Analysis of Defense and Detection Methods for Return Oriented Programming

Colby Saxton, Savita Medlang, John Shi, Kian Alikhani, David Navia, Kevin Greisser

Introduction

Return oriented programming is a software vulnerability that has existed for decades. Despite a large amount of literature surrounding the attack, there are still many methods to use ROP for malicious purposes. When trying to defend against a return oriented programming attack, there are several options to choose from. When choosing a user needs to be well informed, because many of the most popular types of defense come with large downsides. In this report we will be examining the existing defenses that can be used to mitigate ROP attacks, as well as the positive and negative aspects of two of the most popular types of ROP detection methods: compile time and dynamic.

Background

Return oriented programming presents a unique threat by being able to perform attacks using borrowed code from the functions that already exist in the program. This means no code needs to be added to perform the attack, and as such forms of detection or prevention that find or block attacks that need to insert code to function don't work. For example W xor X restrictions which prevents code that is writable to also be executable does not work on return oriented programming attacks because because the code is not written by the attacker[1].

Performing a return oriented programming attack involves tampering with the stack to override the destination of the return statement of a vulnerable function. After doing that the attacker strings gadgets together that when linked accomplish the desired function.

Gadgets are short instructions that end in an indirect jump. These gadgets can be taken from the main code or linked libraries, giving attackers many options to choose from when forming gadgets [1]. These indirect jumps are most commonly return statements, but jump or call statements can be used as well. However, some adjustments must be made if returns aren't used, becauses jumps and calls appear much less frequently.

Most programs of meaningful length can create a turing complete set of gadgets from short statements that end in returns. If a turing complete set can be achieved then the attacker can perform arbitrary functions from the gadgets that were found in the program. Although return oriented programming is still a prominent style of attack today there are many modern ways of defending against it.

Overview of Defences Categories

Survey of return-oriented programming defense mechanisms[2] provides an overview of many of the modern defenses against ROP attacks. It suggests that the methods of defense can be split into 3 main categories randomization, compile time and dynamic[2].

Randomization

Randomization can be broken into two categories ASLR and Instruction randomization. ASLR focuses on randomizing the base address space so attackers don't know which addresses to link when trying to string together their gadgets. The problem with this method is that it only randomizes base addresses, meaning if any base address is found then the rest can be found from that. Instruction randomization randomizes the singular blocks of instructions, which has a similar effect as ASLR. While these methods don't have much overhead they require access to the source code of the program that it's trying to protect and programs that are position-dependent are incompatible with randomization[2].

Compile time detection

While the specific application of Compile time defenses can vary, as the name suggests they all involve provide checks on the code during compile time. Some try to eliminate potential gadgets while others monitor the control flow integrity of the code, and modify the layout when necessary. Similarly to randomization one of it's big downsides is that it also requires source code, among other downsides[2]. One popular compile time defence called G-free will be discussed more in a later section.

Dynamic detection

A form of defense which runs alongside the main program and monitors its control flow dynamically. There are multiple ways in which it monitors the control flow, including keeping shadow stacks to cross check the source and destinations of indirect jumps. The benefit of dynamic checks is that they don't require access to the source code of the program they're

monitoring. The major downside is that the require much more overhead to run[2]. ROPDefender is a popular example of a dynamic defense against ROP attacks that will be discussed in a later section.

This shows that while there are methods of protecting against return oriented programming attacks they all have trade offs. No one system is easy to implement, not performance intensive and protects against all forms of ROP attack meaning the user needs to be well informed when making a decision on ROP defense or else they could be putting their program at risk.

ROP Variations

In addition to the traditional ROP attack type, there are numerous variations that have emerged. While all of these attack variants operate using similar principles as ROP attacks, there are some key differences that separate these attacks, rendering some of the defense techniques useless. An example of this is the Jump Oriented Programming (JOP) attack, which uses pop and jump instructions to modify the control flow which eliminates the reliance on the stack and return instructions. Other variations of ROP attacks utilitilize semantic knowledge of the program's virtual C/C++ functions to execute vulnerabilities. Examples of these are counterfeit object-oriented programming and return into libc/n attacks, which both leverage the C libraries built into an application's executable memory in order to perform more sophisticated ROP attacks. While these attacks can be defended against, they bypass many of the defenses that prevent traditional ROP attacks.

ROP Defenses

SAT Solver

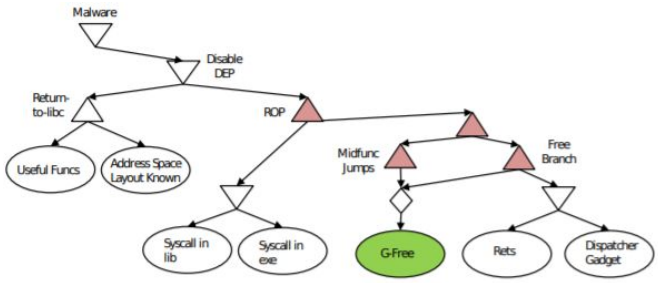
SAT Solver(Satisfiability Solver) is a method of ROP defense that does this by finding any potential loopholes that malware may be able to exploit in code segments. Overall, SAT is an NP-Complete algorithm, however through heuristics and optimizations, instances of SAT can be solved of up to millions of variables and clauses[7]. How this works is creating sets of statements about each potential variables and clauses. The contents of the sets are boolean expressions that are the following: First, dependency statements that establishes if: $a_i \varepsilon A\{\phi\}$ where ϕ is a compound formula consisting of dependencies, defense points and scenario constraints within set, A and a_i is a literal that represents the deployment of a defense mechanism, Such that:

$a_i \rightarrow \chi$ *where* a_i *is a literal and* χ *is a sub* $-$ *formula that may also be a dependancy*. [7]

This statement is stating that variable, a_i is dependant on sub-formula, χ, meaning that this specific defense mechanism is dependant on some other sub-formula, χ. The second is defense point statements that establishes: $a_i \wedge \neg abroken_i \rightarrow \neg a_j$ where a_i is some literal in the set, A which is the deployment of a specific defense, $abroken_i$ represents whether or not defense a_i is broken or not[7]. This statement signifies how if a_i is true, and it is not broken, than a_i protects against attacks that are reliant on a_j. The third type of statement within set, A is a scenario constraint which may be either $a_i = \top$ *or* $a_i = \bot$ which reveals an inherent fact about defense mechanism, a_i [7]. A SAT heuristic on the set of all these various equations allows a system to understand any state changes within its systems which may reveal the presence of a ROP attack.

G-Free

G-Free targets the key capabilities necessary for ROP attacks. More specifically, this method of defense targets free-branch instruction sequences and removes them from vulnerable executable code. This prevents an executable sequence to perform mid-instruction jumps at the functional level. The following equation analyzes the state and performance of G-Free: $(Gfree \wedge \neg gfreebroken) \rightarrow \neg(frbr \vee mdjmp)$ [7]. The way G-Free removes free-branch instructions is through semantics-preserving code transformations. The figure depicts the role of G-Free defense in ROP. The shaded component displays how G-Free affects ROP attacks: They become disabled due to key prerequisites becoming unavailable. However, return-into-libn attacks remain possible since G-Free cannot recognize the I/O functions of non-libC libraries[7].



Data Execution Prevention (DEP)

Data Execution Prevention is a defense mechanism to prevent unwanted executions of code from system memory locations reserved for authorized programs. DEP does this on the OS level- Windows, iOS and Linux each has their own DEP protocol[7]. DEP accomplishes protection by monitoring executions by programs to ensure that all executions completed by programs are using system memory safely. If not, than the OS will close down the program and notify the user[7].

Address Space Randomization

Address Space Randomization is a memory-protection techniques that protects against buffer-overflow threats. This is done by randomizing the memory location of systems executables. If an attacker chooses to exploit an address space that is incorrect, the application accessed will crash in response.

HyperCropII

Another defense approach against ROP is the HyperCropII, which was created in 2013 by scholars from Pennsylvania State University and the Chinese Academy of Sciences. This is an automatic runtime (on-the-fly) approach that checks the contents of the stack for potential ROP attacks and is able to isolate the damage and optimize for system availability amidst ROP attack attempts. This defense approach targets programs that are vulnerable to buffer overflow attacks. HyperCropII changes the stack of each target process to read-only so that it can detect any writes to the stack. To avoid errors upon illegal write operations, the stack is changed back to allow writes, but call the stack-inspecting component to check if an ROP attack is present. Typically, stack contents will include function parameters, local variables, return addresses to application functions, and return addresses to libraries. The stack inspecting component determines that an ROP attack is present if the contents of the stack contains a ratio of return to libraries to the other three possible contents above 0.8. One limitation is that this defense technique will not defend against attacks using instructions other than return, such as call or jump[4].

Control-Flow Integrity

Control-flow integrity policy requires the program's control flow to follow a path allowed by a control-flow graph (CFG). A CFG must be constructed specifically for each program before program execution[3]. CFG nodes are all valid source and destination addresses, and the edges are all legal control flow transfers. Ideally, when control flow integrity is enforced, all control flows that are allowed to execute must have been previously defined by the CFG. If the program attempts a control flow transfer from a source to destination address, where the associated nodes of CFG don't contain a direct edge, this operation will be blocked[3]. This would theoretically prohibit all types of return-oriented programming attacks, as these attacks rely on manipulation of a program's control flow to execute exploits.

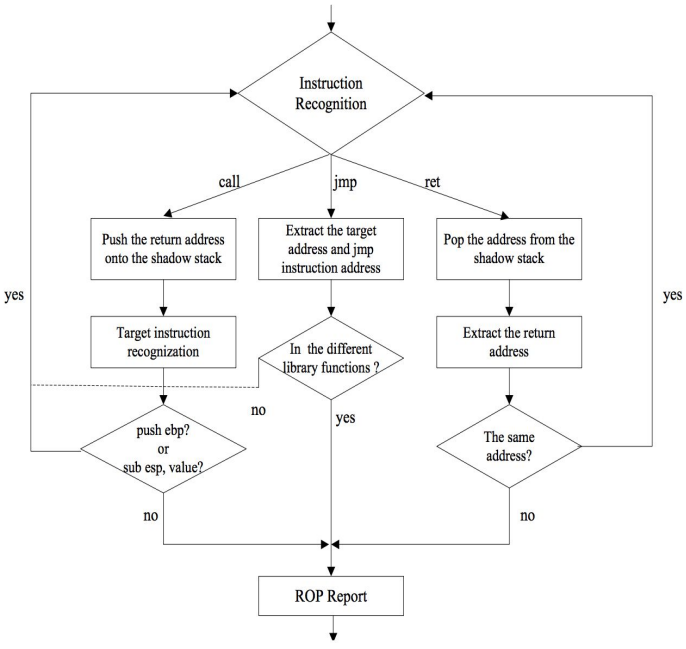
While a full enforcement of control-flow integrity would be an impenetrable defense, its performance and cost overhead would make this approach close to impossible to implement. Therefore, a number of realistic defense mechanisms were developed to implement certain portions of control-flow integrity that researchers found protected against the most common exploits [5]. One implementation of control-flow integrity is called CCFIR (Compact Control Flow Integrity and Randomization). CCFIR implements control-flow integrity directly on

existing program binaries, which significantly reduces policy adoption overhead. Instead of constructing a complete CFG, CCFIR creates a whitelist of allowed target addresses, and restricts all indirect control flow transfers to this list. CCFIR relies on a "Springboard" section to hold the whitelist of allowed targets, and manage all indirect control transfers. The Springboard also randomizes the order of control transfers to further deter control flow exploits.

ROP Detection

When it comes to detecting the occurrence of an ROP attack, there are a large variety of implementation options. ROPdefender, a solution designed by researchers from *System Security Lab* in Germany, aims to provide a dynamic tool for users. This tool is built off of the Pin framework, which is traditionally used for performance monitoring and taint-tracking, but allows the tool to run concurrently with the program being monitored. From a high level perspective, ROPdefender operates by maintaining multiple "shadow stacks," which store copies of return addresses as functions are called, and compares return instructions being issued to the processor to catch any type of return address corruption[12]. The downside of this approach is the performance overhead, which causes the program under analysis to run between 1.49x and 2.17x slower. However, this dynamic solution does provide a major benefit in that it requires no "side information," meaning source code or other debugging context. This reduces the time investment necessary for setting up the system, and users can quickly integrate this monitoring for any third-party program.

Another technique designed by scholars from the State Key Laboratory for Novel Software Technology at Nanjing University introduces the ROP Monitor Component. The ROP Monitor Component uses a different detection method for each of the sensitive instructions: return, call and jump. For returns, the ROPdefender technique is used, detecting attacks by using a return address shadow stack. For the call instruction, the target instructions are checked against the beginning instructions. The entry of the function is the target address of the function. If the instructions change from a frame function to a non-frame function, or vice versa, then an ROP attack has been



detected. This technique can be used to detect call and call-jmp instructions, but cannot detect jump instructions. For jump instructions (specifically indirect jumps), an attack is detected if the instruction if there are changes between different library functions. The ROP Monitor Component records the address of the indirect jump instruction first and then looks at the target address that the program jumps to. The figure to the right shows the monitoring workflow of the different instructions. After evaluating the ROP Monitor Component, it was determined that the performance overhead is much higher than others, but this technique also covers three instructions instead of just returns. There are also some limitations of this method. Programs in which call and return instructions do not appear in pairs will break this detection because the ROP Monitor Component assumes that the call and return will be in pairs. Another limitation is found in C++ code, where if the last operation is a function call, then it is replaced with a jump. This jump would move across function boundaries and result in a false alarm [8].

Another tool developed to aid in the detection of ROP attacks is called EigenROP. Similarly to the previously mentioned ROPdefender, EigenROP is based on the Pin dynamic framework. EigenROP uses a unique anomaly detection method based on directional statistics in order to compute harmful or anomalous program execution. The idea behind their algorithm is that common interactions between different program characteristics will appear as an axis when converted to a high dimensional space model. By monitoring the control flow of the program, interactions that are infrequent or highly improbable can be assumed to be anomalous and potentially an ROP attack. The creators of EigenROP chose to use specific characteristics to measure in every program, which were then used in a Fisher Score to quantify how much a programs execution deviates from expected behavior. The characteristics measured were branch predictability, instruction mix, memory locality, register traffic, and memory reuse. Since EigenROP analyzes more than just the programs returns, it is resilient to variations in the ROP attack that don't implicitly rely on the return or jump operations, which makes it a much more effective tool than other detection methods. EigenROP is generally effective at detecting when an ROP attack or variant is occurring during execution of a program. However, it has some significant shortcomings under specific conditions. First, in programs that exhibit a large spread and has a lot of branches, it becomes hard to separate anomalies from benign executions. Second, programs that use far jumps become harder to predict due to the inherent nature of poor branch predictability and memory locality. Finally, short ROP chains were missed by the detection software due to the large sampling intervals taken during benign execution. Overall, EigenROP achieved a 80% true positive rate, and 0.8% false positive rate, which are ~20% higher than other standardized microarchitectural defenses. Despite these shortcomings, the advantage to EigenROP is its dynamic nature, which allows it to be run with any type of program, and can be combined with other ROP defenses to preserve the integrity of the program while providing the maximum protection against ROP and variant attacks [9].

Discussion of ROP Defenses

In a traditional ROP attack model, the attacker gains control over a program's call stack and executes functions in other parts of the system memory. Memory-based defenses, which include address space randomization, data execution prevention, and HyperCropII, largely guard against these types of attacks[4]. With address space randomization, the location of certain instructions in memory are randomized each time the program is run. This makes it difficult for the attacker to directly point to specific instructions in memory. However, the attacker can find a way to leak the current memory locations of different instructions at runtime, and exploit the desired instructions just as much as if address space randomization wasn't implemented. A more secure memory-based defense is data execution prevention (DEP). With DEP, the execution of instructions is restricted to only the allowed areas of memory[7]. Therefore, if an attacker would like to execute an instruction in memory that the program normally doesn't use, DEP will prevent this attack. However, if the attacker exploits instructions in the allowed areas of memory, such as libc, DEP will be useless in preventing attacks. To achieve an even finer level of control over a program's memory access, HyperCropII can be used. While a target program is running, HyperCropII continuously monitors the call stack, and ensures that the majority of return instructions target the program's local memory, instead of an unrelated portion of memory, such as a library. It also locks the call stack as read-only, and combined with call stack monitoring, HyperCropII is able to restrict all returns to stay mostly within a program's immediate memory space[4]. One of the limitations of HyperCropII is that it will only target return instructions, and miss other more sophisticated control flow manipulations.

Cumulatively, memory-based defenses guard against the most common ROP attacks, but are not able to prevent more sophisticated control flow manipulations. For more complicated attacks, control flow defenses are much more effective. Instead of hindering an attacker from accessing certain parts of memory, control flow defenses, such as G-Free and control-flow integrity, restrict the control flow of a program to guard against unwanted control flow manipulations. With G-Free, all of a program's free-branch instructions are protected such that their original control flow transfers cannot be manipulated[7]. This makes it impossible for an attacker to exploit the program using any free-branch instruction G-Free protects and have it divert control flow to anywhere but the original target address. However, G-Free only considers free-branch instructions from libc, and would ignore the instructions that occur in other libraries, such as return-to-libn[6]. The most secure control flow defense is control-flow integrity. With control-flow integrity, a control-flow graph (CFG) is constructed for each program that specifically models all legal control flow transfers between different addresses in memory. If an attacker attempts to execute a control flow transfer that is not contained in the target program's CFG, the control flow transfer will not execute. Theoretically, a control-flow integrity policy implemented to its fullest extent will guard against all possible types of ROP attacks, as ROP

exploits always rely on changes in a program's normal control flow. The main downside of control-flow integrity is that its implementation in a real-world system is nearly impossible due to its enormous cost and performance overhead. While scaled down versions of control-flow integrity have been developed, they are significantly less secure than the ideal control-flow integrity policy, and only marginally better than other control flow defenses.

JOP attacks follow the same framework as ROP attacks with the exception that instead of utilizing return functions to string together gadgets, it uses pop and jmp instructions that mirror the functionality of ROP attacks[6]. Memory based defenses are still able to guard against jop attacks since the source code is not being changed: example, G-Free eliminates a program's free branch instructions which will continue to hinder JOP effectiveness. However, defense mechanisms such as HyperCropII will not be able to prevent against JOP attacks since it monitors the call stack to ensure the appropriate use of return instructions, however since JOP does not utilize return instructions, instead pop and jmp functions are used, then it will not be able to detect these attack types. Overall, memory based defenses will still be able to defend against JOP attacks, however defense mechanisms involving return instructions will not be able to defend against these attacks.

Return-into-libN attacks utilize additional libraries besides libC to create ROP payloads[7]. These attacks render compiler based defense mechanisms useless such as G-Free. G-Free will not be able to defend against return-into-libN attacks because it only eliminates free branches in libC however, does not counter any potential libN calls. On the other end of the spectrum, return based defense mechanisms such as HyperCropII will be able to defend against return-into-libN attacks because it checks the call stack for suspicious return instructions which will still be triggered by return-into-libN attacks since they still use returns to string together gadgets. Overall, compiler based defense mechanisms will not be capable of defending against return-into-libN attacks, while return instruction based defenses will be able to defend against such attacks.

None of the defense mechanisms above can comprehensively cover all ROP attacks types, therefore we recommend SAT Solver as a complete solution to protect against all potential attacks. This is a comprehensive solution because a user can define their vulnerabilities and defense mechanisms as they wish and it is created to account for any combination of defense mechanisms used[7]. Thus a user can cover all potential ROP vulnerabilities with individual defense methods. SAT Solver allows a user to understand the dependencies, key defense points and scenario restraints of each defense mechanisms. SAT Solver will additionally check for any state changes within all areas to understand if a Defense system has been compromised or not (boolean true or false). Although this is an overall successful method of defense protection, there is one large drawback. That is to have complete defense protection using SAT Solver, a user

must first understand all vulnerabilities towards ROP attacks within their system. However, if a user understands all facets of vulnerabilities within their system, SAT Solver is a complete defense technique which prevents all potential ROP attacks.

Discussion of Detection Methods

From the papers we've reviewed, there is a clear distinction made between two categories of detection methods: compiler-based and dynamic solutions. Compiler-based solutions traditionally rely on access to the source code of the program to be analyzed, or some other form of debugging information. On the other hand, dynamic solutions are typically run concurrently with the program being monitored, and do not require any "side information." Because of this distinction, these detection methods utilize incredibly different methods for flagging the occurrence of an ROP attack. In addition, both types have their own benefits and tradeoffs to be considered before implementation.

Compiler-based detection is an approach with a high up-front cost with little to no performance overhead at runtime. The initial overhead is usually related to the reliance on access to source-code, which will then need to be manipulated and recompiled in some way. One classic approach to compiler-based detection is replacing the return addresses with return indices that point to values in a verified return-address table, to detect any invalid return addresses being used. However, this approach would also require the user to recompile any linked libraries of the program, to reduce the false positive rate of the detection tool. Compiler-based detection methods are also vulnerable to indirect jumps, meaning they can be fooled by return-oriented style attacks that do not use actual return instructions. The primary benefit of this implementation is that after setup, the program can be run normally, with no additional code being run alongside it, meaning any slowdown is virtually undetectable.

Dynamic detection is a solution that is quicker for the user to implement, but comes with a higher performance cost. These programs do not require access to any side information, and can immediately be run alongside the program to be analyzed, albeit with a slowdown in instruction speed. Within this category of detection methods, there is still a large variety in methodology, where some solutions are more thorough than others. For example, a basic detection solution simply monitors the frequency of 'return' instructions being called, and compares it to the frequency of returns during normal program execution. This will detect simple return-oriented programming attacks, but is susceptible to attacks that do not explicitly use returns, and can result in false-positives depending on how the baseline frequency is measured. On the other hand, some solutions are more involved, like ROPdefender, which maintains multiple copies of the return addresses as functions are called. These shadow stacks are then compared against the return addresses being used by the program's execution threads, meaning it

will catch any type of attack that manipulates the return address being used. While this is a more robust solution, it also comes at a higher performance overhead from maintaining these shadow stacks. While most detection techniques fail to detect ROP variations, more sophisticated dynamic analysis tools, such as EigenROP, have shown to be effective at detecting these variations, including jump oriented programming and counterfeit object-oriented programming attacks. This is achieved by analyzing more characteristics of the program, including branches, calls and returns, and memory reuse, due to the statistical nature and analyzing benign program execution. However, EigenROP still suffers from most of the same faults as other dynamic techniques, including increased overhead during program execution and a higher false positive/negative rate than traditional defenses.

From our research, we learned that there is a large selection of detection methods to choose from, each with varying levels of security and implementation costs. No solution is perfect, and the choice will depend on the resources and priorities of the implementing user, though return-oriented attacks are quickly outgrowing some of the more naive detection methods.

Conclusion

Overall, ROP attacks are diverse and target various software vulnerabilities, however there are effective defense and detection methods to counter these threats. Defense methods are divided into memory based protections and control flow protections which are effective at defending against specific types of ROP attacks. However we recommend SAT Solver as a comprehensive solution to defense. This is because it uses boolean statements to understand the state of various defense techniques incorporated in a unified defense system. Detection of ROP attacks are divided into compiler based and dynamic detection methods which have different tradeoffs. Compiler based detection methods are more suited for high upfront cost and low overhead maintenance systems while dynamic detection has quicker implementation but greater maintenance costs. Therefore, these various methods are dependant on system restrictions and user preferences. In conclusion using SAT Solver in conjunction with either main detection method is effective is preventing ROP attacks.

References

[1] László Erdődi. “Conditional gadgets from return oriented programming”, 2013
[2] Yefeng Ruan, Sivapriya Kalyanasundaram and Xukai Zou, “Survey of return-oriented programming defense mechanisms,”, 2015
[3] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical control flow integrity and randomization for binary executables,” in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013, pp. 559–573.
[4] X. Jia, R. Wang, J. Jiang, S. Zhang, and P. Liu, “Defending return‑oriented programming based on virtualization techniques” in Wiley Online Library, 2013
[5] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005, pp. 340-353.
[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” *Proceedings of the 17th ACM conference on Computer and communications security - CCS 10*, 2010, pp. 1-14.
[7] R. Skowyra, K. Casteel, H. Okhravi, N. Zeldovich, and W. Streilein, “Systematic Analysis of Defenses against Return-Oriented Programming,” *Research in Attacks, Intrusions, and Defenses Lecture Notes in Computer Science*, 2013, pp. 82–102.
[8] P. Chen, X. Xing, H. Han, B. Mao, and Li Xie, “Efficient Detection of the Return-Oriented Programming Malicious Code” in *Information Systems Security*, 2010, pp. 140-155
[9] M. Elsabagh, D. Barbará, D. Fleck, A. Stavrou, “Detecting ROP with Statistical Learning of Program Characteristics”, *CODASPY’17*, 2017
[10] S. Sathyanarayan, K. Aliyari, “Return Oriented Programming - Exploit Implementation using functions”, 2018
[11] M. Elsabagh, D. Barbara, D. Fleck, and A. Stavrou, “Detecting ROP with Statistical Learning of Program Characteristics,” *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy - CODASPY 17*, 2017.
[12] L. Davi, A.-R. Sadeghi, and M. Winandy, “ROPdefender,” *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS 11*, 2011.