Learning how to Prevent Return-Oriented Programming Efficiently

David Pfaff, Sebastian Hack, and Christian Hammer

CISPA, Saarland University, Saarbrücken, Germany
{lastname}@cs.uni-saarland.de

Abstract. Return-oriented programming (ROP) is the most dangerous and most widely used technique to exploit software vulnerabilities. However, the solutions proposed in research often lack *viability* for real-life deployment.

In this paper, we take a novel, statistical approach on detecting ROP programs. Our approach is based on the observation that ROP programs, when executed, produce different micro-architectural events than ordinary programs produced by compilers. Therefore, special registers of modern processors (*hardware performance counters*) that track these events can be leveraged to detect ROP attacks. We use machine learning techniques to generate a model of this different behavior, and develop a kernel module that detects *and prevents* ROP at runtime via the learned model. Our evaluation on real-world programs and attacks shows that the runtime overhead of this technique and the number false positives are very low, while preventing all known types of ROP attacks, including recently developed evasion techniques.

1 Introduction

The discovery of recent zero-day exploits against Microsoft Word [1], Adobe Flash Player [2] and Internet Explorer [5] demonstrate that return-oriented programming (ROP) is the most severe threat to software system security. Also, Microsoft's 2013 Software Vulnerability Exploitation trend report [33] found that 73% of all vulnerabilities are exploited via ROP. The core idea of ROP is to exploit the presence of so-called *gadgets*, small instruction sequences ending in a return instruction. By chaining gadgets together, an attacker is able to build complex exploits. The apparent popularity of ROP is explained by its power to bypass most contemporary exploit mitigation mechanisms, such as data execution prevention (DEP) [42] and address space layout randomization (ASLR) [4]. DEP and similar page-protection schemes prevent the execution of injected binary code, but ROP re-uses code already present in the executable memory segments, eliminating the need to inject code. ASLR randomizes the location of most libraries and executables, however, finding code segments left in a few statically known locations is often enough to leverage a ROP attack [38]. Since the inception of ROP by Shacham [37], research on ROP resembles an arms

This is the author's version of the work. The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-15618-7_6

race: emerging defense techniques are continuously circumvented by increasingly subtle attacks [18,22,28].

In this paper, we take a novel, statistical approach on detecting ROP programs. Modern microprocessors spend most of their circuits on machinery that optimizes the execution of programs generated by compilers from "high-level" languages. Among this machinery are caches, translation look-aside buffers, branch predictors, and so on. To assist programmers in detecting performance problems, a modern CPU can record several hundred different kinds of micro-architectural events that occur during program execution (e.g. mispredicted branches, L1 cache misses, etc.). These events are counted by the CPU in special registers, the so-called *hardware performance counters* (HPCs).

In this paper, we claim *and experimentally verify* that the execution of a ROP program triggers such hardware events in a significantly different way than a conventional program that has been generated by a compiler. Essentially, micro-architectural events are a side channel by which a ROP program becomes distinguishable from a normal program at run time. There are several considerations that support this hypothesis: First, ROP programs use only indirect jumps (returns) to control the program flow. Common processor heuristics to detect the target of the return are useless in a ROP program because they do not follow the call/return policy. Second, ROP gadgets are small and scattered all over the code segment. Thus, there is no spatial locality in the executed code which should be observable in counters relevant to the memory subsystem.

We exploit the deviant micro-architectural behavior of ROP programs by training (using existing ROP exploits and benign programs) a support vector machine (SVM) based on profiles of hardware performance counters. Note, that despite our intuition we did *not* short-list any HPC types for training. We receive a classifier to distinguish ROP from benign programs and use it in a *monitor* kernel module that tracks the evolution of the performance counters and classifies them periodically. If the classifier detects a ROP program, defensive actions, like killing the process, can be taken.

We quantitatively evaluate the performance impact of HadROP on benign program runs using the SPEC2006 benchmark [3]: HadROP incurs a run time overhead of 5% on average and of 8% in the worst case. We also establish the effectiveness and practical applicability of HadROP in several case studies that show that HadROP detects and prevents the execution of:

- a ROP payload of an in-the-wild exploit on Adobe Flash Player [2].
- 25 new ROP payloads generated by the ROP-payload generator Q [36] that exploit manually injected vulnerabilities in GNU coreutils.
- Blind ROP [14] of an nginx web server. Using this recent dangerous technique, even amateurs can perform full-scale *remote-code execution* exploits.
- Multiple recent enhancements [18,22,28] that allow ROP to bypass previous hardware-assisted detection schemes. Our diversified monitoring scheme is *not* affected by those attacks in any practical scenario.

In summary, we make the following contributions:

- We present HadROP, a practical and easily deployable ROP defense mechanism, that does neither require instrumentation nor analysis of the code, be it source or binary.
- HadROP exploits the fact that ROP programs trigger micro-architectural events in modern micro-processors differently than conventional programs. Using state-of-the-art machine learning techniques, we train a classifier to detect these differences, which manifest themselves in the values of the HPC registers. HadROP's kernel-level run time monitor identifies ROP programs by periodically classifying the state of the HPCs.
- In several case studies, we evaluate HadROP on a set of existing and new ROP exploits for real-world applications. HadROP detected and prevented all exploits in our benchmark set without reporting a false positive. In a subsequent test of our kernel module on a production machine we encountered only three false positives in 24 hours. The performance overhead of HadROP is low, with an average of 5% on a set of computation-intensive benchmarks.

2 Return-Oriented Programming

Runtime attacks change the behavior of running processes. Typically, attackers exploit a buffer overflow vulnerability on the stack to overwrite the return address [12]. Pointing this address to any memory segment causes the segment to be interpreted as code, not data. By modifying the address to point towards a memory segment containing previously injected machine code, attackers are able to achieve arbitrary execution. Marking memory segments containing user-supplied data as non-executable prevents such code-injection attacks [42]. Many modern processors implement this protection mechanism in hardware in the form of an non-executable bit (NX-bit), which the OS can set for data memory pages.

ROP was developed as a response to circumvent protection mechanisms based on non-executable data pages [37]. For a ROP attack, the attacker overwrites the stack with a set of addresses pointing to already existing executable code. ROP carefully selects this set of addresses by finding suitable code locations to jump to. These code locations contain a few instructions and end on return, which is called a *gadget*. After executing the gadget, the return instruction will jump to the next address as specified on the stack. The addresses on the stack thereby effectively determines the program flow.

Figure 1 illustrates the mechanism of ROP. The payload on the stack (depicted on the the left) contains the addresses of multiple ROP gadgets. In addition to addresses, the payload may also contain parameters and other data that can be copied into registers. Upon execution of the `ret` instruction, the top-most address is interpreted as the return location. In this case, each return will jump towards the next gadget, thereby chaining the gadgets together. By combining appropriate gadgets, it is possible to incrementally build complex functions.

Initial ROP defense attempts primarily tried to prevent execution of malicious code, e.g. by monitoring programs at instruction level while checking for an unusually high frequency of return-instructions [23]. However, often even

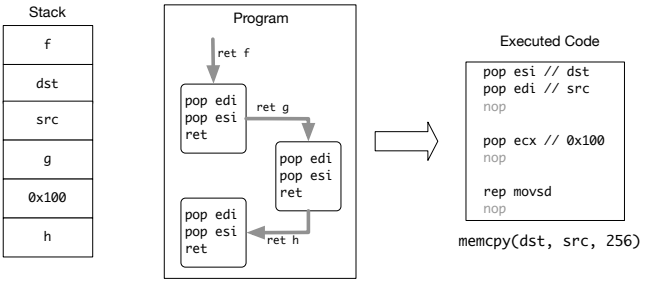

Fig. 1: Cross-section of a ROP exploit.

small adaptations to the ROP paradigm, such as *jump-oriented programming* (JOP) [15], undermine the respective defense technique's core assumption. Therefore, other research focuses on hardening the program itself, e.g., by enforcing Control-Flow Integrity (CFI) [11] at runtime, which has been shown to be theoretically effective. In practice, the inclusion of code fragments for CFI severely degrades *performance* of the target program. Further, the required changes to the program either depend on access to *source code* (which is hard for proprietary software), or deteriorate *compatibility* both with other tools and runtime optimizations. For instance, just-in-time compilation (JIT) becomes impossible, because the generated JIT code does not include the required instrumentation, and thus violates the CFI property. As a result, the wide-spread deployment of those techniques becomes infeasible in practice [40].

3 Our Approach

Our approach builds on the fact that ROP programs differ significantly from conventional programs in terms of micro-architectural events in the CPU. Micro-architectural events are non-functional effects of the program execution on a modern CPU such as cache misses or mis-speculated branches. These events are essentially side channels that can be observed by the HPCs present in every modern CPU (cf. Section 3.1). However, it is hard to identify one particular kind of micro-architectural event that reliably indicates the execution of a ROP program in all its variants [18,22,28]. Therefore, in this work we determine a combination of different events that is characteristic for ROP execution. However, we do not start with a predefined conception of the kinds of events but assume that the exact combination of events *is dependent* on the CPU the program executes on, which is confirmed by our evaluation.

To this end, HadROP uses a statistical approach (cf. Fig. 2), that consists of an offline learning and an online monitoring component. The learning component records HPC profiles for a given set of ROP and conventional programs. Learning on the system the monitor is to be deployed on is imperative as other types of CPUs can lead to different HPC profiles. Then we leverage a SVM to obtain a

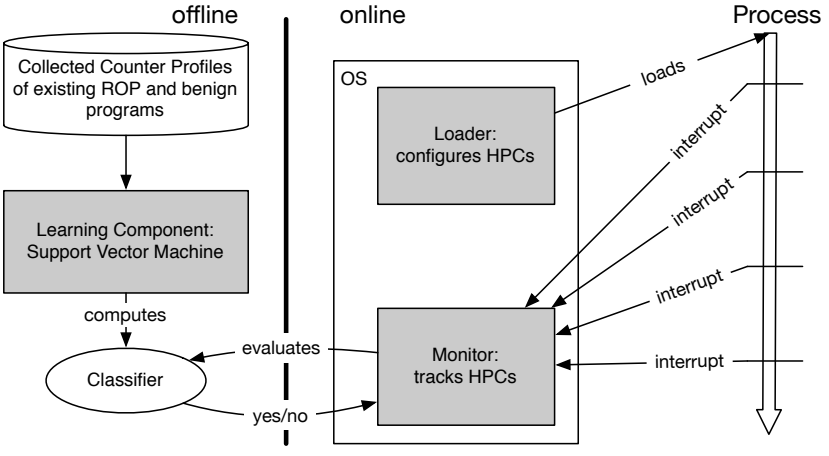

Fig. 2: Components of HadROP

classifier that discriminates ROP programs from benign programs. Once learned, this classifier remains constant on that particular CPU..

The monitoring component consists of a modified program loader and a kernel module. Whenever a new process is created, the modified program loader configures the CPU to track the set of HPCs the classifier needs to compute its result. Furthermore, it tells the CPU to raise an interrupt every N clock cycles. Upon every interrupt, the kernel module computes the difference of the current values of the HPCs to the values recorded at the previous interrupt and feeds those values to the classifier. If the classifier determines that an intrusion is about to happen, the process can be killed, or—depending on the application scenario—another defensive action can be taken such as notifying the user, security personnel, or a security information and event manager, potentially with additional information like a memory dump. Note that the HPCs are updated by the CPU itself, i.e., there is no performance overhead incurred by counting. The only overhead comes from handling the interrupt, reading the counters, and evaluating the classifier.

3.1 Hardware Performance Counters

The original purpose of HPCs is to non-intrusively profile a program by sampling the counters periodically. Usually, a CPU offers a small number of registers to count all sorts of micro-architectural events. On current CPUs 6–8 registers are freely configurable. To find a performance problem in the program, the programmer configures the HPCs to a certain set of events she deems relevant. Furthermore she sets a threshold per counter. If a counter reaches its threshold, the CPU resets the counter and raises an interrupt that is handled by the OS. Note that access to the HPCs is restricted to the operating system. A user-space program can never configure, read, or tamper with the HPCs. Therefore,

in the case of profiling, the profiler needs to be a privileged application that communicates with the kernel to access the HPCs.

In general a CPU counts all configured events in the entire system because a CPU executes obliviously to the concept of a process. To break the counters down to individual processes they have to virtualized by the operating system: Each time the OS is entered the values of the HPCs are retrieved and stored in the process control block of the process that entered the OS. When the OS yields control to another process, it restores the counter values of this process from the respective process control block.

3.2 ROP Classification Using HPC Data

The purpose of the offline learning part as illustrated in Fig. 2 is to determine an unbiased model of which HPCs best indicate ROP attacks. Unbiased means that we make no pre-selection of a particular subset of HPC event types but treat them all equally. To determine this model, we use an SVM, a well-known machine learning technique. SVMs are beneficial here, because the resulting classifier is fast to evaluate, which is integral to our implementation. An SVM computes a hyperplane that separates a set of feature vectors into two clusters. In our setting, each feature vector captures the HPC profile of a particular program run. A feature vector consists of—for each event type—the minimum and maximum number of events per sampling period. The sampling period has to be fixed before data collection starts and cannot be changed afterwards. Remember from the last section that in practice this model can only refer to a small subset of the more than 200 event types available on a modern CPU. So in order to receive an unbiased model of the best combination, we sample event types in smaller batches and align the results after the fact. However, every measurement is noisy for various reasons: the complexity of the CPU makes it hard to reproduce the exact sequence of mirco-architectural events on every run. Furthermore, before the HPCs are saved to the process control block, the context switch might trigger additional events. Because every batch is collected individually, the feature vector of every batch is perturbed in a different way. Due to the large number of different HPC event types, an exhaustive process that trains and compares classifiers for each possible HPC combination is practically infeasible. As our experimental evaluation shows, this effect does not impede the ability of HadROP to successfully detect ROP programs.

Even so, our final classifier must adhere to the constraints of the target CPU, which means that we need to restrict the number of HPCs in our model to a small number that depends on the type of processor. This problem is well-known in machine learning as the *feature selection problem* [27]. Therefore, instead of learning a model based on all possible HPC event types, we leverage feature selection techniques to determine the, e.g., 16-event classifier that best matches the theoretical 200-event one in terms of predictive power.

After combining the batches into a feature vector for the full set of HPC event types, we have two kinds of feature vectors, those from ROP and from benign runs. To train a SVM classifier we feed the two sets of feature vectors into the

learning algorithm, together with parameters like an error penalty C that allows more or less misclassification. Choosing the right parameters is not trivial, so we decided to learn these as well as part of our global optimization problem: *Which is the optimal model and subset of features that predicts ROP attacks best?* We solve this optimization problem using a search algorithm called *dynamic oscillating search* [39] to guide our feature selection algorithm. As the metric for this optimization problem we choose 10-fold cross validation, a standard model validation technique. In cross validation, the data set is partitioned into multiple subsets, of which a single subset is retained for testing and the remainder is used for training. This process is repeated until each set was retained at least once for testing. These evaluation results guide the dynamic oscillating search, which determines the next set of potentially optimal features and parameters. Iterating this process results in an error penalty and optimal[1] hyperplane based on a subset of HPC event types of appropriate size.

In practice, we use libSVM [19] to train the classifier, and the Feature Selection Toolbox [7] to provide the framework and wrapper for the dynamic oscillating search. Recall that we need to collect the HPC profiles of these program runs without recompiling or instrumenting these programs. To that end we replace the monitor in the kernel module presented in the next section with a module that writes the feature vectors to disk for offline classifier computation.

3.3 Data Collection

Producing a stable classifier requires a large set of benign and malicious (ROP-affected) behavior. To obtain a sufficiently large training set, we select several frequent targets of exploits in-the-wild *and* for which a large set of benign inputs are available. This gives us a good chance to collect reproducible exploits along with a number of samples of regular usage, as programs that are frequently exploited tend to be popular programs in general as well. We select Adobe Reader, mcrypt, PHP and the libtiff library to obtain a broad range of program usage scenarios. The programs and their associated vulnerabilities are outlined in Table **??**. Exploits in-the-wild almost always focus on ROP and rarely use exotic variants such as jump-oriented programming or similar techniques. To increase the diversity of our training set we add own exploits based on these paradigms by adapting the in-the-wild exploits. On average, these payloads take 2390 instructions to execute the complete ROP shellcode. The samples range from 200 (a libtiff exploit) at the lowest to over 6000 (an Adobe Reader exploit) at the highest. In addition to malicious input, we also collect benign input from a number of sources: We collect benign PDF files from the arXiv archive [6] and the Intel Software Developer Manuals [9], benign PHP source files from GitHub trending PHP repositories [8], benign TIFF images from the sample sources of the libtiff library [10] and benign usage samples of mcrypt by selectively encrypting the other samples.

[1] Due to the nature of such search algorithms one might potentially only determine a local optimum.

3.4 Kernel Monitor

The active, online component of HadROP is implemented as a patch to the Linux kernel. It consists of three main components: a modified program loader, an extended interrupt handler, and changes to the HPC API.

The program loader is modified in the routines that start a program. It configures the HPCs needed for the classifier in the monitor. Currently, the configuration is the same for all processes, but virtualized for every thread. This means that there is no interaction between the program loader and the monitor beyond the initial configuration of the HPCs. After initializing the HPCs, the program loader proceeds as usual. Note that it is possible to disable this step by preloading a custom program loader routine using `LD_PRELOAD`. This is not a security issue, but a feature of our approach. Since preloading is disabled for security-critical programs (such as setuid applications), we can give unprivileged applications the ability to disable this protection mechanism. As a result, HPCs can be configured by programs and their original purpose, performance evaluations during program development, is preserved.

The interrupt handler in `perf_events` is patched to recognize the configuration specific to the classifier. Configurations not specific to HadROP are handled as usual to preserve the original HPC functionality. Interrupts produced by HPCs are recognized as non-maskable interrupts, which must be handled by the kernel and cannot be ignored (masked). The monitor containing the classifier is called from within the interrupt handler. It reads the contents of all relevant HPCs and applies the SVM classifier on the values. Depending on the outcome of the classifier, it either terminates the interrupt handler to pass control back to user mode, or executes a defensive action, e.g. kills the process.

Finally, we change the API that exposes the HPC to user mode. In particular, we need to restrict processes from modifying their own HPC configuration during observation of HadROP. This restriction is necessary, since otherwise, ROP-affected processes could simply mask their execution by changing the HPC configuration.

4 Evaluation

Our main evaluation environment consists of an Intel Core i7-4800MQ 3.7GHz system running Linux Mint 15 OS with Linux kernel version 3.11 The evaluation is partitioned into multiple experiments that validate different aspects of HadROP.

4.1 Learning Environment and Classifier

The measurement period has to be determined experimentally *before* training the classifier: since the period affects the magnitude of the measurements, we cannot treat it as another parameter to be optimized during the feature selection process. Therefore we repeat the data-collection and the optimization process multiple times for a selection of different values. We compare the accuracy and slowdown of those experimental runs in Figure 3.

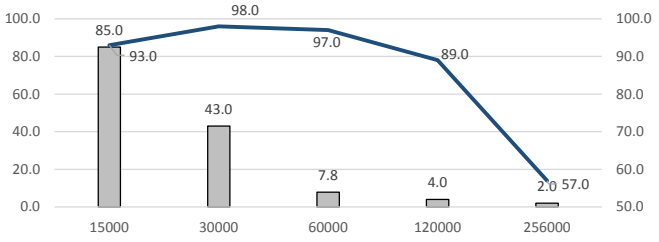

Fig. 3: Trade-off between slowdown (bars) and overall detection accuracy (line).

4.2 Evaluation of HPCs Chosen by the SVM

An evaluation of the classifier entails a thorough analysis of the HPCs used within the classifier. In particular, we want to answer the following questions: Do multiple HPCs event type improve ROP detection? Is the learned information generic or specific to each processor?

To answer the first questions we analyze the distributions of individual HPC events for benign and malicious executions. Figure 4 shows *box-and-whisker* plots of both (normalized) distributions. We observe that the ROP-affected runs display little variance. In contrast, benign executions cover a larger portion of the space. Most importantly, the outliers of benign execution stretch across the region occupied by malicious execution. Hence we cannot reasonably expect to infer ROP-execution from only a single measurements without an unreasonably high false-positive rate. Note that benign execution only shows these outliers on a small subset of HPC events at the same time. Finding an appropriate subset that minimizes the correlation of benign outliers maximizes the indicativeness for ROP-affected behavior. Our automated approach of fine-tuning and optimizing an SVM classifier using feature-selection shows much better performance than any singular measurement.

To answer the second question we take an experimental approach and compare the classifiers of two different processors, CPU1 (Core i7-4800MQ 3.7GHz) and CPU2 (Core i7 860 3.4GHz). Table 1a shows that using the classifier trained on CPU1 for runs on CPU2 decreases the detection rate significantly. Vice versa, the converse yields similar results. Even training the classifier on both CPU1 and CPU2 using a combined data set does not improve the detection rate for either CPU.

4.3 Performance Impact

To assess the overall performance impact of HadROP's monitor to compute-intensive applications, we ran the integer benchmarks from the standard performance benchmark suite SPEC2006 [3]. For each benchmark, we measured the wall-clock-time and compared it to runs during which the monitor was not active. The results shown in Table 1b illustrate that we have an average overhead incurred by HadROP of about 5% and a maximum of 8%. This overhead

Table 1: Evaluation of our trained classifier.

(a) Accuracy of classifiers trained on CPU X for detection on CPU Y.

trained \ used	CPU 1	CPU 2
CPU 1	97.3	83.9
CPU 2	79.5	98.1
CPU 1&2	78.1	79.6

(b) SPEC2006 Integer Benchmark slowdown induced by our kernel patch.

Benchmark	Slowdown	Benchmark	Slowdown
perlbench	4.98%	sjeng	6.60%
bzip	4.60%	libquantum	7.67%
gcc	7.32%	h264ref	7.32%
mcf	4.95%	omnetpp	4.84%
gobmk	8.06%	astar	4.62%
hmmer	4.88%	xalancbmk	4.53%

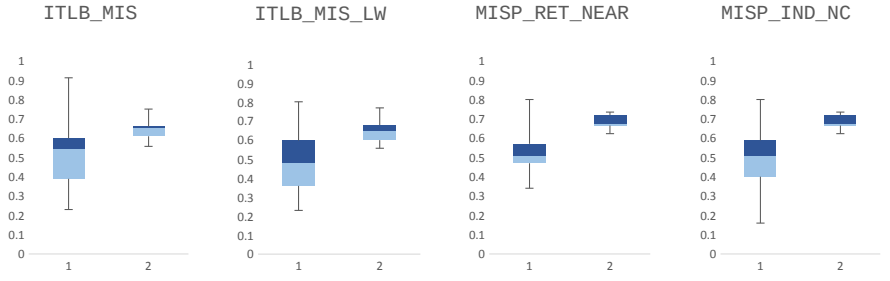

Fig. 4: Distribution of chosen HPCs for benign (1) and ROP-affected (2) runs.

consists of the context-switch costs, the handling of the interrupt, and the evaluation of the classifier. The largest share of this overhead can be attributed to the context-switch costs. None of these (non-trivial) benchmarks raises a false positive.

4.4 Obtrusiveness

An important part in the viability of monitoring schemes is how much they negatively impact everyday usage scenarios. While false positive rates of individual classifiers are useful to give a rough idea on the relative performance, they can and should not be treated as absolutes. Hence we conducted an experiment, wherein patched systems are challenged with typical usage scenarios in a time frame of 24 hours. Typical usage includes browsing the internet on different browsers, coding and debugging and watching videos and listening to audio files. In the course of 24 hours we were notified of only three false positives. In all cases, these false positives happened close to system (re-) starts: twice within a debugger and once during an automated software update. Dumps of the HPC data showed that during start-up periods more misses and mispredictions were recorded, although the impact is usually not severe enough to be reported as a false positive.

4.5 Case Study: Adobe Flash Player

The purpose of this case study is to evaluate the effectiveness of HadROP on attacks that were not part of the training dataset. We ran (an adaption of) a recently discovered Adobe Flash Player exploit [2]. In the wild, this attack was targeting Windows and OS X systems, but the vulnerability and general procedure is applicable to Linux as well. The attack includes several advanced ROP techniques, such as *stack pivoting*, which compiles a ROP-stack in an attacker-controlled buffer on the heap. We replaced the final code as to open a root shell and verified that the attacks succeeds on an unpatched system.

Afterwards, we patched the system with our kernel module. Note that the classifier had not been trained on Adobe-Flash specific HPC data (cf. Section 3.3). As expected, playing benign swf files was *not* classified as a ROP attack. However, opening the malicious swf file, the player was killed by our kernel module before a root shell could be opened. We conclude that our approach is therefore effective even for programs whose behavior was not analyzed during the offline training stage.

4.6 Case Study: Nginx and Blind ROP

Our final case study is concerned with one of the most dangerous scenarios, in which a remote service can be exploited without knowledge of the binary. While automated ROP payload derivation tools such as Q already make it easy for a semi-experienced hacker to build a functional shellcode, Blind ROP (BROP) [14] only requires minimal knowledge and can easily be distributed to amateurs.

A typical BROP attack proceeds in multiple stages. First it searches for the base address of the executable. During all guessing stages, the attacker monitors whether the exploited application crashed due to an invalid memory address. This process will eventually lead to valid addresses. In the following stages, the exploit searches for specific types of gadgets as ROP building blocks. Afterwards, it identifies two useful functions that dump the entire binary in order to derive the final ROP chains for the binary on-the-fly. Note that this exploit consists of two separate ROP chains: one to search for gadgets and to dump the binary and one to execute the real exploit built from the dumped binary.

For our evaluation, we apply this attack to the nginx web server vulnerability CVE-2013-2028. We host the appropriate nginx web service on our machine and configure it to constantly restart after a crash, to fulfill BROPs requirements. Afterwards we start the attack remotely. In our experiments, HadROP detected all malicious queries and the resulting ROP-execution reliably and terminated the web server before the final ROP chain could be fully executed. Note that we performed further case studies like the attacks proposed in [18,22,28] (see discussion in Section 6), for which we do not have space to discuss them in detail here.

5 Related Work

Runtime Monitoring Runtime solutions usually monitor execution at the instruction level. A sophisticated method to achieve transparent monitoring is the use of dynamic binary instrumentation frameworks such as Pin [31]. DROP [20] and DynIMA [23] are exemplary of early runtime monitor attempts to detect the execution of ROP exploits by checking for an abnormally high frequency of return instructions. ROPDefender [24] instruments call and ret instructions to update a separately maintained shadow stack; ROP is then detected by comparing the shadow stack to the actual system stack on every return. All those early tools do no detect jump-oriented programming and similar ROP variants that do not use `ret` gadgets. ROPStop [30] is a sophisticated instrumentation based on a similar observation to ours: regular, compiler-produced programs adhere to a set of normal execution behavior, while ROP executes chains of short gadgets without regarding their location. However, they differ from us in monitoring these properties by analyzing the program binary, including the extraction of a control flow graph, at system-call entries. Like similar methods, this approach is dependent on accurate and fast disassembly. ROPGuard [26] provides a set of system-call wrappers that verify a set of properties on `call`, such as the instruction preceding the return address being a call instruction. This indicates that the return is targeting a genuine return address.

Hardware-assisted Program Monitoring Major drawbacks of most runtime monitoring solutions are their dependency on program instrumentation and high runtime overhead. Recently, multiple solutions surfaced which leverage existing hardware components to enforce security properties. kBouncer [34] utilizes the Last Branch Record (LBR) facilities of modern processors, which record the branch targets of the last 4-16 branch instructions. They verify the validity of program execution by running heuristics on LBR data upon system API entry. ROPecker [21] improves on the idea of kBouncer by not only inspecting the previous indirect branch targets, but also simulating future ones using elaborate prediction mechanisms based on LBR data. Yuan et al. [44] pre-analyze the binary and store instructions preceding return instructions. During runtime, they utilize Branch Trace Store (BTS), which stores branch instruction targets in RAM to compare the previous branch locations to the stored instructions. It was one of the first works to identify HPC events as indicators for code injections attacks. However, their analysis of HPC events does not include ROP and is concerned only with a small subset of potentially useful events. Wicherski [43] successfully links a single branch prediction event to ROP execution. In contrast to HadROP, all of these systems have been shown to be susceptible to attacks [18,22,28]. We consider their focus on a single type of feature as the root cause to their vulnerabilities.

Another security application of HPC was demonstrated by Malone et al. [32], who pre-record HPC data profiles for individual binaries that can be compared to the execution traces at runtime. This approach successfully finds minor deviations in runtime behavior, indicating an integrity breach. Demme et al. [25] use

the same technique to extract malware signatures and contrast them to typical program behavior. Both techniques are focused on intrusion detection instead of attack prevention and therefore orthogonal to our approach. Furthermore, we do restrict ourselves to the detection of ROP, but do not need to analyze the binary. As a result, we manage to capture an intrinsic property of ROP in its effects on HPCs across different programs.

Concurrent work [41] presents a system to detect drive-by attacks in Internet Explorer (and some plugins). While it is close to our research in terms of technology (leveraging machine learning and HPCs) they concentrate their detection on "phase1" after a ROP attack has already disabled DEP. This reflects in their results: They can detect malware in "phase1" with very high success rate, but in the ROP phase their detection rate is low. In contrast, our work targets ROP purely, in all of its variants (even if they do not disable DEP) and for all programs on a system, which led to the development of a kernel module.

6 Discussion

As noted in Section 3.1, modern CPUs only make a limited amount of HPCs available, which are further restricted by non-documented conflicts at measuring related HPC events. Given that previous generations of processors provided more registers for concurrent use, this is a negative development. Furthermore, the accuracy of measurements on the HPCs are subject to noise, i.e., the counters are not required to perfectly correspond to the executed program. Improvements in these areas might improve detection accuracy due to measuring less noisy and more diverse event data, which reduces the number of false positives.

The integration of machine learning techniques also incurs machine-learning specific limitations. Our classifier was trained on a set of data, which might not be representative of all possible exploits. In the process of the classifier derivation, we analyzed roughly 40 exploits based on ROP and its variants. All exploits were based on real-world exploits and adapted to a Linux operating system and variants of ROP exploits that are not encountered in-the-wild (JOP). This learned classifier thus does not fully guarantee that we will immediately detect new, as of yet unseen, ROP variants even though we have successfully prevented a large set of untrained attacks. In the event that a ROP variant is not detected, however, we can include it in our analysis to produce a classifier capable of detecting that class of exploits in the future.

The behavior-based detection of HadROP supports an easy adaption to all currently employed ROP variants, yet it potentially makes it susceptible to attacks that try emulate benign behavior—so-called *mimicry attacks*. We evaluate the feasibility of two possible variants of such attacks in the following. *Call-preceded gadgets* [18] attack the assumption that gadgets do not form call-return pairs. As ROPecker and kBouncer do not determine whether the return-location corresponds to the correct call, their defense mechanisms only limits the number of useful gadgets. In contrast, TOOL is not affected by call-preceded gadgets, as the CPUs return-prediction requires actual call-return pairs. *History flushing*

targets defense mechanisms that only maintain a limited history, such as storing the last 20 branch targets. *Evasion attacks* operate similarly, but instead they evade detection immediately by only using gadgets that directly contradict the defense's core assumptions. In our context, an attacker attempting to evade detection must actively manipulate *all* relevant HPC values for the execution to seem benign. This process is far from trivial for an LBR record, for multiple HPCs this becomes onerous. For the target binaries for which history-flushing and evasion attacks could be constructed, our tests show that these attacks eventually become powerful enough to evade detection, but this comes at a price. Using meaningless but inconspicuous computations to cloak the ROP exploit leaves an attack with less than 10% effective computation progress as noted in [18]. When testing this attack pattern to enhance the original flash exploit from subsection 4.5 we even observed a 20-times slowdown. Thus, the ROP attack does not finish within multiple minutes, during which the process will be unresponsive. Hence we believe that such attacks cannot be feasibly used in practice to attack TOOL. Note that all of these studies were performed on a system with ASLR disabled. Other recent orthogonal techniques (e.g. [13]) render the construction of these kinds of attacks even more infeasible.

7 Conclusion

ROP exploits are the most severe threat to software security. Traditional techniques that have shown to be effective at preventing ROP require the analysis or instrumentation of source or binary code, which severely hampers their deployment. Our approach is novel in using a statistical approach that observes the micro-architectural events of a CPU, which are abnormal for ROP programs.

We detect ROP programs with very high confidence: In our experiments we detected and prevented *all* of our real-world benchmark ROP attacks, be it known exploits or newly-crafted attacks. Our technique produced *no* false positives on all benchmark programs and only three false positives in a 24 hour usage test but—more importantly—*no false negatives*. The run time monitor incurs 5% slow down on average and 8% at maximum. In contrast to related techniques, we neither require analysis nor modification of the application source or binary code. Our implementation as a kernel patch makes our system easy to deploy to production systems.

Acknowledgments This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA).

References

1. Emerging threat: Microsoft word zero day (cve-2014-1761) remote code execution vulnerability. http://www.symantec.com/connect/blogs/emerging-threat-microsoft-word-zero-day-cve-2014-1761-remote-code-execution-vulnerability.

2. Security updates for adobe flash player. http://helpx.adobe.com/security/products/flash-player/apsb14-07.html.
3. Spec standard performance evaluation corporation. http://www.spec.org.
4. The PaX Team. https://pax.grsecurity.net/.
5. Vrt: Anatomy of an exploit: Cve 2014-1776. http://vrt-blog.snort.org/2014/05/anatomy-of-exploit-cve-2014-1776.html.
6. arxiv.org e-print archieve. http://arxiv.org/, 2014.
7. Feature selection toolbox. http://fst.utia.cz/, 2014.
8. Github. http://github.com/, 2014.
9. Intel 64 and ia-32 architectures software developer manuals. http://www.intel.com/content/www/us/en/processors/ architectures-software-developer-manuals.html, 2014.
10. Libtiff. http://www.remotesensing.org/libtiff/, 2014.
11. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.
12. Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), 1996.
13. M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 433–447, San Diego, CA, Aug. 2014. USENIX Association.
14. A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy, S&P*, volume 14, 2014.
15. T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.
16. E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.
17. B. Buck and J. K. Hollingsworth. An api for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.
18. N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, Aug. 2014. USENIX Association.
19. C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.
20. P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Information Systems Security*, pages 163–177. Springer, 2009.
21. Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *The 21th Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.
22. L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, Aug. 2014. USENIX Association.

23. L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 49–54. ACM, 2009.
24. L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.
25. J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 559–570. ACM, 2013.
26. I. Fratric. Runtime prevention of return-oriented programming attacks. http://ropguard.googlecode.com/, 2012.
27. I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.
28. E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, San Diego, CA, Aug. 2014. USENIX Association.
29. A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. In *WOOT*, pages 64–76, 2012.
30. E. R. Jacobson, A. R. Bernat, W. R. Williams, and B. P. Miller. Detecting code reuse attacks with a model of conformant program execution. In J. Jürjens, F. Piessens, and N. Bielova, editors, *Engineering Secure Software and Systems*, volume 8364 of *Lecture Notes in Computer Science*, pages 1–18. Springer International Publishing, 2014.
31. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.
32. C. Malone, M. Zahran, and R. Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the sixth ACM workshop on Scalable trusted computing*, pages 71–76. ACM, 2011.
33. Microsoft. Software vulnerability exploitation trends, 2013.
34. V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Conference on Security*, 2013.
35. D. Rosenberg. Breaking libtiff. http://vulnfactory.org/blog/2010/06/29/breaking-libtiff/.
36. E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium*, 2011.
37. H. Shacham. 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*, pages 552–561. ACM, 2007.
38. K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

39. P. Somol, J. Novovicova, J. Grim, and P. Pudil. Dynamic oscillating search algorithm for feature selection. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4, Dec 2008.
40. L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.
41. A. Tang, S. Sethumadhavan, and S. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In *17th International Symposium on Research in Attacks, Intrusions and Defenses*. Springer, 2014.
42. A. van de Ven. New security enhancements in red hat enterprise linux v.3, update 3. Technical report, Red Hat, Raleigh, North Carolina, USA, 2004.
43. G. Wicherski. Taming rop on sandy bridge. syscan (2013).
44. L. Yuan, W. Xing, H. Chen, and B. Zang. Security breaches as pmu deviation: detecting and identifying security attacks using performance counters. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, page 6. ACM, 2011.