MMU-based Access Control for Libraries

Marinos Tsantekidis[1][a] and Vassilis Prevelakis[2]
[1]*Institute of Computer Science - FORTH, Greece*
[2]*AEGIS IT RESEARCH GmbH, Germany*

Keywords: Runtime Monitoring, Library Calls, Access Control Gates, Security Policies.

Abstract: Code Reuse Attacks can trick the CPU into performing some actions not originally intended by the running program. This is due to the fact that the execution can move anywhere within a process's executable memory area, as well as the absence of policy checks when a transfer is performed. In our effort to defend against this type of attacks, in an earlier paper we present a Proof-of-Concept mitigation technique based on a modified Linux kernel where each library - either dynamically or statically linked - constitutes a separate code region. The idea behind this technique is to compartmentalize memory in order to control access to the different memory segments, through a *gate*. Taking our previous work one step further, in this paper we present an updated version of our kernel-side technique, where we implement security policies in order to identify suspicious behavior and take some action accordingly.

1 INTRODUCTION

As the advancement of technology offers capabilities which result in attackers on every level getting more competent and effective, attacks have become more elaborate. Therefore, we need to establish an adequate level of security in software systems. As the complexity of these systems is ever-increasing, we have not seen a commensurate improvement in code design and development. Hence, systems are becoming ever more vulnerable to attacks at multiple levels. Complete security of a program is unfeasible. Conceding that vulnerable code will be included in production systems, there is a need to detect when one or more of these vulnerabilities is exploited by an attacker. By monitoring the behavior of a program, we can detect any deviations from nominal operation. Once a program goes off-nominal we need to determine whether the cause is security-related and, if so, take appropriate action. Our idea is to implement such actions at an abstract level, between the Operating System (OS) and a running application.

Unlike most run-time security mechanisms, where violations in most cases lead to the termination of the offending process, the call intercept technique offers a variety of options in dealing with the security breach. For example, `systrace(8)` (Provos, 2003) may rewrite function arguments (e.g. truncate or replace strings), or return an error without making the call. A key concern is the level at which the call intercept should be made. Some papers propose to carry out the checks at the machine code level - via call graphs (e.g., CFI (Abadi et al., 2005) and variants), while others at the system call level (e.g., `systrace(8)`). However, because these systems perform the checks at fairly low level, they are quite labor intensive and error prone and exhibit high overhead. Coarse-grained CFI techniques mainly try to deal with the increased performance issues of CFI, trading off security and being left vulnerable to motivated adversaries. Finer-grained solutions provide higher security, although even in this case studies (Evans et al., 2015) have shown that some implementations prove ineffective in defending against code-reuse attacks (CRAs).

Contributions. In this paper, we present our hypothesis that by carrying out behavioral monitoring at a higher level, we can create a flow diagram that is closer to the program logic and hence more understandable. We have chosen the library function call level as the ideal boundary to install our checks, because libraries are used by the main program but are not part of its execution logic. We present an updated version of our previous Proof-of-Concept mitigation technique, behind which the idea is to compartmentalize memory in order to provide access control to

[a] https://orcid.org/0000-0001-6710-5972

Tsantekidis, M. and Prevelakis, V.
MMU-based Access Control for Libraries.
DOI: 10.5220/0010536706860691
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 686-691
ISBN: 978-989-758-524-1

memory segments based on security policies. We extend our previous approach by installing a special custom library, a *gate*, one for each separate segment and redirect the flow of execution through this gate where we enforce the relevant security policies. Our mitigation technique requires kernel support, which is on of its strengths since we can manipulate the memory securely and efficiently. We chose to leverage the Memory Management Unit (MMU) and specifically the NX-bit, because it is easier to us due to our previous development experience. To the best of our knowledge, it is the first time this is performed in this way. Furthermore, our mechanism can be implemented on top of other defense mechanisms.

The remainder of this paper is organized in the following manner: Section 2 offers a brief summary of the timeline of CRAs and related defenses. In Section 3 we detail the design of our approach. In Section 4, we describe the implementation specifics. In Section 5, we evaluate our custom kernel both in terms of effectiveness and performance. Discussion and conclusion are provided in Section 6.

2 BACKGROUND AND RELATED WORK

Code Reuse Attacks (CRA) work on the premise that an adversary can use code already present in a program's memory space in order to mislead the CPU to perform unintended by the program actions. Sophisticated approaches, such as Return Oriented Programming (ROP) (Shacham, 2007; Checkoway et al., 2010; Roemer et al., 2012) and Jump Oriented Programming (JOP) (Bletsch et al., 2011), form snippets of code, "gadgets", chaining together legitimate commands already in memory, eventually forming a sequence of gadgets that perform the unauthorized action desired by the attacker. Initially, CRAs were based on the principle that gadgets are located at known addresses in memory. Address Space Layout Randomization (ASLR) (PaX, 2001) was introduced as a defense measure, but was eventually bypassed (Shacham et al., 2004).

Furthermore, Snow et al. introduced "Just-In-Time" ROP (JIT-ROP) (Snow et al., 2013) which maps an application's memory layout on-the-fly, thus bypassing ASLR. Next, it constructs and delivers a ROP payload based on collected gadgets.

Later, Bittau et al. presented Blind Return Oriented Programming (BROP) (Bittau et al., 2014). BROP works against modern 64-bit Linux with ASLR, NX memory and stack canaries (Wagle and Cowan, 2003) enabled. It can remotely find enough gadgets to perform the `write(2)` system call and then transfer the application's binary from memory to the attacker's socket.

Over the years, important work has been carried out with respect to defenses against CRAs. Backes and Nürnberger developed Oxymoron (Backes and Nürnberger, 2014) to counter JIT-ROP attacks. Oxymoron combines two methods: fine-grained memory randomization with the ability to share the entire code among other processes. However, in order for this defense to be effective, an executable along with all of its shared libraries need to be instrumented using Oxymoron (Mithra and Vipin P., 2015). In contrast, our approach is completely transparent, since we require no access to the executable or its shared libraries.

Venkat, et al. (Venkat et al., 2016) propose a security defense called HIPStR to thwart (JIT-)ROP. HIPStR performs dynamic randomization of run-time program state. However, this measure requires extensive run-time support to piece the randomized instructions back together, imposing significant overhead (around 15% at best), counter to our mechanism which leaves only a minimal performance footprint.

The `systrace(8)` (Provos, 2003) system which supports fine-grained policy generation, guards the calls to the operating system at the lowest level, enforcing policies that restrict the actions an attacker can take to compromise a system. However, this fine-grain control is overly verbose. Additionally, it may leave a library in an inconsistent state if a misconfiguration interrupts the sequence of these calls in the middle of execution (Kim and Prevelakis, 2006). Later research (Watson, 2007) showed that concurrency vulnerabilities were discovered, that gave an attacker the ability to construct a race between the engine and a malicious process to bypass protections. Our technique operates at a higher level, away from low level calls, closer to the application logic, facilitating the generation of policies and being more lightweight in this way.

Access Controls For Libraries and Modules (SecModule) (Kim and Prevelakis, 2006) forces user-level code to perform library calls only via a library policy enforcement engine providing mandatory policy checks over not just system calls, as in the case of `systrace(8)`, but calls to user-level libraries as well. Access to a specific function or procedure is controlled by the kernel, however the overhead of two context switches per function invocation (caller to kernel and kernel to caller), makes the technique quite expensive for more general use. One of the issues identified by the authors of the SecModule paper is the difficulty in encapsulating library modules which was error prone and extremely labor intensive. Another issue is the inability to evaluate call arguments. Although they are contained in a known structure pointed to by a stack pointer, their examination requires lots of casting in the C++ functions. In contrast, our approach automatically encloses libraries in a seamless generic way, with low overhead as well as the ability to evaluate call arguments.

Abadi et al. proposed Control-Flow Intetegrity (CFI) (Abadi et al., 2005) which enforces the execution of a program to adhere to a control flow graph (CFG), which is statically computed at compile time. If the flow of execution does not follow the predetermined CFG, an attack is detected. CRAs such as ROP and JOP by definition divert the program flow, hence they are discovered. This approach, however, suffers from two main disadvantages. First, computing a complete and accurate CFG is difficult since there are many indirect control flow transfers (jumps, returns, etc.) or libraries dynamically linked at run-time. Furthermore, the interception and checking of all the control transfers incur substantial performance overhead. Our mechanism, although implementing a kind of CFI, deals with high-level calls to library functions (away from the low level instructions) with minimal run-time performance cost.

In (Kayaalp et al., 2012) Kayaalp et al. propose Branch Regulation (BR) which limits control flow transfers to specific legitimate points, disallowing - in effect - arbitrary transfers from one function into the middle of another. However, it annotates the binary, resulting in increased code size. In addition, the shadow stack used by the mechanism is not secure since it resides in mapped memory (Christoulakis et al., 2016). Moreover, when a legitimate direct jump is performed (e.g. `longjmp(3)`) it may transfer the control flow in the middle of a function, thus throwing a false-positive exception. The main advantage of our mechanism is that it is source/binary-agnostic.

In our previous work (Tsantekidis and Prevelakis, 2017), we implement access control similar to the work presented above. Under our scheme, each call to an external function of a library is intercepted and its arguments are examined to determine whether the control flow transfer is warranted. This allows high-level checks to be carried out in a similar fashion to the `systrace(8)` engine (Provos, 2003) – which, however operates at the system call level – and Sec-Module (Kim and Prevelakis, 2006) that introduces a form of authentication when calling functions from a library. Our approach is transparent to the user applications, since it can work in the background without the applications ever knowing its existence. Furthermore, there is no need to have or modify the source code of any application, making the mechanism suitable for legacy applications as well as binary programs. Since this is a user-space mechanism, there is no need to invoke the kernel, meaning there are no context switches, which saves performance overhead. Additionally, it can be easily adopted in real-life environments. However, our approach is not without its disadvantages. It relies on dynamic library hooking, so it is unsuitable to trace statically-linked applications. There is, also, a slight increase in code size compared to the original library version. Moreover, it can only protect programs written in C/C++ and it can be bypassed by a highly-skilled attacker who is aware of its existence and is able to manipulate the program's memory.

3 DESIGN

The goal of our proposed approach is to thwart control-flow hijacking attacks by segregating a process's executable areas which correspond to external libraries or the main executable and by imposing strict control over any attempt to invoke such an area, through a policy enforcement engine. In order to achieve this, we set a number of requirements that our mechanism must fulfill:

(R1) *Effectiveness:* Intercept every attempt to access a protected region and redirect it inside the associated gate.

(R2) *Transparency:* Allow applications to continue working as originally intended by the developer, while our access control mechanism delivers a secure run-time environment.

(R3) *Efficiency:* Minimize run-time and memory overhead imposed by our implementation.

To abide by these requirements, we propose a customized Linux kernel that leverages the MMU in order to separate the memory of a process into regions, based on the libraries that are loaded upon a program's execution. When an untrusted, user-space application issues a call to a protected library, our custom kernel intercepts it and redirects it through a *gate* library - which stands as a policy decision mechanism - before allowing it to continue.

In order to meet requirement (R1) and intercept all calls, our system marks all the executable regions (e.g *.text*/code region) that correspond to a linked library or the main executable as non-executable. Furthermore, for each region it maps a special custom library - which we designate as *gate* - in the running process's address space. When an application or other region issues a call to a non-executable region, a deliberate

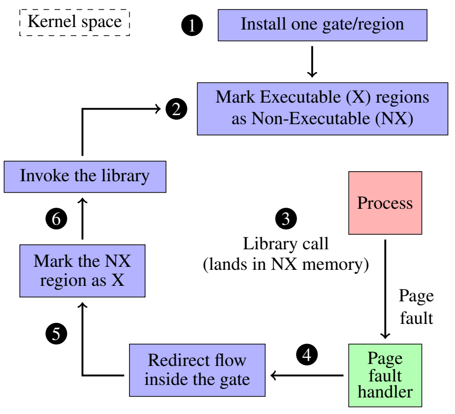
Kernel space
❶ Install one gate/region
❷ Mark Executable (X) regions as Non-Executable (NX)
Invoke the library
Process
❸ Library call (lands in NX memory)
Page fault
❻
Mark the NX region as X
❺
Redirect flow inside the gate
❹
Page fault handler

Figure 1: Design overview.

page fault occurs. The Page Fault Exception Handler (PFEH) (Bovet and Cesati, 2005) is, then, invoked to resolve the issue. In order to continue executing, we modify the PFEH so it redirects the flow inside the associated *gate* library that we had previously mapped in the process's memory area. If the policy check is passed, it marks the specific non-executable region as executable and continues with the call, starting the whole procedure from the beginning. Figure 1 shows this sequence of events. This way we are able to intercept all the calls inside the libraries that an application uses and enforce the execution flow through the policy decision engine.

Threat Model. We consider an adversary model which is composed of two classes of threats. The first class of threats comprises the bugs/vulnerabilities that an application contains. These can be either (a) already discovered and known without, however, a security patch being available or having being applied by the system administrator or (b) zero-day/undiscovered, without anyone knowing about their presence. An adversary is in possession of a technique (e.g. BROP, JIT-ROP, etc.) that can exploit one ore more of these vulnerabilities and mount a CRA that eventually compromises the application. Our mechanism can detect and immediately mitigate any attempt to exploit these bugs until a patch can be applied, by hindering the execution of collected gadgets.

The second threat class is the behavior of a program. At run-time, an application can present abnormal behavior that opposes what is nominally expected. Our mechanism can observe not only the occurrence of calls, but also the number of calls to a particular service or function, as well as the sequence of these calls (similarly to our previous work in (Tsantekidis and Prevelakis, 2019)), in order to create an understanding of the different states that an application is going through during its execution.

Security Policies. The main challenge of our approach is to create comprehensive security policies that include the complete list of communication paths among the different segments (libraries/executables) in which we compartmentalize the running application. This challenge originates from the complexity of the library/application (i.e. its development as well as its functionality), the way that the paths are provided (e.g., by the developers of the library/application) and the effort required to track these paths.

Based on the value of a property (e.g., the access rights of a file, the number/sequence of function calls, etc.), if our mechanism detects a suspicious activity which is further characterized as malicious, it applies the corresponding security measure defined in the policy. The evaluation of the security policies is continuous, throughout the execution of the monitored application and is triggered whenever a call to a separate memory region is performed.

4 IMPLEMENTATION

We provide an updated version of the Proof-of-Concept (PoC) implementation of our previous prototype mechanism (Tsantekidis and Prevelakis, 2019), using Linux kernel version 4.16.7, the latest one at the start of the development phase.

Our mechanism identifies the regions of contiguous virtual memory (Virtual Memory Areas - VMAs) that are executable and correspond to a linked library or the main executable and maps the custom *gate* library in the process's memory, one for each identified VMA. Then, these VMAs are marked as non-executable wich is computationally inexpensive, moving towards meeting requirement (R3). From then on, when a process tries to perform a library call, it will land on an address in a non-executable area, which will deliberately produce a page fault. Before handling it, we redirect the execution flow inside our policy enforcement engine in order to check the security policies associated with the running program. This procedure is performed automatically on the kernel side, without requiring access to the source code/binary of the application or linked libraries, thus making our approach completely transparent, fulfilling requirement (R2).

In this updated version of our previous work, we introduce the notion of *gate* libraries as a policy enforcement engine, which we install one for each identified executable area. Every call originating from a region different than the destination one (i.e. between different libraries) is intercepted and analyzed, even if the same call was previously checked again at another point during execution. We can express the security policies based on the outline presented in Section 3, using a policy definition language (such as KeyNote (Blaze et al., 1999)). Alternatively, the library/application designer can provide us with the policy in a similar format/language.

5 USE CASE

In order to evaluate the applicability and efficiency of our proposed prototype, we analyze a use-case where the NGINX HTTP server is compromised by Denial-of-Service (DoS)/arbitrary code execution attack, after a vulnerability is triggered.

CVE-2013-2028 (Common Vulnerabilities and Exposures, 2013) is a stack-based buffer overflow related to the chunk size of an HTTP request with the header `Transfer- Encoding:chunked`. Although it was addressed in versions later than 1.4.0, we can use it to examine the effectiveness our prototype in defending against CRAs. Due to its simplicity, we chose the CRA (ROP) attack described in (sorbo, 2013), which *is a generic exploit for the 64-bit NGINX server and uses the Blind ROP attack technique (Bittau et al., 2014)*. Here we use our previous technique in (Tsantekidis and Prevelakis, 2019) to extract the legitimate addresses from the vulnerable NGINX binary, in order to form the minimal security policy.

Additionally, as the exploitation unfolds, we monitor its progress and we manage to identify a characteristic sequence of calls to executables/libraries (including the internal ones) and produce a trail of it. The security policy can subsequently be updated to reflect this information, so when the *gate* enforces the policy, it will recognize the same fingerprint and be able to intercept and mitigate the attack.

We leverage the Phoronix Test Suite (PTS) (pts, 2021) to evaluate the efficiency of our mechanism. Specifically, we run the OpenSSL benchmark test. Our test-bed is a machine with 16Gb of RAM and an AMD FX-8370 Eight-Core processor, running Ubuntu 16.04 x86_64 and Linux kernel version 4.16.7. We run the benchmark test for both cases: (a) default untouched kernel, (b) kernel customized with our mechanism. The outcome reports on the number of RSA 4096-bit sign operations per second. Figure 2 summarizes the results of the tests. As expected, the default setup is performing better. However, there is only minimal decrease in performance when using our custom kernel, of about 2%.

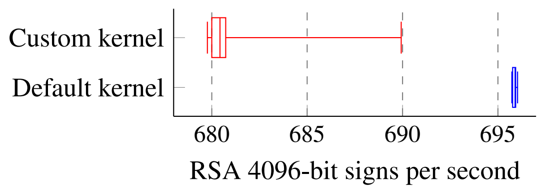

Figure 2: Performance evaluation of our mechanism using the OpenSSL benchmark of PTS.

6 DISCUSSION AND CONCLUSION

In this paper, we present a Proof-of-Concept prototype mitigation technique as an extension of our previous work in (Tsantekidis and Prevelakis, 2017; Tsantekidis and Prevelakis, 2019) that segregates the memory of a running process into regions at the granularity of executables/external libraries (irregardless of them being statically or dynamically linked), leveraging the MMU. With each of these regions we associate a special *gate* library that enforces security policies and redirect execution through it after intercepting calls to the regions. This way, we are able to monitor access to executables/libraries and change their functionality when there is suspicion/detection of foul play, based on security policies. Our approach is efficient and transparent and can be used on binary/legacy applications and existing environments, as well as serve as a complimentary measure of defense alongside already implemented mechanisms.

ACKNOWLEDGEMENTS

This work is supported by the European Commission through the following H2020 projects: SENTINEL under Grant Agreement No. 101021659, ROXANNE under Grant Agreement No. 833635, THREAT-ARREST under Grant Agreement No. 786890 and CONCORDIA under Grant Agreement No. 830927.

REFERENCES

(2021). Phoronix Test Suite. https://www.phoronix-test-suite.com.

Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2005). Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1102120.1102165.

Backes, M. and Nürnberger, S. (2014). Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 433–447, USA. USENIX Association.

Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., and Boneh, D. (2014). Hacking Blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242.

Blaze, M., Feigenbaum, J., Ioannidis, J., and Keromytis, A. (1999). The keynote trust-management system version 2. *RFC*, 2704:1–37.

Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. (2011). Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1966913.1966919.

Bovet, D. P. and Cesati, M. (2005). Page Fault Exception Handler. In *Understanding the Linux Kernel, 3rd Edition*, chapter 9.4. O'Reilly.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1866307.1866370.

Christoulakis, N., Christou, G., Athanasopoulos, E., and Ioannidis, S. (2016). HCFI: Hardware-enforced Control-Flow Integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 38–49, New York, NY, USA. ACM.

Common Vulnerabilities and Exposures (2013). CVE-2013-2028. https://www.cvedetails.com/cve/CVE-2013-2028/.

Evans, I., Fingeret, S., Gonzalez, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., and Okhravi, H. (2015). Missing the point(er): On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy*, pages 781–796.

Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., and Ponomarev, D. (2012). Branch regulation: Low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 94–105.

Kim, J. W. and Prevelakis, V. (2006). Base Line Performance Measurements of Access Controls for Libraries and Modules. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 356–356, Washington, DC, USA. IEEE Computer Society. http://dl.acm.org/citation.cfm?id=1898699.1898911.

Mithra, Z. and Vipin P. (2015). Evaluating the theoretical feasibility of an srop attack against oxymoron. In *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1872–1876.

PaX, T. (2001). Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt.

Provos, N. (2003). Improving host security with system call policies. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 18–18, Berkeley, CA, USA. USENIX Association. http://dl.acm.org/citation.cfm?id=1251353.1251371.

Roemer, R., Buchanan, E., Shacham, H., and Savage, S. (2012). Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34. http://doi.acm.org/10.1145/2133375.2133377.

Shacham, H. (2007). 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*, CCS '07, pages 552–561, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1315245.1315313.

Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA. ACM. http://doi.acm.org/10.1145/1030083.1030124.

Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. (2013). Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA. IEEE Computer Society. http://dx.doi.org/10.1109/SP.2013.45.

sorbo (2013). Nginx 1.4.0 (Generic Linux x64) - Remote Overflow. https://www.exploit-db.com/exploits/32277.

Tsantekidis, M. and Prevelakis, V. (2017). Library-Level Policy Enforcement. In *SECURWARE 2017, The Eleventh International Conference on Emerging Security Information, Systems and Technologies*, Rome, Italy. http://www.thinkmind.org/index.php?view=article&articleid=securware_2017_2_20_30034.

Tsantekidis, M. and Prevelakis, V. (2019). Efficient Monitoring of Library Call Invocation. In *Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 387–392, Granada, Spain. https://doi.org/10.1109/IOTSMS48152.2019.8939203.

Venkat, A., Shamasunder, S., Shacham, H., and Tullsen, D. M. (2016). HIPStR: Heterogeneous-ISA Program State Relocation. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 727–741, New York, NY, USA. ACM.

Wagle, P. and Cowan, C. (2003). Stackguard: Simple stack smash protection for gcc. In *Proc. of the GCC Developers Summit*, pages 243–255.

Watson, R. N. M. (2007). Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the First USENIX Workshop on Offensive Technologies*, WOOT '07, pages 2:1–2:8, Berkeley, CA, USA. USENIX Association. http://dl.acm.org/citation.cfm?id=1323276.1323278.