oscillating search, which determines the next set of potentially optimal features and parameters. Iterating this process results in optimal parameters, e.g., for SVM in an error penalty and optimal hyper-plane, based on a subset of HPC event types of appropriate size. Considering the bias-variance trade-off in k-fold cross-validation, we choose $k = 10$ as the model selection method as recommended by Kohavi et al. [20].

D. Online Kernel Monitor

Figure 4 presents our online monitoring process using the machine learning model, which consists of a modified *program loader*, a *kernel module* and a *classifier*. The program loader configures the CPU using Linux's *perf* tool to track the set of HPCs that are relevant for the trained model to classify an execution as ROP or benign. Moreover, it notifies the CPU to raise an interrupt every N clock cycles. At each interrupt the kernel module computes the deltas of the HPC count values and feeds those to the classifier, which is implemented in the kernel space to synchronize with the readings of the HPCs during each interrupt. Whenever the classifier determines that a ROP attack behavior occurred, the process can be suspended or other defensive actions, such as notifying security personnel, can be taken. As HPCs are updated in hardware, the performance overhead of online monitoring stems only from handling the interrupt, reading the counters, and evaluating the classifier.

1) Program Loader: We modified the routine that starts a program (program loader). Concretely, we modified the *_libc_start_main* function, which calls the main function of a program, to configure the HPCs selected for classification. In particular, we are adding the *perf record* command, which samples these HPC events, before the call to the main function of the program. The *perf record* command uses the *frequency* N and selected *features* (HPC events) of the optimal classification model as input in addition to the target program to be runtime monitored. To use this modified program loader we leverage the environmental variable *LD_PRELOAD*.

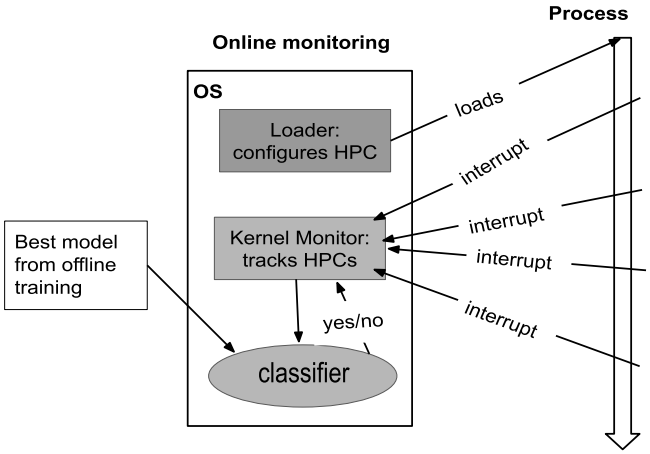


Fig. 4: Our online monitoring approach to detect and prevent ROP attacks

LD_PRELOAD contains one or more paths to shared libraries, or shared objects, that will be loaded before any other shared library including the C runtime library (libc.so). However, in order not to end up calling the modified loader recursively we need to unset *LD_PRELOAD* after configuring the HPCs.

2) Kernel module: The kernel module contains a modification of perf's interrupt handler to recognize the configuration specific to the classifier. Note that interrupts produced by HPCs are recognized as non-maskable interrupts, which must be handled by the kernel and cannot be ignored (masked). The interrupt handler extracts the delta (change of counts) readings of the selected[7] HPC events at each interrupt and passes it as an array to the monitor that contains the classifier, which in turn performs the classification and redacts the execution based on the output of the classifier.

3) The classifier: During offline training, several machine learning techniques are evaluated (see section IV). Given that the SVM provides the best classification model we use that model for online monitoring. LibSVM determined the best HPC events that characterize ROP attacks during offline training. However, we cannot directly use LibSVM for online monitoring since the kernel module, which extracts the HPC readings during the interrupt, is not correctly synchronized with user space, which would call LibSVM's *svm-predict* function. Due to the lacking synchronization most of the HPC readings would be missed due to delays in the user space actions such as extracting the model file and performing the SVM prediction calculations. Hence, we decided to implement the SVM prediction (classifier) directly in kernel space. Yet, there are two problems with implementing the SVM's prediction in the kernel module.

1) Reading the SVM model file in the non-maskable interrupt context
2) Using floating point arithmetic in the kernel module

[7]HPC events selected by feature selection to provide the optimal classification model

The kernel space does not support reading files in the interrupt context. Hence we cannot read the relevant classification inputs directly from the optimal classification model file. Since this data is static, we extract the relevant data from the model file and store it into arrays and/or variables before online monitoring.

Moreover, the kernel module is unable to support floating point arithmetics [3, ch. 5], which prohibits implementing the SVM prediction formula directly. Hence, we opted for solving this problem using *fixed-point arithmetic*[8]. For instance, considering the SVM prediction formula for the RBF kernel given below, we need to use fixed-point arithmetic to represent e, γ and the support vector coefficients (a_i), which usually have floating point values. The bound *nSV* in the formula represents the number of support vectors.

$$y = \sum_{1}^{nSV} a_i * e^{-\gamma * |X - X_i|} + b$$

With both of these issues solved the classifier can now receive HPC readings from the interrupt handler and predict the class of the program execution. Based on the output of the classifier the online monitor can then decide what to do with the running application. If the behavior of a ROP attack is detected the monitor can suspend the process or notify the responsible body (e.g. security personnel) about the issue.

IV. Evaluation

Our evaluation environment consists of Raspberry Pi 4 Model B and Raspberry Pi 3 Model B with kernel version 5.4. The evaluation comprises multiple experiments to obtain the parameters that provide an optimal model of ROP attack classification, to measure the accuracy of ROP attack detection and of the performance overhead of online monitoring.

A. Optimal model selection and accuracy of offline training

To find the optimal model during offline training, we perform many experiments with varying sampling frequencies, several machine learning techniques and their classification parameters on both Raspberry Pi 3 and Pi 4.

*1) **Accuracy evaluation with respect to different frequencies:*** In order to reduce the space for the sampling rate we conducted initial experiments with only one ML technique [29]. Table I shows the classification accuracy of SVM models for different frequencies. The cost (C) and gamma (γ) parameters that provide the best accuracy for their respective frequencies are also provided. The experiments use the crossover value $k = 10$. The result of this evaluation shows that data collected by HPC recording with frequency 4,000 Hz on Raspberry Pi 3 provides the best accuracy, which is 91%, and on Raspberry Pi 4 data collected by recording with frequency 9,000 Hz provides the best, which is 92%, but the recording with 4,000 Hz on Raspberry Pi 4 also provides similar results, i.e., it is 91%. However, this does not mean we always get the same results each time we collect data with these frequencies

[8]https://en.wikipedia.org/wiki/Fixed-point_arithmetic

TABLE I: Accuracy evaluation of ROP attacks using different frequencies on both Raspberry Pi 3 and Pi 4. We also show the optimal C and γ that provide the best accuracy for each frequency.

Frequencies	Pi 3 model B			Pi 4 Model B		
	Cost (C)	**Gamma (γ)**	**Accuracy**	**Cost (C)**	**Gamma (γ)**	**Accuracy**
3000	4	0.000031	0.89	4	0.000031	0.87
4000	64	0.000488	0.91	256	0.000488	0.91
5000	4	0.000122	0.85	256	0.000488	0.89
6000	4	0.000031	0.80	256	0.000488	0.88
7000	16	0.000031	0.82	64	0.000488	0.87
8000	256	0.000031	0.82	256	0.000122	0.84
9000	4	0.000031	0.82	256	0.000031	0.92

TABLE II: Precision, recall and accuracy evaluation of ROP attacks using different machine learning techniques for both Raspberry Pi 3 and Pi 4.

Model Name	Pi 3 model B (4000Hz)			Pi 4 Model B (4000Hz)			Pi 4 Model B (9000Hz)		
	Precision	**Recall**	**Accuracy**	**Precision**	**Recall**	**Accuracy**	**Precision**	**Recall**	**Accuracy**
KNeighbors	0.89	0.87	0.89	0.72	0.81	0.78	0.87	0.95	0.91
AdaBoost	0.85	0.96	0.90	0.80	0.97	0.89	0.84	0.90	0.88
GradientBoosting	0.88	0.96	0.92	0.88	0.97	0.92	0.86	0.93	0.90
DecisionTree	0.82	0.96	0.88	0.85	0.97	0.91	0.87	0.83	0.85
RandomForest	0.83	0.96	0.89	0.85	0.94	0.90	0.88	0.90	0.90
ExtraTrees	0.85	0.96	0.90	0.85	0.97	0.92	0.87	0.96	0.92
LinearDiscriminant	0.70	0.98	0.79	0.72	0.98	0.84	0.85	0.98	0.92
SVM RBF	0.87	0.98	0.91	0.82	0.98	0.90	0.85	0.98	0.92

due to the non-deterministic behavior of HPC readings. The HPC readings vary at each execution for the same application even with the same frequency. Although the frequencies that provide the best model vary, we observe that there is no significant difference between Raspberry Pi 3 and Pi 4 in the offline classification accuracy.

*2) **Accuracy evaluation with respect to different machine learning techniques**:* With the reduced set of sampling rates, we then evaluate our offline training with respect to multiple machine learning techniques, the result of which is provided in Table II. In addition to the accuracy, we also evaluate recall and precision of the models obtained using the frequencies 4,000 Hz on Raspberry Pi 3, and 4,000 Hz and 9,000 Hz on Raspberry Pi 4. Note that *recall* indicates the percentage of feature vectors (FV) correctly classified as ROP attack of all FVs of ROP attacks. Similarly *precision* indicates the percentage of the FVs *correctly* predicted as ROP attack in all FVs classified as ROP. So both the recall and precision provided in Table II are with respect to the ROP attacks. In contrast, accuracy measures the overall classification accuracy for both ROP and benign executions. The training data used in all experiments is also balanced, i.e., we use the same number of ROP and normal feature instances as input to the machine learning techniques.

As we observe from the result, our evaluation indicates that the detection accuracy of the optimal model using an SVM kernel is at least in the top two. For instance, considering the Raspberry Pi 4 ExtraTrees, LinearDiscriminant and SVM RBF kernel provide the best accuracy which is 92%. However if we also take the recall both LinearDiscriminant and SVM RBF kernel have a higher value than ExtraTrees though somehow less in precision. Similarly if we consider Raspberry Pi 3 GradientBoosting provides the best accuracy of 92% whereas SVM RBF kernel provides 91%. However, the SVM RBF kernel provides better recall than GradientBoosting with almost similar precision. From these results on both Raspberry Pi 3 and Pi 4 we conclude that the model obtained using the SVM RBF kernel is optimal for online monitoring. All subsequent experiments are thus based on a SVM with RBF kernel only.

*3) **Accuracy evaluation with respect to ROP and JOP**:* We further evaluated the classification accuracy between ROP and JOP attacks to understand how different attack types affect the behavior of the HPC values. Table III provides the accuracy of both ROP and JOP attacks using different frequencies on both Raspberry Pi 3 and Pi 4. On Raspberry Pi 3 the models from ROP attacks yield a higher accuracy than from JOP attacks for frequencies 3 kHz and 5 kHz. In contrast, considering 8 kHz and 6 kHz the model from JOP attacks has higher accuracy than from ROP attacks. For the other frequencies, the accuracy is similar for both variants. Similarly, on Raspberry Pi 4 the model from ROP attack provides higher accuracy than from JOP attacks for 4 kHz but for 3 kHz and 6 kHz frequencies, the model from JOP provided higher accuracy than the model from ROP attacks. For the other frequencies, there is not much difference. These results indicate that we can not generalize that one attack type is more detectable than the other, i.e., it depends on the frequencies and even may vary when recording it again even with the same frequency.

*4) **HPC events providing the best classification model**:* As we observe from the offline training evaluation, we got a classification accuracy of 92% on Pi 4 and 91% on Pi 3. The selected HPC events that provide these results for the Pi 4 are: *branch-misses*, *branch-load-misses*, *ld_spec*. Similarly, the selected HPC events for Pi 3 are: *cache-misses*, *branch-misses*, *cid_write_retired/*. It is interesting to note that the occurrence

TABLE III: Accuracy evaluation of ROP vs JOP attacks using for difference frequencies on both Pi3 and Pi4.

Frequency	Pi3 model B		Pi4 Model B	
	ROP	**JOP**	**ROP**	**JOP**
3000	0.89	0.85	0.85	0.90
4000	0.86	0.87	0.89	0.82
5000	0.84	0.81	0.85	0.84
6000	0.83	0.87	0.80	0.85
7000	0.81	0.82	0.83	0.82
8000	0.81	0.85	0.82	0.84
9000	0.85	0.85	0.85	0.87

TABLE IV: Performance overhead evaluation of real-world applications on both Pi 3 and Pi 4.

Application	**vulnerability**	**slowdown Pi 3**	**slowdown Pi 4**
Crashmail 1.6		8.7%	4.6%
Pms 0.42		11%	9%
Php 5.3.5	CVE-2011-1938	4%	11%
Netperf 2.6.0		7.6%	9%
Wifirix		5.4%	5%
Dnstracer 1.8.1	CVE-2017-9430	5%	4%
Mcrypt 2.6.8	CVE-2012-4409	3%	2.6%
Nethack 3.4.0	CVE-2012-4409	5.6%	4.3%

of ROP attacks impact the count of these three HPC events but significantly of the others, as our feature selection process could have chosen up to eight HPCs for simultaneous tracking. These HPC events will also be used for online monitoring, i.e., to detect and prevent the ROP attacks.

B. Accuracy of online monitoring

To evaluate the detection and prevention accuracy of our online monitoring, we used three real-world vulnerable applications (php, dnstracer, mcrypt), which were not used for training the model. Since we have benign as well as both ROP and JOP attacks for these applications, we have a total of 9 tests, out of which 7 are consistently detected correctly, i.e., the ROP and JOP attacks in *mcrypt* go undetected more often than not. Note that since HPC values are nondeterministic those detected at one time may not be detected another time and vice-versa. Moreover, since the HPC recording during an execution of a program provides many feature instances there is a high possibility that instances are predicted wrongly during the attack and benign executions. To minimize this, we have to look for the optimal maximum number that instances are predicted 1 (ROP attack) consecutively to determine that a real attack has started. In our case we used 10, i.e, if 10 consecutive instances are predicted as 1 we assume there is an attack and the program execution will be suspended, otherwise we assume that a false prediction of the instances has occurred and consider the execution as benign. Hence the ROP and JOP attacks of mcrypt are being mostly undetected probably since their exploit has small gadget chains relative to the others. In general, we have also tested our online monitoring with small hand-crafted ROP attack examples and the detection accuracy of our online monitoring is around 75%, on average.

C. Performance Overhead of the Online Monitoring

The performance overhead of runtime monitoring is measured by comparing the time of execution for the ROP attacks with and without the usage of the runtime monitor. Table IV shows the performance overhead (slowdown in %) of 8 real-world vulnerable applications on Raspberry Pi 3 and Pi 4. For most of them, the slowdown is higher on Raspberry Pi 3 than in Pi 4 but for Php 5.3.5 and Netperf 2.6.0 we got a higher overhead on Raspberry Pi 4 than on Raspberry Pi 3. However, the execution time of both applications is still smaller in the Raspberry Pi 4 than on Raspberry Pi 3 if we consider it separately even with the application of the online monitor. For instance, the execution time for Php 5.3.5 on Raspberry Pi 3 is 1.5s whereas on Raspberry Pi 4 it is 0.5s with the application of the runtime monitor. In general, the overhead evaluation using these 8 applications shows that our implementation of the online monitoring provides a slowdown in the range of 2.6% –11%. On average the slowdown is 6.3%, on Raspberry Pi 3, and 6.2% on Raspberry Pi 4, indicating that the performance overhead is almost identical on both.

V. Related Work

Recently, several hardware-based ROP defense tools such as HDROP [39], SIGDROP [37], HadROP [29], ROPSentry [9] were proposed for x86. They use heuristics or machine learning models which leverage branch misprediction events that occur at return instructions. HDROP utilizes HPCs such as mispredicted return events to defend against ROP exploits. However, it requires the instrumentation of source code to insert checkpoints and provides substantially high overhead. Later on, SIGDROP was proposed, which has strict policies to leverage HPC to efficiently capture and extract the signatures to detect ROP attacks. However, the policies can be bypassed by a determined adversary, for example, by inserting one redundant call-ret paired gadget without causing any misprediction at the return instruction.

The most closely related solution to our work, HadROP [29] was proposed by Pfaff et al., which uses machine learning techniques to generate a kernel module that detects and prevents ROP attacks at runtime using HPC as input data. However, our solution focuses on the more challenging ARM architecture, which is leading the market and capable of outperforming x86 [14]. We also feed much larger data sets from real-world applications into the offline learning technique than HadROP. Unlike HadROP, we also evaluated our SVM model with respect to seven other machine learning techniques.

Das et al. proposed ROPSentry [9], a defense framework, which can detect ROP by analyzing the ROP exploits and spraying attacks using hardware events and reduces the performance overhead by an adaptive and a return miss-based sampling technique, i.e., fetching HPC values at every return miss. But similar to HadROP, ROPSentry is only available for x86, not for ARM.

More recently Omotosho et. al [26] provided a primary investigation on the Xtensa processor architecture to detect ROP attacks on firmware-only embedded devices using hardware performance counters. However, as a primary work it

does not include online monitoring and the evaluation lacks consideration of real-world vulnerable applications.

ROP attack detection on ARM. As explained earlier, most ROP detection approaches are based on x86. However, there are also initial results on the ARM platform. The work of Huage et. al. [17] is one of the earlier ROP attack detection approaches using dynamic binary instrumentation (DBI), which, however, induces a high performance overhead. To overcome this limitation Lee et. al. [21] proposed a *meta-data* driven approach that uses the ARM CoreSight traces supplemented with offline binary analysis to generate meta-data information missed in the debug traces. Then using the information from the meta-data and the debug traces they apply the shadow call stack (SCS) [27] approach to verify the integrity of the direct and indirect call/jump instructions and detect ROP attacks. However, since this needs high memory/storage overhead they tried to improve it in their later papers [22], [23] by instrumenting the binary in the way CoreSight debugger traces provide full information required for the control flow verification. These papers support ROP attack detection using the SCS [27] approach and JOP attack detection using the Branch Regulation (BR) [19] approach. More recently CFVerifier [24] was proposed to overcome the storage overhead caused by the meta-data in [21] by maintaining table entries only for branch instructions instead of every instruction. Moreover, CFVerifier extends CRAs detection on a multiprocessor system where a number of programs run concurrently across multiple CPUs.

Unfortunately, it has been shown [12] that these detection approaches can be circumvented via advanced attacks such as print-oriented programming [7] attacks, counterfeit object-oriented programming [32], or data-oriented programming [16], which are not directly related to the branch integrity. Besides, most of the approaches use the ARM CoreSight debugger based hardware monitor which could drop traces given a sufficiently high branch rate since the monitor requires more time to process a trace than the rate at which branches occur on the target processor [12]. Another limitation of using debugger traces to detect CRA attacks is that the hardware debugger can be used by an attacker to circumvent the security of the system. If the attacker can access the debug interface, he could use it to tamper with the code and data memory, or even disable the hardware monitor by tampering with the tracing mechanism [12]. Moreover, most of these papers use Branch Regulation (BR) analysis, which provides only partial indirect branch protection since indirect branches to any address within the current function are allowed. This leaves BR somewhat vulnerable to unintended branches since it allows CRAs which do not cross function boundaries. In contrast, the HPC and machine learning-based approach does not use an external hardware debugger and cannot drop traces during the monitoring. Besides, it does not use any meta-data that leads to storage overhead. Moreover, since it is using machine learning it is not specific to a set of given attack types or branch regulations. To the best of our knowledge we are the first to investigate ROP attack detection via HPCs and machine learning on the ARM platform.

VI. Limitations and Future Work

A. Limitations

Even though we have found good detection accuracy in the offline training, the online detection accuracy has somehow decreased, potentially due to the fixed-point arithmetics we resorted to [3], and the lack of preprocessing of the data since we want to detect attacks at real-time. The following limitations of our work could also have affected this.

- **Data Size:** The machine learning classifier was trained based on data collected from 10 exploits from 5 real-world vulnerable applications and this might not be representative of all possible exploits, even though the data count of the HPCs gathered is sufficient to apply machine learning.
- **Data Quality:** As Weaver et al. [38] investigated, the non-determinism and over-counting characteristic of HPC readings for hardware events deviates from the expected result for identical runs.

Moreover, even though the ARM processor architecture supports Cortex-A (for OS-based applications), Cortex-M (for microcontrollers) and Cortex-R (for real-time applications), our implementation targets only the Cortex-A, i.e., our approach is implemented on Raspberry Pi machines, which are OS-based Cortex-A ARM processors. Most Cortex-M and Cortex-R do not provide HPCs yet.

VII. Conclusion

This paper addresses the practicability of detecting and preventing ROP attacks using HPCs and machine learning on the ARM processor, which is getting high attention since it supports low power consumption with considerable performance, an attractive combination for mobile technologies and IoT. First, we crafted several real-life exploits using ROP attacks from selected vulnerable programs, which provided HPC data about the execution behavior of these vulnerable programs. For the ROP attack executions, a small debugger tool called "tracer" was implemented to record only the real ROP attack execution part, i.e., after the first ROP gadget starts execution. The SVM RBF kernel is used for offline training and online monitoring of our ROP attack detection approach, as when the detection accuracy of the offline training is evaluated with respect to 7 additional machine learning techniques, it is consistently in the very top, i.e., it provides 92% detection accuracy and no one provides more than that. The detection accuracy and performance overhead of the online monitoring is also evaluated and it provides around 75% detection accuracy with an average 6.2% slowdown overhead. Last but not least our ROP attack detection and prevention approach using HPC and machine learning techniques demonstrates that the characteristics of the hardware events on ARM processors can be used to investigate whether or not an attack is in progress.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) under research grant number 01IS18065D.

References

[1] W^X ("write xor execute"), https://en.wikipedia.org/wiki/W%5EX

[2] Bhavsar, P., Safro, I., Bouaynaya, N., Polikar, R., Dera, D.: Machine learning in transportation data analytics. In: Data analytics for intelligent transportation systems, pp. 283–307. Elsevier (2017)

[3] Billimoria, K.N.: Linux Kernel Programming: A comprehensive guide to kernel internals, writing kernel modules, and kernel synchronization. Packt Publishing Ltd (2021)

[4] Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. pp. 30–40 (2011)

[5] Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. pp. 30–40 (2011)

[6] Buchanan, E., Roemer, R., Savage, S., Shacham, H.: Return-oriented programming: Exploitation without code injection. Black Hat **8** (2008)

[7] Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R.: {Control-Flow} bending: On the effectiveness of {Control-Flow} integrity. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 161–176 (2015)

[8] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 559–572 (2010)

[9] Das, S., Chen, B., Chandramohan, M., Liu, Y., Zhang, W.: Ropsentry: Runtime defense against rop attacks using hardware performance counters. Computers & Security **73**, 374–388 (2018)

[10] Das, S., Werner, J., Antonakakis, M., Polychronakis, M., Monrose, F.: Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 20–38. IEEE (2019)

[11] Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Return-oriented programming without returns on arm. Tech. rep., Technical Report HGI-TR-2010-002, Ruhr-University Bochum (2010)

[12] De Clercq, R., Verbauwhede, I.: A survey of hardware-based control flow integrity (cfi). arXiv preprint arXiv:1706.07257 (2017)

[13] Elsabagh, M., Barbara, D., Fleck, D., Stavrou, A.: Detecting rop with statistical learning of program characteristics. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. pp. 219–226 (2017)

[14] Gupta, K., Sharma, T.: Changing trends in computer architecture: A comprehensive analysis of arm and x86 processors. International Journal of Scientific Research in Computer Science, Engineering and Information Technology pp. 619–631 (06 2021). https://doi.org/10.32628/CSEIT2173188

[15] Homescu, A., Stewart, M., Larsen, P., Brunthaler, S., Franz, M.: Microgadgets: Size does matter in turing-complete return-oriented programming. WOOT **12**, 64–76 (2012)

[16] Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 969–986. IEEE (2016)

[17] Huang, Z.J., Zheng, T., Liu, J.: A dynamic detective method against rop attack on arm platform. In: 2012 Second International Workshop on Software Engineering for Embedded Systems (SEES). pp. 51–57. IEEE (2012)

[18] Huang, Z.S., Harris, I.G.: Return-oriented vulnerabilities in arm executables. In: 2012 IEEE Conference on Technologies for Homeland Security (HST). pp. 1–6. IEEE (2012)

[19] Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., Ponomarev, D.: Branch regulation: Low-overhead protection from code reuse attacks. ACM SIGARCH Computer Architecture News **40**(3), 94–105 (2012)

[20] Kohavi, R.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: Proceedings of the 14th international joint conference on Artificial intelligence. p. 1137–1143. IJCAI'95, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1995)

[21] Lee, Y., Heo, I., Hwang, D., Kim, K., Paek, Y.: Towards a practical solution to detect code reuse attacks on arm mobile devices. In: Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy. pp. 1–8 (2015)

[22] Lee, Y., Lee, J., Heo, I., Hwang, D., Paek, Y.: Integration of rop/jop monitoring ips in an arm-based soc. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 331–336. IEEE (2016)

[23] Lee, Y., Lee, J., Heo, I., Hwang, D., Paek, Y.: Using coresight ptm to integrate cra monitoring ips in an arm-based soc. ACM Transactions on Design Automation of Electronic Systems (TODAES) **22**(3), 1–25 (2017)

[24] Oh, H., Cho, Y., Paek, Y.: A metadata-driven approach to efficiently detect code-reuse attacks on arm multiprocessors. The Journal of Supercomputing **77**(7), 7287–7314 (2021)

[25] Oh, H., Yang, M., Cho, Y., Paek, Y.: Actimon: Unified jop and rop detection with active function lists on an soc fpga. IEEE Access **7**, 186517–186528 (2019)

[26] Omotosho, A., Welearegai, G.B., Hammer, C.: Detecting return-oriented programming on firmware-only embedded devices using hardware performance counters. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing. pp. 510–519 (2022)

[27] Ozdoganoglu, H., Vijaykumar, T., Brodley, C.E., Kuperman, B.A., Jalote, A.: Smashguard: A hardware solution to prevent security attacks on the function return address. IEEE Transactions on Computers **55**(10), 1271–1285 (2006)

[28] Parikh, V., Mateti, P.: Aslr and rop attack mitigations for arm-based android devices. In: International Symposium on Security in Computing and Communication. pp. 350–363. Springer (2017)

[29] Pfaff, D., Hack, S., Hammer, C.: Learning how to prevent return-oriented programming efficiently. In: International Symposium on Engineering Secure Software and Systems. pp. 68–85. Springer (2015)

[30] Prakash, A., Yin, H.: Defeating rop through denial of stack pivot. In: Proceedings of the 31st Annual Computer Security Applications Conference. pp. 111–120 (2015)

[31] Sadeghi, A.R., Wachsmann, C., Waidner, M.: Security and privacy challenges in industrial internet of things. In: 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2015)

[32] Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In: 2015 IEEE Symposium on Security and Privacy. pp. 745–762. IEEE (2015)

[33] Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 552–561 (2007)

[34] Somol, P., Novovicová, J., Grim, J., Pudil, P.: Dynamic oscillating search algorithm for feature selection. In: 2008 19th International Conference on Pattern Recognition. pp. 1–4. IEEE (2008)

[35] van der Veen, V., Andriesse, D., Stamatogiannakis, M., Chen, X., Bos, H., Giuffrdia, C.: The dynamics of innocent flesh on the bone: Code reuse ten years later. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1675–1689 (2017)

[36] Wah, Y.B., Rahman, H.A.A., He, H., Bulgiba, A.: Handling imbalanced dataset using svm and k-nn approach. AIP Conference Proceedings **1750**(1), 020023 (2016). https://doi.org/10.1063/1.4954536

[37] Wang, X., Backer, J.: Sigdrop: Signature-based rop detection using hardware performance counters. arXiv preprint arXiv:1609.02667 (2016)

[38] Weaver, V.M., Terpstra, D., Moore, S.: Non-determinism and overcount on modern hardware performance counter implementations–extended (03 2021)

[39] Zhou, H., Wu, X., Shi, W., Yuan, J., Liang, B.: Hdrop: Detecting rop attacks using performance monitoring counters. In: International Conference on Information Security Practice and Experience. pp. 172–186. Springer (2014)

Appendix

A. Machine learning types

Some of the machine learning algorithms that are used in our evaluation.

- *Support Vector Machine (SVM):* Constructs a (set of) hyperplane(s) for classification, regression, or other tasks. In handling a binary classification task, a SVM provides

higher accuracy in predictive modeling compared to other techniques such as Discriminant Analysis [36].

- *K-neighbors:* It is an algorithm that stores all available cases and classifies new cases by a majority vote of k-neighbors.
- B*oosting:* a family of machine learning algorithms that convert weak learners to strong ones.
- *Decision trees:* uses a tree-like model of decisions and their possible consequences. Decision trees often perform well on imbalanced data sets because their hierarchical structure allows them to learn signals from both classes.
- N*aïve Bayes:* It is a classification technique based on Bayes theorem with an assumption of independence between predictors, i.e., the presence of a particular feature in a class is unrelated to the presence of any other feature. It is frequently used in imbalanced dataset problems.