Hardware-Assisted Application Misbehavior Detection

Blinded Submission.

[1]Blinded Author names.

Abstract. *Programming is an error-prone task, thus resulting in application misbehavior. From the safety point of view, crashes are undesirable as they affect user experience. From the security one, vulnerability exploitation can lead to security violations. Although fuzzing and other techniques can help to minimize such occurrences, these still are seem in practice. As an additional layer, real-time monitoring can help to catch such cases. However, approaches like Control Flow Integrity (CFI) are too specific to be extended to the general case. To overcome this challenge, we propose a hardware-assisted flow learning technique, able to profile and detect deviations from the standard behavior, thus ensuring proper application execution.*

1. Introduction

Humans are susceptible to fail, so our developed programs. Recent study [DHS 2013] shows 90% of security incidents result from software defects. Moreover, the OWASP project classifies buggy implementation as the root cause of most web attacks [Morana 2010]. The impact of such implementation flaws goes from operational issues [Guardian 2015] to financial [Register 2011] and privacy [Forbes 2017] losses– when system are exploited.

To reduce the number of application bugs, good software engineering is essential [DoD 2005]. In this field, fuzzy testing is noticeable for its ability to test program paths [Li et al. 2017].

However, despite the taken efforts, application misbehavior still appears in practice, as can be seem in the number of exploited applications [Kaspersky 2017]. To handle such cases in actual scenarios, runtime monitoring approaches have been proposed. As an example, CFI policies have been applied to counter ROP attacks [Pappas et al. 2013]. However, such policies present many drawbacks, such as requiring recompilation [Tice et al. 2014] and granularity issues [Göktaş et al. 2014]. In addition, such kind of policy is attack-specific, thus not handling other kind of flow changes. As to adopt specific policies for each attack class is impractical, we need to develop more general solutions.

As an alternative approach, we propose a learning solution which profiles an ordinary application execution to build a baseline and compares it to other executions to identify misbehavior cases. Such learning characteristic allows us to detect attacks without writing specific monitoring rules. In addition, our solution is hardware-based, thus not requiring code-recompilation nor imposing high overheads.

The remaining of this work is organized as follows: In Section 2, we present related work and how the one here presented differs from these; in Section 3, we introduce the key concepts of our solution proposal; in Section 4, we discuss design and implementation of our solution; in Section 5, we evaluate our solution with synthetic and real applications, showing its effectiveness; in Section 6, we discuss the advances and limitations presented by our proposed concept; finally, we draw our conclusions in the section 7.

2. Related Work

Our solution is related to multiple research topics, each one presented below as a way of positioning our solution among them.

2.1. Fuzzing

Fuzzing solutions try to cover most execution paths of a given binary code [Li et al. 2017, Pham et al. 2016, Böhme et al. 2016]. Our solution is similar to fuzzy ones in the sense our learning phase also tries to cover multiple paths. However, we do not try to generate inputs to reach each code branches, but we rely on user interactions to do so. By runtime monitoring application execution, our solution is able to learn the most frequently taken branches, thus identifying when an abnormal one is taken, which might indicate an exploitation path.

2.2. Control Flow Integrity (CFI)

CFI policies are popular solutions to mitigate ROP attacks. They ensure, for instance, `RET` instructions must be preceded by `CALL` ones, thus mitigating the effects of buffer overflows. Solutions such as Kbouncer [Pappas et al. 2013] and ROPecker [Cheng et al. 2014] are able to runtime monitor code execution and detect flow integrity policy violations. Our solution is related to CFI policies in the sense we try to detect abnormal execution paths. However, we do not rely on a specific rule, such as the `CALL-RET`, but we infer such implicit rules from a learning-based technique.

2.3. Branch Monitoring

Branch monitors are hardware features able to provide information about branch executed instructions with low overhead. Their usage goes from application profiling [Akiyama and Hirofuchi 2017], coverage testing [Shye et al. 2005], to security (**Blinded**).

In this work, we rely on branch monitors to track application execution with low performance penalty. Branch information is used as input to our learning algorithm to decide whether taking a given path is allowed or not.

2.4. Hardware-Assisted approaches

The development of hardware extensions to handle security problems is not a new field. In the literature, we can find a wide range of proposals, from real-time Control Flow Graph (CFG) checking [Arora et al. 2005] to syscall clustering [Das et al. 2016]. The closest related work to ours is presented in [Zhang et al. 2004], on which authors modify a processor to detect flow transitions which violate given policies. In this work, we propose to implement a similar concept but using existing hardware instead.

The rationale behind hardware-assisted approaches is to avoid performance degradation. Therefore, most proposals opt to move all processing components to hardware. In our understanding, the major drawback regarding software solutions is data capture, not threat intelligence, thus we propose moving only this first step to the hardware. In this sense, our work is related to [Ozsoy et al. 2015], which proposes a two level monitor, notifying other components–such as an AV–when a violation is detected.

3. Concepts and Solution proposal

Computer programs can be seem as state machines, on which states are transitioned based on inputted data until reaching the final state, thus outputing the computation result, as shown in Figure 3. If the machine is not well modelled or implemented, undesired transitions can lead to unexpected states, which corresponds to bugs.

In memory, computer programs are organized as a sequence of instructions. Each instruction is responsible to change the current program state. The order instructions are executed define the output result. In the state machine analogy, branch instructions are the transition functions, as shown in Figure 3, thus monitoring them allows us to understand which states the program is leaving and entering.

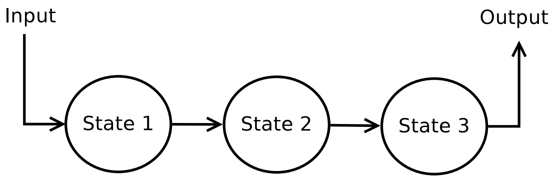

Figure 1. Program as a state machine. Data is inputed to an initial state and transitions will lead to the final state, outputing the result.

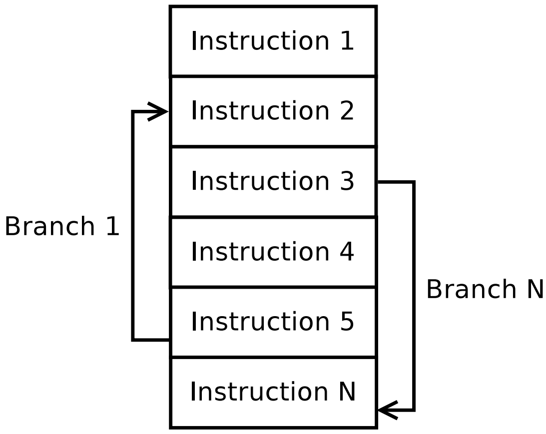

Figure 2. Program on memory. Branch instructions are responsible for state transitions.

When a program is executing a task (opening a file, for instance), we can see the state machine are traversing a given path, represented by branches. On a general way, always the same task is performed, the same path is traversed, thus the same branches are executed. Therefore, If a sequence of known branches is succeeded by unknown ones, the programming is behaving–which may be related to a identified bug or even to an exploitation attempt. If we were able to known which are the usually executed branches and runtime check the executed one, we would be able to detected application misbehavior events in runtime. In this work, we present a monitoring mechanism and a learning solution for this task.

To implement such solution, we propose a two-phase mechanism: In the first step, a profiling phase, our solution learns which are the allowed branches; In the second step, the matching phase, our solution monitors the taken branches and matches them against the database of allowed branches previously learned, as shown in Figure 3. When a violation is detected, a warning is raised.

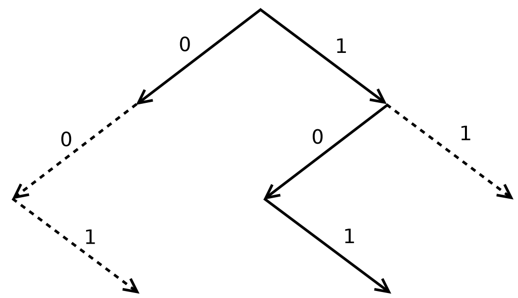

Figure 3. Expected Branches Policy. The solid arrows represent paths previously seem, thus representing expected branches. The dotted arrows represent so-far unknown branches, which might indicate a misbehavior.

In practice, such proposal presents a significant challenge: Despite taking almost the same branches while performing the same task, the taken paths are data-dependent. `IF-ELSE` constructions for odd/even values, for instance, may lead to distinct intermediary paths for a more complex operation. Our proposal to tackle these cases is to rely on user interactions to achieve a good code coverage. The hypothesis behind such decisions is that the users will have exercised the most common paths after some time and that the same paths will continuously be executed in the future. We consider this hypothesis as reasonable as, in fact, the programs often executes the set of instructions (hot code regions), such as loops [Gordon-Ross and Vahid 2003].

Therefore, as both the training phase as well as the monitoring consist on monitoring taken branches during user interaction, we can build an unified framework, differing only on the applied policy (learning or matching), which makes solution more flexible.

Once a violation is detected, our solution notifies an upper instance about its occurrence. Such upper instance can be an Antivirus solution, an O.S. subsystem, and so on. For the sake of evaluation, we implemented an intelligence model able to apply two distinct policies: i) On the a strict model, any unexpected branch is considered as a flow violation. Whereas very effective, this mode require a huge effort regarding training to increase the coverage and thus not generate false positives; ii) On the more flexible policy, we do not look to single branches, but a series of them, by relying on a moving window. It allows relaxing the training requirements while still detecting flow violations–ROP payloads, for instance, are composed by a sequence of gadgets terminated by branches (`RET`) [Göktaş et al. 2014]. On both policies, the misbehaving program is terminated when the violation is detected. The number of unexpected branches within a given window to define the execution as a flow violation is given by a defined threshold. A threshold of 1 would turn the solution back to the strict mode.

The presented operation modes rely on both automated learning (non-supervised) and detection phases. An even more relaxed policy could be implemented by adding user interaction to the system: When a violation occurs, for instance, the user could be prompted to decide whether the process should be terminated or the taken branches were correct. In this case, the solution would add the so-far unknown branches to the database, which would convert our solution on a semi-supervised approach.

4. Implementation

In this section, we present implementation details regarding the monitoring solution as well as the learning mechanism.

4.1. Data collection

To collect data from the processor branch monitor, we relied on an framework available on Github **blinded**. We have set the solution to monitor only the code image section from the target binary, filtered by its PID.

As our running operating system is ASLR-enabled, branches from distinct executions would be not directly comparable, as base addresses would differ. To allow comparison, we have performed an address normalization procedure, discarding branch base addresses and considering only their offsets inside the code images–thus unique regardless of execution. The effect of such procedure is shown in Table 1.

Table 1. ASLR-aware data collection. Offset normalization. Observe that despite the image base address, the branch offsets are unique.

Branch	Execution 1	Execution 2	Execution N	Offset
I	0x7FF1D30	0x7FF3D30	0x7FF5D80	0x1D30
II	0x7FF1E30	0x7FF3E30	0x7FF5E80	0x1E30
II	0x7FF1EF0	0x7FF3EF0	0x7FF5F40	0x1EF0

4.2. Automated Learning Approach

As the branch-based framework is able to provide us with all required branch information, we need only to collect data from a given path (source and target addresses) and store them on a database of allowed branches. For each taken branch address, we store their immediate successors, on a multi-level hash structure, as shown in the Figure 4.2. The hash indexing allows us to check branches presence at O(1) at the same time we do not have to worry about repeated entries.

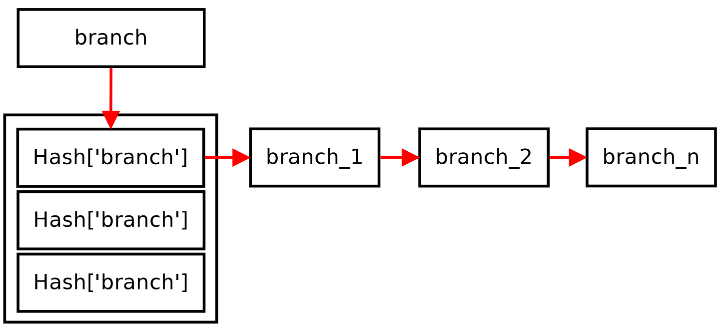

Figure 4. Branch Database. Source addresses are used to index allowed target addresses. Unidentified entries are considered as unexpected branches.

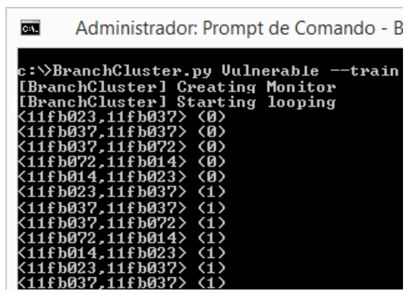

Figure 5. Automated learning. Flags 0 and 1 indicates whether a given branch was expected (allowed) to occur or not.

Figure 4.2 shows an example of the training procedure in action while learning the allowed paths from the code show in Listing 1. The `zero` (0) flag indicates the first taken branches were unknown–as the database was not previously populated–thus making the system to learn. In the second time they have appeared, the system had such data in the database, as shown by the `hit` (1) flag.

4.3. Detection

In the detection phase, the data is sampled on a moving window. For each taken branch within a given moving window, the next allowed branches are looked for on the database. If the following branch instruction is found, the `allowed` flag is set. otherwise, the `not_allowed` flag is set. This procedure is repeated for all window instructions. The ratio of `not_allowed` over `allowed` branches is compared against a threshold (according the considered policy), thus leading to the misbehavior conclusion in case of a high score. Figure 4.4 shows the detection window of a given branch violation.

4.4. Semi-Supervised Learning Approach

This approach can be considered as an extension of the detection mechanism. However, when a violation is identified, the monitored programmed is not immediately terminated, but the user is asked to decide which action will be taken. If the user specifies the unexpected branches are allowed, the solution adds them to the database. The next time these branches were executed, they will be considered as expected, thus not triggering warning anymore. Figure 4.4 shows the solution asking user to validate a given violation detection as a true violation.

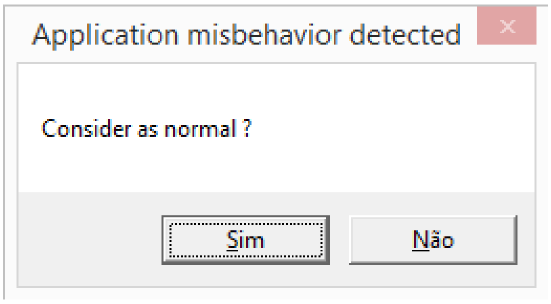

Figure 6. Semi-supervised learning. Solution asks for user confirmation.

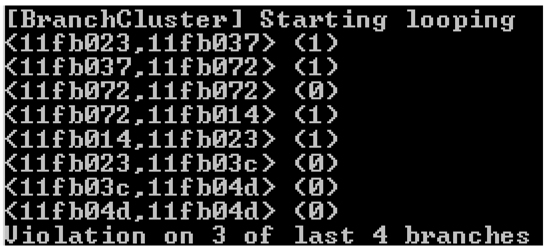

Figure 7. Misbehavior Detection. Solution detects violations using a threshold value over data from a moving window.

5. Evaluation

In this section, we evaluate our solution's proper working. We first validated it with a synthetic example to demonstrate its correctness. Secondly, we evaluate its use agaisnt a real world exploit, demonstrating its application on practical scenarios.

5.1. Validation

To validate our proposal, we have developed the synthetic code example presented in Listing 1:

Listing. Validation code. It presents three distinct paths. The latter also presents an overflow vulnerability which allows the execution of so-far unreachable code.

```
main(){
  char string[MAX_STRING];
  int loop=0;
  int opt=0;
  do{
    scanf("%d",&opt);
    if (opt>0){
      printf("Greater than zero\n");
    }else if(opt<0){
      printf("Smaller than zero\n");
    }else{
      printf("Bad choice\n");
      // An string overflow here
      // changes the loop control
         variable
      scanf("%s",string);
    }
  }while(!loop);
  printf("Should never be executed\n");
```

This code presents 3 main decision paths:

1. Numbers greater than zero trigger the first `printf` as they follow only the first path and return to the loop.

2. Numbers smaller than zero trigger the second `printf` as they follow only the second path amd return to the loop.

3. The number zero triggers the third path, which fills the buffer with a string. There is a clear buffer overflow at `scanf` statement, as the string is allocated in the stack right before the `loop` control variable. Overflowing such variable would flip `loop` bits, causing execution to exit the loop, thus calling the last `printf`, which should had never be executed.

Our test procedure consisted in the following steps: i) training the solution for the paths 1 and 2; ii) running the solution with the trained dataset and ensure both trained paths are properly flagged; iii) exercise the third path, with no overflow, which should trigger the supervised learning, since this path was not trained in the last step; iv) exercise the same path, without triggering detection, since the solution learned this path as allowed in the last step; v) finally, exercise the third path with overflow, thus triggering the detection mechanism, forcing application to quit. Our solution has proven to be able to pass all described tests.

5.2. Real Application

We also evaluated our solution's effectiveness on a real scenario. To do so, we launched a real ROP attack, based on known exploit [Knaps 2015], against a real-world software (Easy File Share). After the learning step, the software was monitored while being exploited. An excerpt of the result of such monitoring is shown in Listing 2.

Listing 2. Real application under a ROP-based attack. Differences between the expected and the observed branches.

```
Unexpected Branches: [0x150C, 0x1C80C, 0x13020]
Unexpected Branches: []
Unexpected Branches: [0x1731A, 0xD31A, 0x7C81A, 0x33B1A, 0x2AC1A
    , 0xFC21A, 0x12941A, 0x29A1A]
```

We notice while some branches sources were succeeded by the same branch targets than in the training step, thus triggering an empty difference set, some branches were followed by unexpected branches, showing our solution's ability to detect abnormal behavior in real scenarios.

5.2.1. Exploit or crash ?

In addition to enabling real time detection, the presented experiment suggested our solution can also be used on other contexts. Inspired on BranchTrace [Willems et al. 2012], we believe crash reports can be enhanced by including branch data. As an example, consider the execution of the vulnerable file sharing application as already presented. On a given point, the branch window present the following consecutive target branches: `0x34A3 (1)` and `0x6fB8 (0)`. This means the first branch target was expected whereas the second was an unexpected branch resulting from the execution of the previous code block, as well as its successor branches. If such information were submitted to the application developers, it could help them to find the bug cause easier, because the buggy construction is probably located around the unexpected branch. In fact, by disassembling the code around these branches, we identified the following pieces of code, as shown in Listing 5.2.1.

The legitimate call starts executing at `line 1`. This function body is responsible for manipulating the stack and then calling a function pointed by the `ecx` register (line `line 12`), previously loaded from the stack (line `11`). Since this branch leads to an unexpected target, it suggests the stack may have been corrupted. The call target code pops value from the stack and returns (line `13-15`). The execution jumps to an unknown location (line `16`), followed by a nop sled (line `17`), until reaching another code portion (line `17`). In fact, by knowing the exploit, we can verify the stack was effectively corrupted, the `pop-pop-ret` sequence is a ROP-like payload, followed by the usual `JMP` to the payload, which aligns itself through `NOPs`, until reaching the shellcode.

5.3. Overhead

Whereas effective, we must ensure our developed solution does not impose significant overhead, so the original application keeps performing well. From the data capture point of view, as our solution is based on the BranchMonitor framework, our solution's overhead is bounded by performance penalty imposed by it (around a mean of 20%). We observe, however, the framework was originally set to use a small interrupt threshold of 1 instruction, which increases performance penalty. We can reduce the performance penalty by using a larger interrupted threshold, as we do not need branch-by-branch support.

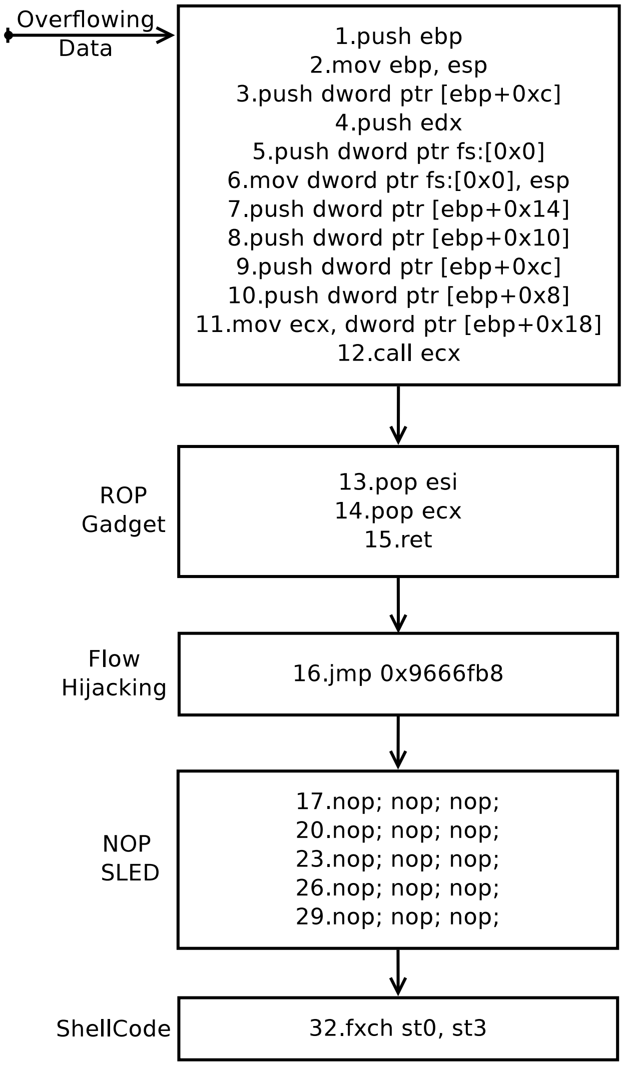

Figure 8. Exploit Execution. After a buffer overflow, the stack holds user-injected addresses which are used to redirect the flow to a malicious code portion acting as a shellcode-analogous.

In practice, our measurements indicates the overhead is application, context-dependent, as each one presents a distinct rate of taken branches. A high-branch-density application will be more interrupted than an I/O-bound application, for instance. On our tests, the `vulnerable` test program presented the lowest overhead footprint, as all their branches fit on a single O.S page. In the other hand, the Chrome browser execution is severely penalized, because each newly opened tab creates a new system process, thus executing a large amount of branches.

We also observed distinct performance penalty impacts according the core the monitored application is running. Operating systems schedulers often make the CPU core 0 to present high loads whereas the other cores present lower ones. When running on a heavy-loaded core, such as core 0, the overhead is increased, because our solution has to filter out much more data–we remark the framework captures data in a system-wide way. In the other hand, the solution presents smaller overhead when running in the other cores.

6. Discussion

In this section, we discuss the impact of the proposed solution and how this could be integrated on real, practical systems.

6.1. Advances, Implications and Limitations

Our solutions is the first attempt to solve the misbehavior detection problem by using hardware assistance. It enables us the perform the task without significant performance penalty, which is a hard-to-achieve requirement for such class of solution.

We demonstrated our solution is feasible on real cases. Such development implicates on an increased bug detection capabilities on many branch-monitor-equipped systems. Moreover, as it is based on a hardware-feature, it is able to monitor Commercial- Off-The-Shelf (COTS) binaries, without code instrumentation and/or modification. We believe this is a significant contribution towards increased bug detection.

As short-term limitation, the branch monitor is limited to collect data on a system-wide manner, thus suffering the overhead of monitoring all running processes prior the filtering mechanism. This limitation can be overcome by using newcoming hardware features, such as the processor tracer [R. 2013], present on newer processor families.

On the long-term, we pinpoint the major challenge such kind of approach is subject is the solution's capability to decide whether a given misbehavior is derived from an exploit or from an ordinary bug. Such information is critical to perform real time threat detection on a more complete way, thus turning our solution into a antivirus solution, for instance. Such development is beyond implementation constraints as it is also limited by theoretical aspects.

6.2. O.S. self-monitoring proposal

Our presented evaluation demonstrated our solution is do able to handle application misbehavior without any prior written rules. Although the results are preliminary, we understand many systems and applications could benefit from using such kind of approach.

As an example, we suggest using our approach for operating systems self-monitoring. Currently, modern operating systems already collect telemetry data [ZDNet 2016] from applications. These systems could also be extended to monitor application execution to profile them and detect abnormal behaviors, as proposed by us.

In addition to detection, the systems could also be able to launch automatic remediation procedures, such as automatic backup recovery or some other system configuration restore, when a misbehavior were detected.

Even on unrecoverable cases, the profiled data could be sent to application maintainers as part of bug/crash reports, enriching the existing fault data collection mechanisms, so developers would be able to more precisely locate which instruction block triggered the faulting behavior.

6.3. Usage scenarios and Policies

Our proposed solution is suitable to operate on distinct scenarios. As an example, its permanent, real-time capabilities make it a candidate to monitor critical systems, where faults must be immediately identified.

In addition, we believe there is a real demanding field regarding the monitoring of recently installed applications, third-party software components and on unpatched systems. Our system could be launched by the O.S., for instance, when a new application is installed, thus starting the profiling step. After some time monitoring the application without any significant occurrence, the mechanism could be turned off.

6.4. A Cooperative Learning Model

Our solution relies on user interaction to learn the allowed branches. On the one hand, it covers most of exercised behavior by a given user, regardless of the usage pattern. On the other hand, rare but legitimate paths may be not learned if not previously exercised.

Whereas such coverage may be enough for most cases, some scenario may require a really increased branch coverage. Therefore, more branches should be exercised, thus making our approach closer to the fuzzing ones. However, as we intend to provide an alternative to these, it is not reasonable to generate multiple random inputs to exercise our solution. Thus, an alternative approach must be developed.

A way of achieving high coverage is to perform a distributed learning procedure, grouping the branches taken on multiple machines. It could be implemented by regularly sending the newly learned branches to a remote repository and downloading new allowed paths definitions, as an next-generation antivirus definition update.

We believe such kind of implementation is more more feasible for immediate application on a mobile software ecosystem, because their application stores already update applications' configurations based on performance data [Google].

6.5. Future Work

As future work, we will extend our framework in two ways: i) by investigating new, additional hardware features which could enable us to profile applications and establish an execution baseline without significant performance impact; ii) by extending our threat intelligence component to perform more complex matches. Machine-learning solutions could be used, for instance, to predict whether given unexpected branches constitute a violation or not.

7. Conclusions

In this paper, we have presented a hardware-assisted, branch learning solution able to infer application misbehavior by comparing a given branch trace to a learned profile. We evaluated the proposed solution with conceptual and real applications demonstrating its ability to handle buggy and exploitable software. We also discussed the conceptual application of such solution as an Operating System built-in feature.

We will release our framework as an open source solution. To keep the double blind review process, the code samples here presented as well as the developed kernel drivers and prototypes will be published after publication acceptance.

References

Akiyama, S. and Hirofuchi, T. (2017). Quantitative evaluation of intel pebs overhead for online system-noise analysis. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*, ROSS '17, pages 3:1–3:8, New York, NY, USA. ACM.

Arora, D., Ravi, S., Raghunathan, A., and Jha, N. K. (2005). Secure embedded processing through hardware-assisted run-time monitoring. In *Design, Automation and Test in Europe*, pages 178–183 Vol. 1.

Böhme, M., Pham, V.-T., and Roychoudhury, A. (2016). Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA. ACM.

Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H. (2014). Ropecker: A generic and practical approach for defending against rop attacks. .

Das, S., Xiao, H., Liu, Y., and Zhang, W. (2016). Online malware defense using attack behavior model. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1322–1325.

DHS (2013). Software assurance. https://www.us-cert.gov/sites/default/files/publications/infosheet_SoftwareAssurance.pdf.

DoD (2005). Secure software engineering. http://www.sis.pitt.edu/jjoshi/Devsec/secureSoftware.pdf.

Forbes (2017). Google just discovered a massive web leak... and you might want to change all your passwords. https://www.forbes.com/sites/thomasbrewster/2017/02/24/

google-just-discovered-a-massive-web-leak-and-you-might-want-to-chang
#50e20e923ca3.

Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. (2014). 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. USENIX Association.

Google. Política de privacidade do google. https://policies.google.com/privacy.

Gordon-Ross, A. and Vahid, F. (2003). Frequent loop detection using efficient non-intrusive on-chip hardware. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 117–124, New York, NY, USA. ACM.

Guardian, T. (2015). Us aviation authority: Boeing 787 bug could cause 'loss of control'. https://www.theguardian.com/business/2015/may/01/us-aviation-authority-boeing-787-dreamliner-bug-could-cause-loss-of-c

Kaspersky (2017). Era of exploits: number of attacks using software vulnerabilities on the rise. https://usa.kaspersky.com/about/press-releases/2017_era-of-exploits-number-of-attacks--using-software-vulnerabilities-on-

Knaps (2015). Easy file sharing web server 7.2 - remote buffer overflow (seh) (dep bypass with rop). https://www.exploit-db.com/exploits/38829/. Access Date: 2017.

Li, Y., Chen, B., Chandramohan, M., Lin, S.-W., Liu, Y., and Tiu, A. (2017). Steelix: Program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 627–637, New York, NY, USA. ACM.

Morana, M. (2010). Vulnerability analysis, secure development and risk management of web 2.0 applications. https://www.owasp.org/images/c/c1/OWASP_Cincy_Web2_Threats_and_Countermeasures.pdf.

Ozsoy, M., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., and Ponomarev, D. (2015). Malware-aware processors: A framework for efficient online malware detection. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 651–661.

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent ROP exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, Washington, D.C. USENIX.

Pham, V.-T., Böhme, M., and Roychoudhury, A. (2016). Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 543–553, New York, NY, USA. ACM.

R., J. (2013). Processor tracing. `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`. Access Date: May/2017.

Register, T. (2011). Finance software bug causes $217m in investor losses. `https://www.theregister.co.uk/2011/09/22/software_bug_fine`.

Shye, A., Iyer, M., Reddi, V. J., and Connors, D. A. (2005). Code coverage testing using hardware performance monitoring support. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 159–163, New York, NY, USA. ACM.

Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G. (2014). Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA. USENIX Association.

Willems, C., Hund, R., Fobian, A., Felsch, D., Holz, T., and Vasudevan, A. (2012). Down to the bare metal: Using processor features for binary analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 189–198, New York, NY, USA. ACM.

ZDNet (2016). Windows 10 telemetry secrets: Where, when, and why microsoft collects your data. `http://www.zdnet.com/article/windows-10-telemetry-secrets/`.

Zhang, T., Zhuang, X., Pande, S., and Lee, W. (2004). Hardware supported anomaly detection: down to the control flow level. `http://hdl.handle.net/1853/96`.