# ROPBench: A Modular Testbed for Return-Oriented Programming

Eric Zou, *University of Pennsylvania*

***Abstract*—Return-Oriented Programming (ROP) describes a class of attacks used to bypass existing system defenses using pre-existing "safe" code snippets linked by return addresses on a compromised program stack. Projects using low-level languages like C++ give developers access to sensitive memory internals, which can create exploitable vulnerabilities if unsafe code is present. Given the broad nature of the attack and its potential consequences, such as arbitrary code execution, many defenses have been proposed and tested. Solutions can involve either compilation-based fixes, runtime-based fixes, or a mixture of both. However, a more comprehensive evaluation of existing solutions is desirable to understand security coverage and limitations, especially with the introduction of novel attacks and security bypasses. In this work, we introduce a novel testbed framework for testing ROP defenses across the stack. Our approach is a benchmark-focused approach for the x86_64 architecture that utilizes a standardized but extensible set of programs to characterize solutions in terms of performance, along with modern open-source ROP gadget and chain discovery tools to run test cases. We apply this benchmark to a variety of existing tools aimed to defend against control-flow integrity attack to test our solutions validity by verifying it against past results.***

## I. Background

Return-oriented programming (ROP) [1,2,3] is a sophisticated exploit technique that bypasses traditional memory protection mechanisms like W⊕X [4]. The primary method of attack involves compromising the integrity of the stack via a buffer overflow, then injecting a payload of return addresses that link to sections of executable memory containing useful functionality, often referred to as gadgets. By chaining these gadgets, an attacker can execute arbitrary code in a Turing-complete manner [2]. While various defenses have been developed on top of W⊕X, such as address space layout randomization (ASLR) [5], stack-protection mechanisms like stack canaries [6], and shadow stacks [7], these solutions are not foolproof. Vulnerabilities remain due to limitations in their implementation, making them susceptible to ROP exploits [8,9,10].

Other defenses, including memory safety mechanisms like Softbound+CETS [11,12,13] and control-flow integrity (CFI) [14], are more comprehensive but often come with significant performance penalties that are unacceptable in high-performance environments. Even when CFI is used, some implementations remain vulnerable to certain types of attacks that they are theoretically designed to mitigate [15].

## II. Existing Defenses Against ROP

Many detection schemes for ROP attacks have been proposed, including DROP [16], kBouncer [17], and ROPecker [18], which rely on heuristics to identify suspicious code patterns indicative of ROP chains. IDROP [19] extends this by leveraging machine learning techniques, such as LSTM neural networks, to predict potential ROP chains. Shadow stacks, as seen in TRUSS [20] and ROPDefender [21], also help detect discrepancies by maintaining a separate virtual stack to verify return addresses. Additionally, compile-time solutions such as G-Free [22] and CCFIR [23] aim to prevent ROP by eliminating gadgets or enforcing stricter control flow. However, most of these solutions are difficult to compare in a unified way due to differences in implementation, effectiveness, and trade-offs.

## III. Motivations

Given the challenges in comparing existing ROP defenses, we propose the creation of a robust, extensible benchmarking suite designed to evaluate ROP mitigations on the x86_64 architecture. The envisioned threat is an attacker who can provide crafted input to potentially vulnerable programs, such as OpenSSH, gzip, and curl, exploiting vulnerabilities in these widely-used, open-source binaries. By leveraging known ROP attacks, our benchmark will test the resilience of defenses against these exploits. This model will be expanded upon in the next section.

The benchmark includes all necessary source code, build tools, and runtime support to allow users to easily integrate their defenses or modify the setup to suit their specific requirements. It also leverage open-source ROP gadget discovery tools like ROPGadget [24] and Q [25], enabling users to create custom attack chains when none exist in the literature. A comprehensive literature review was be conducted to identify edge cases and attack vectors that may challenge existing defenses, ensuring that the benchmark reflects the latest advancements in ROP exploit techniques.

One of the key goals of this benchmark is comparability. By testing solutions in a controlled environment where operating system details and security mechanisms (such as ASLR and stack canaries) are held constant, researchers will be able to directly compare the effectiveness of different defenses. Standard methods for bypassing each defense will be included, ensuring that the ROP attacker code has access to all necessary tools to exploit vulnerabilities as needed. We contrast this to a manual approach that was taken in works like [26].

The extensibility of this benchmark is central to its design. As new vulnerabilities and attack techniques are discovered, the benchmark can be updated to reflect these developments. Additionally, the modular nature of the setup allows researchers to add new test cases, defenses, or attack methods with minimal effort, ensuring the benchmark remains relevant and applicable over time. This flexibility ensures that the benchmark will continue to be useful as the field evolves, supporting ongoing research into ROP defenses and enabling faster iteration on new ideas.

## IV. Threat Model

The threat model for our benchmark involves simulating a scenario where an attacker exploits vulnerabilities in C programs, commonly found in applications that accept user-provided input. These vulnerabilities typically stem from unsafe memory handling, such as buffer overflows or improper bounds checking, which allow an attacker to manipulate the stack or heap in ways that lead to arbitrary code execution. The key characteristics of the attack vector include vulnerable programs, C-based applications, which are known to process user input in potentially unsafe ways. These programs often operate with elevated privileges, making them attractive targets for attackers.

As is common in the literature, we model the attacker's primary tool as crafted user input. This input can take the form of data sent through standard input, network connections, or file uploads. When the program processes this input without proper validation, the attacker can exploit buffer overflows or other memory corruption bugs to overwrite key data structures like return addresses or function pointers, redirecting the program's control flow. In this model, the attacker is assumed to have knowledge of the program's structure, either through reverse engineering or by exploiting publicly disclosed vulnerabilities. The attacker may also know or guess the locations of certain "gadgets" in the program's executable memory, which are short sequences of instructions that, when chained together, can perform arbitrary actions. The attacker can manipulate the program's stack or heap to redirect the execution to these gadgets, bypassing traditional security mechanisms. The machines at risk are those running applications written in low-level languages (like C) that rely on the stack and heap for memory management. These machines could be running a wide range of operating systems, including Linux, which commonly implements security mechanisms like ASLR (Address Space Layout Randomization) [5] and stack canaries [6] to mitigate ROP attacks. However, these protections are not foolproof [8, 9, 10], and vulnerable programs are still at risk, especially when misconfigured or outdated.

If an attacker successfully exploits a vulnerability using ROP, they can gain arbitrary code execution within the context of the vulnerable program. Depending on the privileges of the compromised program, this could lead to severe consequences, such as privilege escalation, remote code execution, or exfiltration of sensitive data. The attacker may also maintain persistence on the system, using ROP chains to hide their presence from traditional security measures like antivirus software.

## IV. Testbed Design

Broadly, the testbed does not require a tested ROP defense to sit at a particular abstraction layer. Since the tester only evaluates if an ROP attack can be successfully executed on a system, this allows for characterization of the defense of the system as a whole, enabling testing of hybrid designs. Additionally, simple environment configuration supported by YAML and Python offers incredible flexibility and isolation, since users can selectively enable and disable defenses during any particular run of the suite.

Our testbed takes inspiration from the dimensional view of the Runtime Intrusion Prevention Evaluator (RIPE) [27] on a related problem of buffer overflow attacks. This testbench characterizes intrusions based on a few dimensions that aim to be orthogonal. While not all of these dimensions are necessarily applicable to ROP attacks, we maintain this style and establish this framework early in the lifecycle of this testbed to prepare it for future extensions that will encompass different combinations of values in each of these dimensions. The formulation of testing as iterating through different aspects of each dimension allows for systematic testing and provides the groundwork for automatic generation of test cases, which can greatly expand the coverage of tests.

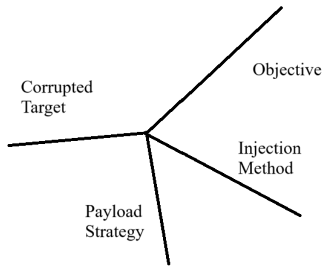

Figure 1. Testbed Dimensions

The injection vector dimension refers to the type of vulnerability that allows an attacker to inject malicious data into the target program's memory. These vectors form the primary entry point for ROP-based attacks. The importance of identifying the injection vector lies in the fact that different types of vulnerabilities exploit different memory regions and may require distinct mitigation strategies.

Stack overflow is one of the most common injection vectors for ROP attacks, where an attacker writes more data to a buffer on the stack than it can hold, leading to overwriting the return address or other control structures. This type of vulnerability is crucial in the context of ROP attacks because it directly facilitates control flow manipulation. Heap overflow is also another potential injection vector. In this case, the attacker overflows a buffer located on the heap, which may corrupt function pointers or other critical data structures that can redirect execution. Then, with a stack pivot, the user can redirect control to the heap to execute the ROP chain, potentially evading some stack-centric detection mechanisms. While stack overflows are more widely known in ROP attacks, heap overflows represent another important vector, particularly in scenarios involving dynamic memory allocation. Other vectors exist, use-after-free, double-free, format string vulnerabilities, function pointer overwrites, and global offset table and procedure linkage table overwrites, are currently planned, but do not have coverage in the current version of the testbed. These are valuable to consider for a more advanced testbed where heap-based attacks, memory corruption related to dangling pointers, or manipulation of function pointers (e.g., in the GOT or PLT) could be part of an attacker's arsenal. However, for the scope of this initial testbed and its current trajectory, heap and stack overflow are prioritized for their direct applicability to common ROP attack techniques.

The corrupted target dimension refers to the specific memory structures or data that the attacker aims to corrupt in order to execute a ROP-based exploit. The type of corrupted target determines how the attacker can manipulate program control flow. The return address is often the primary target in classic ROP attacks. By overwriting the return address, an attacker can redirect the execution flow to their malicious payload (typically a chain of ROP gadgets). A jmp_buf structure holds information for performing longjmp operations, which can also be targeted by an attacker to gain control over the program's execution flow in a process known as Jump-Oriented Programming (JOP) [28]. It's particularly relevant in programs that use non-local jumps for error handling or other purposes. The other potential targets—function_pointer, vtable, sigcontext, and got_entry—are currently commented out but represent important targets for future tests. A function pointer or vtable may be modified to point to malicious functions, while sigcontext and got_entry can be used to exploit vulnerabilities in signal handling or function address resolution. However, the current priorities focus on the return address as this is most commonly manipulated in ROP attacks.

The payload strategies dimension refers to the various methods attackers use to construct their ROP chains, enabling the manipulation of control flow and achieving the attack objective. Different payload strategies target specific weaknesses or leverage different techniques within the system's memory. A ret2win is a classic ROP technique where the attacker's goal is to return to a "win" function, usually one that grants access to a shell or other privileged functionality. This is a very common goal in exploiting buffer overflows in C programs, and it represents the lowest level of difficulty of attack present in the testbed. A ret2libc attack [1] involves returning to functions in the libc library, such as system calls, to execute arbitrary code. This is another popular ROP technique because it avoids the need for injecting custom shellcode, instead relying on already-present functions in the system's libraries. This particular attack is concerning since the vulnerable program does not need to call these functions for them to become available to the attacker.

Other strategies and longer ROP chains that achieve arbitrary arithmetic, output, input, and memory operations, such that these programs approach or achieve Turing-completeness [2], are currently not included in the testbed. These strategies represent more advanced or specialized techniques. For instance, ret2dlresolve exploits dynamic linking, while ret2vtable is relevant for exploiting virtual function tables in C++ programs. These techniques will be incorporated as the testbed evolves, but for now, the focus remains on the most common approaches and expanding them comprehensively to create attacks that utilize the other dimensions.

Finally, our testbench considers an objectives dimension that defines the specific goals that the attacker aims to achieve by executing their ROP chain. Each objective represents a different level of impact, ranging from gaining control over the system to performing unauthorized computations. One of the most common objectives of ROP attacks is to spawn a shell with elevated privileges, allowing the attacker to control the system. A win function objective aims to redirect execution to a specific function in the program (e.g., a "win" function) to achieve arbitrary goals set by the attacker, such as altering program behavior or achieving a specific computation. A read flag objective may be to read a file or a flag to prove the attack's success. The attacker may also seek to modify specific program variables or data structures, leading to changes in program behavior or a potential data leak. An unauthorized computation objective refers to running computations that the attacker is not authorized to execute, such as running privileged code or bypassing restrictions on program functionality. Each of these objectives is essential for testing a range of ROP exploits, allowing for the evaluation of different types of attacks that may target specific goals. The prioritization of spawn_shell and win_function reflects the more traditional and high-impact attack objectives seen in real-world exploits.

We believe these dimensions offer a comprehensive view of possible ROP-style attacks, but also leave room for future additions to the testbed and the dimensions currently considered. With this in mind, the core of the testbench is a Python program that delivers payloads to vulnerable C programs with gadgets added to their binaries. The user is able to easily specify how programs should be compiled to take advantage of hardware and compiler solutions. Additionally, the Python program provides a variety of utilities for writing new tests in a uniform format so that they can be executed easily by the central running script. Outputs of data are provided in CSV and YAML formats for human readability and easy processing with common data analysis tools. The Python language features allow for the mixing of manual and automatically generated tests, leaving room for counterexample development in the future, targeting specific defenses.

## V. Experimental Results

We evaluate our current testbench, consisting of four tests, on four different security environment configurations: Linux standard (a nonexecutable or NX stack and W⊕X without ASLR and stack canaries, standard defenses with ASLR, standard defenses with stack canaries, and standard defenses with ASLR and stack canaries. Our results are as follows:

| **Test Name** | **Result** |
|---|---|
| **Standard Defense** | |
| Ret2Win | Attack Success |
| Ret2Libc | Attack Success |
| Procedurally Generated Ret2Win | Attack Success |
| Format String Canary Leak Ret2Libc | Attack Success |
| **Stack Canary** | |
| Ret2Win | Attack Fail |
| Ret2Libc | Attack Fail |
| Procedurally Generated Ret2Win | Attack Fail |
| Format String Canary Leak Ret2Libc | Attack Success |
| **ASLR** | |
| Ret2Win | Attack Fail |
| Ret2Libc | Attack Fail |
| Procedurally Generated Ret2Win | Attack Fail |
| Format String Canary Leak Ret2Libc | Attack Fail |
| **ASLR and Stack Canary** | |
| Ret2Win | Attack Fail |
| Ret2Libc | Attack Fail |
| Procedurally Generated Ret2Win | Attack Fail |
| Format String Canary Leak Ret2Libc | Attack Fail |

Figure I:

## VI. Discussion

The results confirm existing knowledge in the literature, providing an initial validation of the testbench. We expect that ROP, an attack specifically crafted to dodge defenses like NX bits and W⊕X, should have no trouble achieving attacker goals on systems without more defenses. The results confirm that ASLR proves to be one of the most effective defenses against ROP due to the nature of the attack. Because ROP relies on precise memory locations to target redirection of control flow, randomization of these locations is often an insurmountable obstacle if there are no leaks by the program indicating where certain critical functions are located. We note that there is value in including counterexample attacks on certain defenses, as shown in the example of the stack canary leak. In real programs, it is often the case that memory leaks and bugs are present that can leak critical information about the execution or location of items in memory, which can then be targets for an attack. In this case, the vulnerable printf statement leaks the stack canary to the adversary, who can then include this information in their payload. These kinds of attacks can not only help understand the limits of these kinds of solutions from a research perspective but also allow future defense developers to account for these scenarios and edge cases early on in development and avoid relying on unchecked assumptions.

## VII. Limitations And Opportunities

While the current testbed provides a foundational structure for evaluating Return Oriented Programming (ROP) and memory corruption attacks, it has several limitations that must be addressed in order to broaden its applicability and improve its effectiveness. These limitations stem from a combination of narrow focus, lack of comprehensive validation, and the complexity inherent in ROP attack modeling. Below, we address the current issues in the testbed, propose solutions, and outline a roadmap for future work.

Currently, the testbed includes only **four** working tests due to development timing constraints, which are primarily limited to short ROP chains and ret2libc/ret2win strategies. This narrow scope restricts the variety of attack types and scenarios that can be explored, leaving out more complex techniques such as JOP, which are critical in evaluating more sophisticated attack vectors. In the future, the testbed should incorporate a wider array of attack strategies, extending beyond basic ROP chains and ret2libc approaches. Adding tests for techniques such as JOP, ROP arithmetic, and ROP input output would provide a more comprehensive testing framework. Additionally, more complex ROP chains of varying lengths should be supported, including both basic and advanced chaining techniques. This could be achieved by implementing a modular approach to ROP chain construction, enabling tests to easily scale and adapt to more complex payloads. As of now, the testbed has been validated against only two defense mechanisms, ASLR and stack canaries. While these are important security features, the lack of validation against other widely recognized ROP mitigations, such as Control Flow Integrity [14, 23] and G-Free [22], limits the testbed's reliability in evaluating the overall effectiveness of modern defenses. Additionally, this characterization would allow for a more complete view of the state-of-the-art in terms of defense against ROP. Understanding where solutions fall short or succeed greatly can help in identifying grave errors before they can be exploited by bad actors, reframing priorities, and examining possible opportunities for hybrid technologies.

Moving toward a more holistic view of security features, the testbed only assesses the correctness of attacks, ensuring that the ROP chains and exploit techniques successfully achieve their intended objectives (or are defended against). However, it does not evaluate performance in terms of exploit reliability, efficiency, or the cost of bypassing defenses. This oversight limits the ability of the testbed to provide insight into real-world exploit practicality, where performance and reliability are crucial factors. Future work should introduce performance-related metrics into the testbed. This could include the time required for exploit construction and execution, the number of gadgets needed for a given attack, and the success rate under different configurations. Metrics such as exploit time-to-success, gadget count, and resilience under repeated executions would provide a clearer picture of the practical feasibility of the attacks. Additionally, the testbed could support stress testing where the robustness of exploits under various system loads and defense mechanisms is evaluated.

The current testbed lacks a performance benchmarking component, limiting its ability to evaluate defensive mechanisms beyond correctness. In practical systems, particularly those with tight latency or memory constraints, solutions that impose significant runtime or memory overhead are often unacceptable, even if they offer strong security guarantees. Without quantifying the execution time, memory usage, or binary size changes introduced by each defense, the testbed cannot inform tradeoff-aware decisions or guide adoption in real-world environments. Many consumers and system designers prioritize lightweight defenses, and the absence of such metrics makes it difficult to compare or recommend solutions in contexts like embedded systems, cloud workloads, or interactive applications. Furthermore, neglecting performance obscures key failure modes, such as defenses that are theoretically sound but prohibitively expensive in practice.

To address this, future versions of the testbed should integrate a systematic performance benchmarking framework. This module would collect quantitative data on solution speed, memory overhead, and cache behavior under different defense configurations. It would enable empirical comparisons of security-to-cost ratios across mechanisms, allowing researchers and practitioners to evaluate solutions along multiple axes. Ideally, the testbed would support automated profiling, reporting not only protection success but also cost curves across workloads and platforms. This infrastructure could eventually support multi-objective optimization, identifying Pareto-optimal defenses under constrained budgets or specific deployment targets. Including performance benchmarking would extend the testbed's relevance beyond academia, positioning it as a tool for real-world security engineering and practical system design.

The current testbed treats the dimensions of injection vectors, corrupted targets, payload strategies, and objectives as orthogonal, but in practice, these dimensions exhibit dependencies and interactions that may affect the outcome of the tests. For example, certain payload strategies may only be effective when specific injection vectors are present, or certain corrupted targets may only be vulnerable under particular conditions (e.g., when ASLR is disabled). These dependencies create complexities in the testbed that could require future refactoring for more accurate modeling. To address this issue, a more sophisticated modeling approach is necessary. One potential solution is to introduce a dependency map or constraint system that models the interactions between dimensions. This system would allow the testbed to track and enforce the conditions under which certain payload strategies or injection vectors are applicable, ensuring that the tests remain realistic and aligned with the interactions seen in real-world attacks. Furthermore, the testbed should be refactored to allow dynamic adjustments based on these dependencies, ensuring that the correct combinations of configurations are tested without violating logical constraints.

Although the testbed simulates ROP attacks in a controlled environment, it currently lacks sufficient realism to fully replicate the conditions of real-world exploitation. For instance, the absence of network effects, multi-user scenarios, or real operating system interactions means the testbed does not fully capture the complexities of exploitation in production systems. This limitation could impact the accuracy of the testbed in modeling sophisticated ROP attacks in more dynamic and complex environments. To improve the realism of the testbed, future work should include the simulation of more realistic system environments, including aspects such as network-based attacks, multi-threaded exploits, and interactions with real OS components like the kernel, memory allocators, and signal handlers. Additionally, introducing the ability to simulate attacks in multi-process or multi-user scenarios would better reflect the conditions under which ROP attacks may be executed in production systems. Integrating with containerization tools could provide the isolation and control needed to simulate these environments.

At present, the testbed operates mostly independently from other established security benchmarking tools besides pwntools [] and those specifically designed for ROP, without integration into existing security testing frameworks or tools, such as fuzzers or static analyzers. This isolation limits the scope of the testbed in terms of its ability to benefit from existing, widely adopted security testing pipelines, which may include automated vulnerability discovery, static analysis, or dynamic instrumentation. Future work should integrate the testbed with existing security testing tools for fuzz testing or for dynamic analysis. This integration would allow the testbed to benefit from real-time analysis, automated vulnerability discovery, and enhanced testing coverage. Such integration would also make the testbed more useful in a continuous security testing pipeline, where exploits can be tested alongside other security checks in a highly automated fashion.

The testbench also has lessened coverage due to its current dependency on Linux, x86_64, and higher versions of Python 3 and gcc, all of which could affect the results present in the benchmark. With pwntools, it is possible to write code that accounts for different architectures and calling conventions. This is another future goal of the dataset so that it can be generalized to more test systems and provide more extensive coverage for solutions that are not entirely bound in hardware.

## VIII. Conclusion

In this work, we present a foundational prototype for a comprehensive testbed to evaluate ROP defenses by applying a comprehensive, modern, and mutable set of test cases. In the future, we hope security researchers and hardware designers will be able to use this benchmark to test their solutions for validity and efficiency, improving development times and encouraging standardization for comparison with other security proposals. We believe that this will allow for faster iteration on new ideas and help interested individuals identify gaps in existing defenses, leading to a more unified effort to mitigate the security risk posed by ROP attacks.

## IX. Attributions and Acknowledgements

The author would like to thank Professor Andre DeHon for his advice in shaping the parameterization system of this testbed and for providing resources and recommendations to help guide this process. These were invaluable in the final design of the initial testbench. The author would also like to express their appreciation for Professor DeHon for covering this topic in the class.

## X. Availability And Dependencies

The source code for the testbed is present at https://github.com/ezou626/ropbench. The systems that can be tested with the currently completed testcases are only x86_64 at the moment. The testbench currently runs with Python 3.12.3 and requires ROPgadget, pwnlib, and PyYAML. The C files are compiled using GCC version 13.3, and most tests occurred on a Ubuntu 22.04 LTS WSL2 instance running on Microsoft Windows 11. The testbench depends on a version of Linux, C, Python, and the Python packages provided being available in order to run properly.

## References

[1] 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, in CCS ’07. New York, NY, USA: Association for Computing Machinery, Oct. 2007, pp. 552–561. doi: 10.1145/1315245.1315313.

[2] 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, in CCS ’08. New York, NY, USA: Association for Computing Machinery, Oct. 2008, pp. 27–38. doi: 10.1145/1455770.1455776.

[3] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” in Proceedings of the 17th ACM conference on Computer and communications security, in CCS ’10. New York, NY, USA: Association for Computing Machinery, Oct. 2010, pp. 559–572. doi: 10.1145/1866307.1866370.

[4] Accessed: Feb. 21, 2025. [Online]. Available: https://pax.grsecurity.net/docs/mprotect.txt

[5] Accessed: Feb. 21, 2025. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[6] C. Cowan et al., “{StackGuard}: Automatic Adaptive Detection and Prevention of {Buffer-Overflow} Attacks,” presented at the 7th USENIX Security Symposium (USENIX Security 98), 1998. Accessed: Feb. 21, 2025. [Online]. Available: https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention

[7] L. Delshadtehrani, S. Eldridge, S. Canakci, M. Egele, and A. Joshi, “Nile: A Programmable Monitoring Coprocessor,” IEEE Computer Architecture Letters, vol. 17, no. 1, pp. 92–95, Jan. 2018, doi: 10.1109/LCA.2017.2784416.

[8] D. Jang, “Badaslr: Exceptional cases of ASLR aiding exploitation,” Computers & Security, vol. 112, p. 102510, Jan. 2022, doi: 10.1016/j.cose.2021.102510.

[9] B. Bierbaumer, J. Kirsch, T. Kittel, A. Francillon, and A. Zarras, “Smashing the Stack Protector for Fun and Profit,” Proceedings of the 33rd International Conference on ICT Systems Security and Privacy Protection (IFIP SEC), 2018, doi: 10.1007/978-3-319-99828-2_21.

[10] B. Lan, Y. Li, H. Sun, C. Su, Y. Liu, and Q. Zeng, “Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses,” in 2015 IEEE Trustcom/BigDataSE/ISPA, Aug. 2015, pp. 190–197. doi: 10.1109/Trustcom.2015.374.

[11] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, “SoftBound: highly compatible and complete spatial memory safety for c,” SIGPLAN Not., vol. 44, no. 6, pp. 245–258, Jun. 2009, doi: 10.1145/1543135.1542504.

[12] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, “CETS: compiler enforced temporal safety for C,” SIGPLAN Not., vol. 45, no. 8, pp. 31–40, Jun. 2010, doi: 10.1145/1837855.1806657.

[13] B. Orthen, O. Braunsdorf, P. Zieris, and J. Horsch, “SoftBound+CETS Revisited: More Than a Decade Later,” in Proceedings of the 17th European Workshop on Systems Security, in EuroSec ’24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 22–28. doi: 10.1145/3642974.3652285.

[14] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” ACM Trans. Inf. Syst. Secur., vol. 13, no. 1, p. 4:1-4:40, Nov. 2009, doi: 10.1145/1609956.1609960.

[15] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, “Out of Control: Overcoming Control-Flow Integrity,” in 2014 IEEE Symposium on Security and Privacy, May 2014, pp. 575–589. doi: 10.1109/SP.2014.43.

[16] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, “DROP: Detecting Return-Oriented Programming Malicious Code,” in Proceedings of the 5th International Conference on Information Systems Security, in ICISS ’09. Berlin, Heidelberg: Springer-Verlag, Nov. 2009, pp. 163–177. doi: 10.1007/978-3-642-10772-6_13.

[17] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Transparent {ROP} Exploit Mitigation Using Indirect Branch Tracing,” presented at the 22nd USENIX Security Symposium (USENIX Security 13), 2013, pp. 447–462. Accessed: Feb. 21, 2025. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/pappas

[18] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, “ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks,” in 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014, The Internet Society, 2014. Accessed: Feb. 21, 2025. [Online]. Available: https://www.ndss-symposium.org/ndss2014/ropecker-generic-and-practical-approach-defending-against-rop-attacks

[19] J. Li, W. Niu, R. Yan, Z. Duan, B. Li, and X. Zhang, “IDROP: Intelligently detecting Return-Oriented Programming using real-time execution flow and LSTM,” in 2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Dec. 2022, pp. 167–174. doi: 10.1109/TrustCom56396.2022.00033.

[20] S. Sinnadurai, Q. Zhao, and W. Wong, “Transparent Runtime Shadow Stack : Protection against malicious return address modifications,” 2006. Accessed: Feb. 21, 2025. [Online]. Available: https://www.semanticscholar.org/paper/Transparent-Runtime-Shadow-Stack-%3A-Protection-Sinnadurai-Zhao/36f05c011a7fdb74b0380e41ceada8632ba35f24

[21] 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, in ASIACCS ’11. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 40–51. doi: 10.1145/1966913.1966920.

[22] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, “G-Free: defeating return-oriented programming through gadget-less binaries,” in Proceedings of the 26th Annual Computer Security Applications Conference, in ACSAC ’10. New York, NY, USA: Association for Computing Machinery, Dec. 2010, pp. 49–58. doi: 10.1145/1920261.1920269.

[23] C. Zhang et al., “Practical Control Flow Integrity and Randomization for Binary Executables,” in 2013 IEEE Symposium on Security and Privacy, May 2013, pp. 559–573. doi: 10.1109/SP.2013.44.

[24] J. Salwan, JonathanSalwan/ROPgadget. (Feb. 21, 2025). Python. Accessed: Feb. 21, 2025. [Online]. Available: https://github.com/JonathanSalwan/ROPgadget

[25] E. J. Schwartz, T. Avgerinos, and D. Brumley, “Q: exploit hardening made easy,” in Proceedings of the 20th USENIX conference on Security, in SEC’11. USA: USENIX Association, Aug. 2011, p. 25.

[26] R. Skowyra, K. Casteel, H. Okhravi, N. Zeldovich, and W. Streilein, “Systematic Analysis of Defenses against Return-Oriented Programming,” in Research in Attacks, Intrusions, and Defenses, S. J. Stolfo, A. Stavrou, and C. V. Wright, Eds., Berlin, Heidelberg: Springer, 2013, pp. 82–102. doi: 10.1007/978-3-642-41284-4_5.

[27] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, “RIPE: runtime intrusion prevention evaluator,” in *Proceedings of the 27th Annual Computer Security Applications Conference*, in ACSAC ’11. New York, NY, USA: Association for Computing Machinery, Dec. 2011, pp. 41–50. doi: 10.1145/2076732.2076739.

[28] 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, in ASIACCS ’11. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 30–40. doi: 10.1145/1966913.1966919.

[29] “pwntools — pwntools 4.14.1 documentation.” Accessed: Apr. 30, 2025. [Online]. Available: https://docs.pwntools.com/en/stable/