The Neverending Story: Memory Corruption 30 Years Later

Oscar Llorente-Vazquez[1], Igor Santos[2], Iker Pastor-Lopez[1], and Pablo Garcia Bringas[1]

[1] University of Deusto, Bilbao, Spain
{oscar.llorente, iker.pastor, pablo.garcia.bringas}@deusto.es,
[2] Mondragon Unibertsitatea, Arrasate-Mondragon, Spain
isantos@mondragon.edu

Abstract. Memory errors have been present in software for more than three decades now, leading to numerous security issues. Low-level languages like C and C++ that are prone to this class of errors are in widespread use, meaning that a large number of current systems are susceptible to attacks that target memory corruption vulnerabilities. Therefore, hardening programs and protecting against this type of attacks is of the utmost importance to build secure systems. This paper explains the basis of memory errors and their characteristics, analyzes the huge research effort that has been carried out in relation to memory error detection, exploitation and attack prevention, and depicts the most relevant approaches in this area.

1 Introduction

Among all the software errors that lead to security vulnerabilities, memory corruption is one of the most dangerous and oldest problems in which the systems security community has focused on, and despite the numerous approaches proposed to mitigate them, they are still found in generalized and specialized software [48, 10]. Memory corruption occupies the top positions in the CWE most dangerous software weaknesses list [11], appearing separately as out-of-bounds writes, out-of-bounds reads, Use-after-Frees (UAF), and so on.

Several factors influence the prevalence of memory errors, but one of the main reasons is that a large amount of proposed detection and protection solutions are not deployed in practice due to performance overhead, compatibility problems, lack of completeness in protection, or dependence conflicts [47, 44].

The primary intention of this paper is to provide a general understanding about memory errors, pointing out the most recent research in detection, exploitation and convenient defense methods.

The remainder of this paper is organized as follows. Section 2 depicts the basis of memory errors and classifies them in accordance with their properties. Section 3 systematizes the main approaches in the context of memory corruption detection. Then, section 4 describes software attacks that exploit memory corruption focusing on code-reuse attacks and data-only attacks, the two most

actively researched classes. Section 5 analyzes attack countermeasures. Section 6 reviews the research related to our work. Finally, section 7 concludes the paper.

2 Background

Memory corruption becomes apparent as a consequence of the lack of memory and type safety in low-level languages such as C and C++. Since these languages provide low level control over implementation details for example through manual memory management, they open up the possibility for the program to access memory in unintended ways.

It is commonly said that the execution of a program is memory safe as long as a set of memory errors never occur.

Memory errors include every access to undefined memory, i.e., memory that the program has not allocated or not initialized, but also accesses to not designated memory regions (i.e., a pointer dereference that does not point to its assigned memory area). According to their characteristics, they are commonly classified into spatial and temporal memory errors.

- **Spatial Memory Errors:** On the one hand, spatial errors comprise every memory access outside the established bounds of its referent. The most prominent example are buffer overflows, where data is written beyond the limit of a buffer, potentially overwriting other values. Therefore, we can observe out-of-bounds reads and writes as part of this class.
- **Temporal Memory Errors:** On the other hand, temporal errors are those caused by the usage of pointers whose referents are not valid at the time of dereference, because they have not been initialized or they have been previously freed, for instance. In that context, we can observe Use-After-Free (UAF) errors, uses of uninitialized memory, null pointer dereferences and illegal frees (i.e., calling free to a non-allocated or an already-freed pointer).

3 Memory Error Detection

Over the years, many detection systems that use different strategies and program analysis techniques have been proposed, usually putting together diverse analysis methods [44, 42].

In the same way as user-space programs, operating system kernels such as Linux suffer from memory corruption. Detection in OS kernels entails its own challenges since their characteristics (e.g., generalized use of machine-level code, large and complex code bases) differ from userland programs. Among the proposed approaches, we can find static-analysis-based solutions [22, 29] and dynamic analysis frameworks [41].

In this section we refer to general detection methods, not specific to any domain.

3.1 Static Analysis

Given the source or binary code of a program, several approaches make use of static analysis methods to find memory errors without running the program. The most common include: pointer analysis [50], model checking [19], abstract interpretation [9] and pattern matching [29].

In general, pure static analysis implementations have the advantage of examining the whole program rather than only executed paths, but are more likely to report false positives.

3.2 Instrumentation and Runtime Monitoring

Program instrumentation is a remarkably versatile method that enables a great variety of analyses. The so-called sanitizers and many other vulnerability detection tools leverage instrumentation capabilities from e.g., compiler toolchains, to insert reference monitors into the program to then enforce a security policy at runtime [42].

To detect memory errors, a diverse range of strategies have been implemented and numerous sanitizers have been proposed [43]. To illustrate, AddressSanitizer [36] introduces red zones around memory objects to detect out-of-bounds accesses and delays the reuse of freed memory regions to discover temporal safety issues. Moreover, SoftBound+CETS [24, 25] track pointers with bounds and allocation status metadata and perform checks on dereference operations to provide full memory safety at the cost of higher overhead and compatibility issues.

Likewise, when program source code is not available, Dynamic Binary Instrumentation (DBI) frameworks like Valgrind [26] or DynamoRIO [3] provide the means to build memory corruption detection tools. Excellent examples are Memcheck [37] and Dr. Memory [4]. However, DBI usually introduces excessive performance overhead to the execution.

3.3 Fuzzing

Fuzz testing is a dynamic approach that has become one of the most used and effective techniques in recent years, while developing several new strategies and algorithms [18]. The main idea behind fuzzing resides in randomly generating inputs to test target programs, sometimes with the aim of discovering memory errors [48].

Since one of the main problems of sanitizers and other instrumentation-based detection tools is that they can only detect vulnerabilities when they are getting triggered, the combination with fuzzing turned out natural [13, 28]. Fuzzing has also been shown to be effective in finding memory corruption in embedded systems, even though it exhibits different challenges to those of commodity systems [23].

3.4 Symbolic Execution

Rather than regularly executing a target program, input values and variables are represented as symbolic values instead of concrete values in a symbolic execution. These values generate path conditions that represent the state of the program and transformations between states, opening up the possibility to analyze all possible states [1].

These techniques have been successfully used to detect memory errors [31], and combined with other techniques such as Natural Language Processing (NLP) [49].

In practice, symbolic execution approaches are infrequently adopted because of its limitations in terms of scalability due to the path explosion problem, and complex constraint solving [1].

4 Memory Corruption Exploitation

Over the last years, there has been a constant arms race between memory corruption attacks and defenses. Triggering a memory error is usually the first step to carry out several attacks. By making a pointer go out of bounds or making it become dangling and then dereferencing it, a potential attacker could modify code pointers and variables, for instance [44].

The most usual objective is to divert program execution away from the intended control flow. In that context, code corruption attacks try to modify program code in memory while code injection schemes introduce new code, in any case to make the program execute the attacker-provided code. However, they are easily prevented using current processor features. Memory errors can be further exploited to disclose memory contents [27], usually to leak secrets in order to bypass several defenses [35].

Amid existing attacks, the security community has lately paid more attention to two classes: code-reuse and data-only attacks.

4.1 Code-Reuse Attacks

Since general attack mitigations became ubiquitous and code corruption and injection were no longer viable, new attacks emerged that reuse program code already present in memory [32].

Return-into-libc is the simplest form of these kind of attacks. By redirecting the control-flow to libc functions, an attacker is able to execute sequences of arbitrary function calls chained together [12].

These attacks evolved into Return-Oriented Programming (ROP), that instead of targeting whole functions, chains the execution of short instruction sequences (i.e., gadgets) that end with a return instruction [38, 46]. While maintaining the basis of the technique, further ROP attacks have been proposed over the years to overcome emerging defenses [33], showing its effectiveness.

4.2 Data-Only Attacks

Unlike previous approaches, data-only attacks (also called non-control-data attacks) do not require hijacking the control-flow of the target program. Instead, this class of attacks focus on modifying security-sensitive data such as user input, configuration data, user identity, or decision-making data to bypass authentication checks, escalate privileges and so on [7].

Moreover, Hu et al. [15] proposed FlowStitch, a tool that successfully automates the process of generating data-oriented attacks from common memory errors by systematically joining data flows in the program. Afterwards, a follow-up work [16] introduced the notion of Data-Oriented Programming (DOP), a systematic technique to construct Turing-complete data-only exploits by finding data-oriented gadgets, in a similar fashion to ROP. Likewise, Ispoglou et al. [17] proposed Block-Oriented Programming (BOP), which uses entire basic blocks as gadgets and produces payloads that follow the valid Control-Flow Graph of the program but not its Data-Flow Graph.

5 Defense Strategies

As a consequence of the increasing number of developed attacks over the years, several countermeasures have been equally proposed. However, many of them are not deployed in practice, or a less strict version is used, since they usually incur high performance overhead.

The most basic and widely used defenses include: Address Space Layout Randomization (ASLR), the Write XOR Execute policy for memory pages, and stack canaries. ASLR randomizes the arrangement of the process memory layout so that attackers do not know the location of interesting targets. Write XOR Execute ensures that memory pages can be writable or executable but not both to thwart code injection. While stack canaries introduce values (i.e., canaries) into the stack that are checked when a function terminates to prevent control-flow hijacking. Unfortunately, they have been shown to be ineffective against more sophisticated attacks [32, 35] and consequently more advanced defenses have been developed.

5.1 Enhanced Software Diversity

Since the main problem of secret-based defenses like ASLR is that they are susceptible to information disclosure attacks [27], a lot of research effort has been put in developing leakage-resistant software diversity techniques [20]. As an example, ASLRGuard [21] uses a secure memory region to store most code locators and encryption for the rest of the code pointers with the aim of preventing its disclosure. Another approach is live re-randomization, which prevents the attacker from exploiting the obtained knowledge about the program by systematically re-randomizing the address space to invalidate current code pointers [2, 8].

5.2 Control-Flow Integrity

An interesting alternative are integrity-based defenses, being Control-Flow Integrity (CFI) [5] the most extended approach. CFI builds the Control-Flow Graph (CFG) of the program to obtain possible execution paths and then ensures that execution follows one of those paths at runtime. Nevertheless, precise CFI enforcement entails high performance overhead and, consequently, several implementations trade security for performance. Therefore we discern two broad classes of CFI: a more relaxed integrity checking called coarse-grained CFI [51, 45] and a more precise fine-grained CFI [30].

In order to improve overall performance, researchers have explored the application of hardware features, successfully enforcing CFI using Intel's processor trace [14] for example.

5.3 Data-Flow Integrity

In a similar fashion, Data-Flow Integrity (DFI) aims to mitigate non-control data attacks by preventing the program from deviating from the intended data-flow.

With the aid of static analysis (i.e., reaching definitions analysis) a Data-Flow Graph (DFG) is built that, in short, maps instructions to definition identifiers. Then, read and write instructions are instrumented to ensure at runtime that all data flows are within the DFG [6, 40]. Any deviation from the DFG terminates the execution.

5.4 Software Fault Isolation

To deal with untrusted code, Software Fault Isolation (SFI) isolates untrusted modules in a sandbox in the host's address space so that it cannot access memory outside the sandbox. As representative examples, Google's NaCl [34] loads the untrusted code into a predefined area and instruments it to confine its memory and instruction references to the sandbox, while ARMlock [52] uses the ARM hardware support for memory domains to create sandboxes that constrain every memory access inside them.

6 Related Work

In the context of this research, several systematization of knowledge works have been already proposed.

Van der Veen et al. [47] provide a historical overview and analysis of memory errors. They found that memory errors are unlikely to lose significance in the near future and that state-of-the-art detection and containment techniques fail to protect against motivated attackers.

Szekeres et al. [44] organize the knowledge about diverse protection methods by setting up a general model for memory corruption attacks to find that stronger protection policies are needed.

Shoshitaishvili et al. [39] propose angr, a unified framework for the automated identification and exploitation of memory corruption vulnerabilities at binary-level.

In relation to defense approaches, Larsen et al. [20] systematize the understanding of software diversity and highlight fundamental trade-offs between fully automated approaches.

With regard to program reasoning and vulnerability detection, Baldoni et al. [1] survey the main aspects of symbolic execution and the most notable techniques used in testing and security applications, while Klees et al. [18] perform an extensive evaluation of the most recent fuzzers and analyze their experimental methodologies.

Lastly, Song et al. [42] bring together the knowledge about sanitizers. They propose a new taxonomy of available tools and the security vulnerabilities they cover, describe their performance and compatibility properties, and emphasize different trade-offs.

7 Conclusion

As observed, memory errors have been extensively studied from different perspectives but they still remain as one of the principal difficulties to build secure systems, which suggests that further research is needed.

By using techniques like runtime monitoring or fuzzing, researchers have effectively mitigated memory corruption. While, software diversity and integrity-based defenses have been found to be capable of preventing complex attacks. However, in order to keep exploiting memory vulnerabilities, attacks have evolved from basic code-injection to more advanced code-reuse and data-only attacks.

We hope that this review provides security researchers with solid background knowledge to contribute to the field.

Acknowledgments

This work is partially supported by the Basque Government under a pre-doctoral grant given to Oscar Llorente-Vazquez.

References

1. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 1–39 (2018)
2. Bigelow, D., Hobson, T., Rudd, R., Streilein, W., Okhravi, H.: Timely rerandomization for mitigating memory disclosures. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 268–279 (2015)
3. Bruening, D., Amarasinghe, S.: Efficient, transparent, and comprehensive runtime code manipulation. Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering (2004)

4. Bruening, D., Zhao, Q.: Practical memory checking with dr. memory. In: International Symposium on Code Generation and Optimization (CGO 2011), pp. 213–223. IEEE (2011)
5. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-flow integrity: Precision, security, and performance. ACM Computing Surveys (CSUR) **50**(1), 1–33 (2017)
6. Castro, M., Costa, M., Harris, T.: Securing software by enforcing data-flow integrity. In: Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI), pp. 147–160 (2006)
7. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: 14th USENIX Security Symposium (USENIX Security 5), vol. 5 (2005)
8. Chen, X., Bos, H., Giuffrida, C.: Codearmor: Virtualizing the code space to counter disclosure attacks. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 514–529. IEEE (2017)
9. Clang static analyzer. `https://clang-analyzer.llvm.org/`
10. Cloosters, T., Rodler, M., Davi, L.: Teerex: Discovery and exploitation of memory corruption vulnerabilities in sgx enclaves. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 841–858 (2020)
11. Cwe - 2020 cwe top 25 most dangerous software weaknesses. `https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html`
12. Designer, S.: Return-to-libc attack. Bugtraq, Aug (1997)
13. Dinesh, S., Burow, N., Xu, D., Payer, M.: Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 1497–1511. IEEE (2020)
14. Ge, X., Cui, W., Jaeger, T.: Griffin: Guarding control flows using intel processor trace. In: 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, pp. 585–598. Association for Computing Machinery (2017)
15. Hu, H., Chua, Z.L., Adrian, S., Saxena, P., Liang, Z.: Automatic generation of data-oriented exploits. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 177–192 (2015)
16. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 969–986. IEEE (2016)
17. Ispoglou, K.K., AlBassam, B., Jaeger, T., Payer, M.: Block oriented programming: Automating data-only attacks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 1868–1882 (2018)
18. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 2123–2138 (2018)
19. Kroening, D., Tautschnig, M.: Cbmc–c bounded model checker. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 389–391. Springer (2014)
20. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: Sok: Automated software diversity. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 276–291. IEEE (2014)
21. Lu, K., Song, C., Lee, B., Chung, S.P., Kim, T., Lee, W.: Aslr-guard: Stopping address space leakage for code reuse attacks. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 280–291 (2015)

22. Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., Vigna, G.: DR.CHECKER: A soundy analysis for linux kernel drivers. In: 26th USENIX Security Symposium (USENIX Security 17), pp. 1007–1024 (2017)
23. Muench, M., Stijohann, J., Kargl, F., Francillon, A., Balzarotti, D.: What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In: Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS) (2018)
24. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: Softbound: Highly compatible and complete spatial memory safety for c. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 245–258 (2009)
25. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: Cets: compiler enforced temporal safety for c. In: Proceedings of the 2010 International Symposium on Memory Management, pp. 31–40 (2010)
26. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices **42**(6), 89–100 (2007)
27. Oikonomopoulos, A., Athanasopoulos, E., Bos, H., Giuffrida, C.: Poking holes in information hiding. In: 25th USENIX Security Symposium (USENIX Security 16), pp. 121–138 (2016)
28. Österlund, S., Razavi, K., Bos, H., Giuffrida, C.: Parmesan: Sanitizer-guided greybox fuzzing. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 2289–2306 (2020)
29. Padioleau, Y., Lawall, J., Hansen, R.R., Muller, G.: Documenting and automating collateral evolutions in linux device drivers. Acm sigops operating systems review **42**(4), 247–260 (2008)
30. Payer, M., Barresi, A., Gross, T.R.: Fine-grained control-flow integrity through binary hardening. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), pp. 144–164. Springer (2015)
31. Ramos, D.A., Engler, D.: Under-constrained symbolic execution: Correctness checking for real code. In: 24th USENIX Security Symposium (USENIX Security 15), pp. 49–64 (2015)
32. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Transactions on Information and System Security (TISSEC) **15**(1), 1–34 (2012)
33. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In: 2015 IEEE Symposium on Security and Privacy (SP), pp. 745–762. IEEE (2015)
34. Sehr, D., Muth, R., Biffle, C.L., Khimenko, V., Pasko, E., Yee, B., Schimpf, K., Chen, B.: Adapting software fault isolation to contemporary cpu architectures. In: 19th USENIX Security Symposium (USENIX Security 10) (2010)
35. Seibert, J., Okhravi, H., Söderström, E.: Information leaks without memory disclosures: Remote side channel attacks on diversified code. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 54–65 (2014)
36. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp. 309–318 (2012)
37. Seward, J., Nethercote, N.: Using valgrind to detect undefined value errors with bit-precision. In: USENIX Annual Technical Conference (USENIX ATC 2005), pp. 17–30 (2005)

38. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM conference on Computer and Communications Security (CCS), pp. 552–561 (2007)
39. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 138–157. IEEE (2016)
40. Song, C., Lee, B., Lu, K., Harris, W., Kim, T., Lee, W.: Enforcing kernel security invariants with data flow integrity. In: Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS) (2016)
41. Song, D., Hetzelt, F., Das, D., Spensky, C., Na, Y., Volckaert, S., Vigna, G., Kruegel, C., Seifert, J.P., Franz, M.: Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In: Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS) (2019)
42. Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., Franz, M.: Sok: sanitizing for security. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1275–1295. IEEE (2019)
43. Stepanov, E., Serebryany, K.: Memorysanitizer: fast detector of uninitialized memory use in c++. In: 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 46–55. IEEE (2015)
44. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 48–62. IEEE (2013)
45. Van Der Veen, V., Göktas, E., Contag, M., Pawoloski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., Giuffrida, C.: A tough call: Mitigating advanced code-reuse attacks at the binary level. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 934–953. IEEE (2016)
46. van der Veen, V., Andriesse, D., Stamatogiannakis, M., Chen, X., Bos, H., Giuffrdia, C.: The dynamics of innocent flesh on the bone: Code reuse ten years later. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 1675–1689 (2017)
47. Van der Veen, V., Cavallaro, L., Bos, H., et al.: Memory errors: The past, the present, and the future. In: International Conference on Recent Advances in Intrusion Detection (RAID), pp. 86–106. Springer (2012)
48. Wang, H., Xie, X., Li, Y., Wen, C., Li, Y., Liu, Y., Qin, S., Chen, H., Sui, Y.: Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 999–1010. IEEE (2020)
49. Wang, J., Ma, S., Zhang, Y., Li, J., Ma, Z., Mai, L., Chen, T., Gu, D.: Nlp-eye: Detecting memory corruptions via semantic-aware memory operation function identification. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), pp. 309–321 (2019)
50. Yan, H., Sui, Y., Chen, S., Xue, J.: Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 327–337. IEEE (2018)
51. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 559–573. IEEE (2013)
52. Zhou, Y., Wang, X., Chen, Y., Wang, Z.: Armlock: Hardware-based fault isolation for arm. In: Proceedings of the 2014 ACM SIGSAC conference on Computer and Communications Security (CCS), pp. 558–569 (2014)