ROPK++: An Enhanced ROP Attack Detection Framework for Linux Operating System

Vahid Moula
Kish Island International Campus
University of Tehran
Tehran, Iran
vahidmoula@alumni.ut.ac.ir

Salman Niksefat
APA Research Center
Amirkabir University of Technology
Tehran, Iran
niksefat@aut.ac.ir

Abstract

A major security challenge for today's computer software is buffer overflow and other memory-related attacks. To exploit buffer overflow vulnerabilities in presence of the classical defense mechanisms such as write-xor-execute, attackers take advantage of code reuse attacks. The code reuse attacks allow an adversary to perform arbitrary operations on a victim's system by constructing a chain of small code sequences called gadgets that are present in vulnerable program's memory. In order to remedy code reuse attacks, many defense approaches have been proposed, each using a different mechanism for detecting attacks and having its own merits and downsides.

In this paper, we analyze and scrutinize one of the most influential Linux-based defense mechanisms called ROPecker. Our analysis shows that ROPecker has weaknesses that may allow an attacker to bypass detection. Then we propose ROPK++ which by adding additional integrity checks, fixes the weaknesses in ROPecker and offers a more effective defensive approach against code reuse attacks in Linux-based systems. We compare the proposed approach with ROPecker in terms of security features and performance overhead and show its superiority and advantages.

Keywords

Code reuse attacks, Return Oriented Programming (ROP), Jump Oriented Programming (JOP), Buffer overflow, ROPecker

I. INTRODUCTION

Among the attacks that exploit memory vulnerabilities, code injection attacks have been well protected by a number of defense mechanisms such as DEP and W⊕X (Write-xor-eXecutable), which prevent the execution of injected binary code (e.g. shellcode) [1, 2]. After advent and deployment of these defense mechanisms in operating systems, a new class of attacks called *code reuse attacks* were introduced by attackers. Two of the most popular variants of such attacks are return-oriented programming (ROP) [3] and jump-oriented programming (JOP) [4, 5]. Code reuse attacks are the predominant technique extensively used by attackers to exploit vulnerabilities in modern software.

To date, many defense mechanisms have been proposed to deal with code reuse attacks; each has its own specific mechanisms of detection and prevention. However, making a balance between security and performance is one of the significant challenges in this area. There are several key factors such as type of detection function and frequency of checks, which makes the defense mechanisms different from each other and affects their security and performance [6].

One of the most prominent defense mechanisms for Linux-based systems is ROPecker [7]. This system proposes a novel sliding window mechanism to select run-time checkpoints, and thus it is one of the most powerful runtime defense mechanisms against ROP in Linux-based systems. However, as we show in this paper, this system has some significant limitations in the detection function, which may allow the attackers to bypass it.

In this paper we propose ROPK++, an enhanced version of ROPecker which can perfectly fix the ROPecker known limitations. This is accomplished by several ideas including adding a new function to ROPecker kernel module code in order to control return instructions directly.

A. Our Contributions

- We enhanced the ROPecker kernel module by arming it with a new security function which makes sure that the return instructions are all call-preceded.
- We implemented our proposed approach on a Linux operating system and also deployed the ROPecker on the same machine so we can do a precise and fare comparison between two systems.
- By doing a number of experiments as well as theoretical feature analysis, we compared the proposed approach with ROPecker and also a number of Anti-ROP approaches in terms of security features and performance overhead and show the superiority and advantages of the proposed system.

B. Paper Organization

This paper is organised as follows. We give a brief introduction on code reuse attacks and available defense mechanisms in section II. Then in section III, we present a number of ROPecker known limitations and weaknesses. In section IV, we propose ROPK++ which is a new mechanism to overcome ROPecker weaknesses and also present some technical details about our implementation. Evaluation and experimental results are shown in Section V. Section VI concludes the paper and outlines future work.

II. CODE REUSE ATTACKS AND DEFENCES

By the advent of mitigation techniques against buffer overflow attacks, attackers have demonstrated subtle ideas and new techniques to exploit software vulnerabilities. One such techniques is code-reuse attacks in which an adversary, instead of injecting malicious code into the address space of the application, exploit the vulnerability by chaining small pieces of code which are already present in the address space of the vulnerable program. These code-reuse attacks have instituted as so-called return-into-libc attacks [8] and have been later generalised to return-oriented programming (ROP) [3] and jump-oriented programing (JOP) [4, 5] attacks. In most real-world attacks, the attackers usually use a combination of code reuse and buffer overflow attacks. For example, they first disable the defense mechanisms against code injection attacks by code reuse attacks and then proceed with the classical attack scenario, i.e., injection of the malicious code into the address space of the vulnerable application and setting the stack pointer to the beginning of the code to execute the malicious code on the target computer [9].

A. Techniques to mitigates code reuse attacks

In this section, we first review the most important techniques for mitigating code reuse attacks.

1) Address Space Layout Randomization (ASLR)

One category of mitigation techniques against code reuse attacks is called address space layout randomization (ASLR) [10]. ASLR randomises the base address and code segments of a running process. Unfortunately, ASLR techniques have two fundamental problems: first, the low entropy in 32-bit systems may allow attackers to bypass ASLR by brute force attacks [6, 11]. Secondly, all ASLR techniques are vulnerable to memory disclosure attacks in which an attacker obtains a run-time address of the memory and uses that address to learn the memory structure and other information to exploit the target system [12, 13].

2) Control Flow Integrity.

Control Flow Integrity (CFI) mechanisms verify indirect branches and validate the control flow of the running programs. There are two key factors for assessing the effectiveness of such defense mechanisms.

The first factor is the exact method used for detecting the code-reuse attack. In fact each defense mechanism has certain indicators to detect attacks. Some methods take into account the behaviour of the indirect branches and by comparing them with a shadow stack which is filled in the information gathering phase (through analyzing the program source) detect the code-reuse attacks [14-16]. In other methods, defense mechanisms detect code reuse attacks by finding the gadget chains in the program and if the number of gadgets in one chain exceeds the permitted threshold, a ROP attack is reported and the program execution is terminated.

The second factor for assessing CFI-based methods is the timing and frequency of the CFI checks. The more the frequency of the CFI checks is, the less the performance of the protected system becomes. Primary fine-grained CFI methods [17] monitor all indirect branches which causes too much overhead. Because of this fact, the basic fine-grained CFI methods were not practical [6]. In contrast, the coarse-grained CFI methods monitor control flow only when certain conditions are met and therefore they impose less performance overhead on the system. An example of such mechanisms is the sliding window technique introduced in ROPecker system [7]. However coarse-grained mechanisms should be used with utmost care because a wise attacker can bypass the system using techniques such as LBR flash [6, 18].

In the next section, we present and analyse the ROPecker system [7] which is one of the most influential Linux-based defense mechanisms against code reuse attacks.

III. ROPECKER AND ITS WEAKNESSES

ROPecker is a Linux-based anti-ROP approach suggested by Cheng et al [7]. It uses the LBR CPU registers to find the potential gadgets execution. However, it emulates the program execution by a special emulator to detect gadgets that are going to be invoked in the near future. To accomplish this, a static offline phase is required to generate a database of all possible instruction sequences which is called IG[1] database.

To limit false positives, this method sets a threshold on instruction code sequences terminating in an indirect branch. ROPecker inspects each LBR entry to identify indirect branches that have redirected the control-flow to a potential gadget. This decision is based on the IG database that ROPecker derived in the static analysis phase. ROPecker also inspects the program stack to predict future execution of gadgets.

CFI checks in ROPecker is triggered whenever the program execution leaves a *sliding window* or a critical system call is invoked. A sliding window is the part of the application code which is considered by ROPecker to be executable, while the instructions outside of the window are considered as non-executable. Therefore, ROPecker sets all memory code pages to be non-executable except the pages where the program is being executed and the most previously executed pages. When the program aims to execute code from a non-executable page, ROPecker adjusts the sliding window and performs CFI validations. In addition, ROPecker hooks into some critical Linux functions, and before executing those functions, CFI validation is enforced. These functions are *mprotect()*, *mmap2()*, and *execve()*.

All in all, ROPecker has a number of interesting features such as a sliding window mechanism and the ability to detect both ROP and JOP code reuse attacks which make it a prominent defensive mechanism in Linux-based systems and also a subject of debate among researchers. In the next section, we point towards a number of weaknesses in this system that an attacker can misuse to bypass ROPecker and exploit the target system.

A. ROPecker weaknesses

A number of vulnerabilities have been identified in ROPecker that allow an expert attacker to bypass this defense

[1] Instruction and Gadget

mechanism and still perform a code reuse attack [18, 19]. In this section we give a detailed review of such weaknesses.

1) Short gadget chain length

ROPecker is highly dependent on the structure and behavioural attributes of ROP attacks and identifies such attacks by using their specification. One of the parameters taken into account by ROPecker in detection algorithm is the length of the gadget chain that is going to be executed at run-time. When ROPecker detects a suspicious gadget, it suspends the program execution and starts to control the past program execution using LBR and the future program execution using a new emulator tool which emulates the near-future execution environment and finds ROP gadgets. If a gadget chain is found with a length more than the determined threshold, ROPecker terminates the program execution. In a recent work [19], a method is proposed which launches a successful code reuse attack with a gadget chain that is less than the defined threshold detectable by ROPecker. This specific attack only uses three gadgets and therefore can easily bypass ROPecker detection.

2) Using long-NOP gadgets

ROPecker has a pre-processor phase, which analyses the binary code of the programs to find possible gadgets and put them into a so-called Instruction and Gadget Database (IGD). This database is created in the offline phase and the detection function uses it to minimise performance overhead during CFI checks. ROPecker's choice to define a sequence of instructions to be considered as a gadget is a sequence of six or fewer instructions. This makes it nearly trivial to find gadgets that are longer than 6 instructions but still have the desired behaviour. For example, on 64-bit systems, gadgets consisting of popping off the registers *r10* through *r15* followed by a *ret* instruction is seven instructions long. So not only this is a useful gadget, but this kind of gadget is very common in program binaries and ROPecker's failure to recognise it as a gadget is a serious limitation [18]. Therefore, an attacker can interleave gadgets with higher than six instructions into the chain of gadgets to reset the ROPecker's gadget counter, bypass ROPecker and exploit the victim system [18]. The ROPecker evasion attack works by inserting a termination gadget (long-NOP gadget) between normal gadgets. These termination gadgets are not counted by ROPecker due to their length, and therefore the number of consecutive gadgets becomes less than the defined threshold and ROPecker cannot detect the attack.

IV. The Proposed Enhanced Architecture: ROPK++

In this section, we come up with our ideas for enhancing the ROPecker system which fixes the above mentioned weaknesses. By thorough analysis of ROPecker and also analyzing the related defensive mechanisms against ROP [16, 20], we observed that stating a subtle policy for validating the return branches directly can mitigate these weaknesses and improve the efficacy of the system.

In fact our idea in ROPK++ is to do some extra checks using the LBR registers to make sure that the return instructions are all call-preceded.

A. Threat Model

In this work, similar to other Anti-ROP approaches, we assume that the operating system is protected by the DEP (Data Execution Prevention) or W⊕X that prevents the traditional code injection attacks to be launched. However like ROPecker, we do not assume that the Address Space Layout Randomization (ASLR) mechanism is enabled and in fact the proposed defense mechanism does not rely on ASLR.

B. Call-Preceded Gadgets

When we monitor the control flow of a process in the case of a ROP attack versus the case where the process is executing normally, we see a distinctive behavior by monitoring the ret instructions of the running process. At runtime, this crucial difference enables us to easily distinguish between a ROP attack and a normal behavior.

In a typical program, the target of a legitimate ret instruction corresponds to the instruction right after the call instruction of the respective caller function. In contrast, in the case of an attack, a ret instruction at the end of a gadget transfers control to the first instruction of the next gadget, which is unlikely to be preceded by a call instruction. This is because gadgets are found in arbitrary locations across the code image of a process, and often may correspond to non-intended instruction sequences that happen to exist due to overlapping instructions [3].

In ROPK++, ret instructions that transfers control to an instruction not preceded by a call are considered as illegal, and the observation of such ret instructions shows that a ROP attack is happening. Therefore the program execution is terminated. In fact, alongside with enforcing gadget chain policy[2] that is used in the ROPecker system, we use an extra constraint which makes sure that all ret instructions are call-preceded. As there is no need to trace the execution of all instructions and just the ret instructions are monitored, the implementation of the approach is simple and imposes a little overhead on the system.

C. ROPK++ detecton algorithm

To do this, we improved the detection algorithm of ROPecker with a new workflow as depicted in Figure 1.

In Figure 1, we show the sequence of code reuse attack detection mechanism in ROPK++. Whenever a page fault is triggered due to jump of the program control flow out of the sliding window or invocation of a critical function call, the ROPK++ kernel module is activated and invokes the attack detection function which ensures the accuracy of the performed jump. So, at the start of the ROP attack detection function, we control the accuracy of return addresses in the LBR, and if we find illegal *ret* gadgets in the LBR, program execution terminates and ROPK++ announces a code reuse attack. Otherwise, the detection function proceeds and analyses the past and future gadgets in control flow and counts them. If the

[2] Gadget chain policy: This policy considers a defined threshold for the maximum number of gadgets that can be chained together before issuing a ROP attack alert.

detection function finds a suspicious gadget chain with a length more than the defined threshold, program execution terminates.

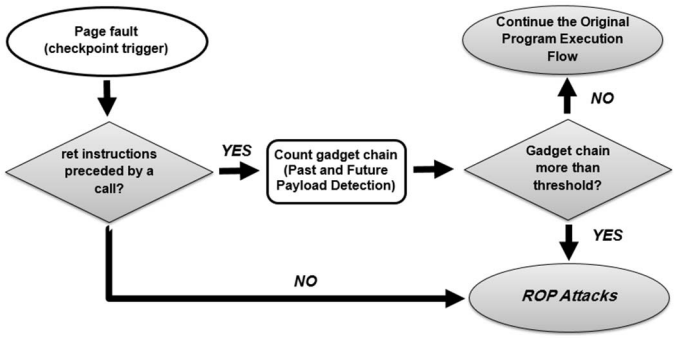

Figure 1: ROPK++ code reuse attack detection and prevention diagram

ROPK++ controls return branches by use of LBR to find illegal *ret* instructions. When the ROP attack detection function is triggered, at first it checks whether the targets of the previously executed return instructions are preceded by a *call* instruction or not. If there is a *ret* instruction which is not call-preceded it is considered as an illegal *ret* instruction.

In x86 architecture, when a *call* instruction is invoked, the address of the *call* instruction is pushed into the stack and when the function returns, this address is popped from the stack and the program execution proceeds from the next memory address after the *call* instruction. In ROPK++, each time ROP attack detection function is invoked by the kernel module, it makes sure that all returns are call-preceded with the help of the LBR. In this procedure, first of all we find all return instructions in LBR. Next, we control the memory word before the target address of return instruction and finally, if we find opcode of *call* instruction in memory word before target address of the return instruction, ROP detection function counts past and future gadgets and control chain length. On the other hand, if we cannot find any *call* instruction in memory word before return instruction, program execution terminates and a ROP attack is announced.

To implement the above functionality, we developed a new function called *ret-check()* which is embedded into the ROPK++ kernel module. In this function, the module only checks return addresses of user space codes. Since noisy records may increase the false-negative or false-positive rates of the ROPK++, our solution is to traverse the LBR queue backwards starting from the most recent one, until we reach one LBR entry that indicates a context switch. The context switch entry can be recognised by its unique feature. Namely, its source address is in the kernel space, and its destination address is in the user space. Other types of branches must have the source and destination addresses in the same space. The ROPK++ module does not use LBR entries before the context switch because the older entries are likely noisy records (e.g., belong to unprotected applications).

D. The ret-check() function

As we mentioned above, we have improved the ROPecker kernel module code by adding the *ret-check()* function. The kernel module controls ret-based gadgets by the help of this function. In this section, we present an in-depth details of the *ret-check()* function and describe how it detects non call-preceded gadgets. First of all, *ret-check()* retreives all LBR entries by use of another function named *get-lbr-pairs()*. We faced a number of challenges when starting to use the LBR feature of the Intel CPUs. For example this feature is disabled by default and can only be enabled/disabled through certain *Model Specific Registers* (MSRs). When LBR is enabled, the CPU stores the last executed branches in a set of MSRs, on-chip. The values of these registers can be read by using a special CPU instruction, called *rdmsr*, which works only from the privileged mode (ring 0). Next, the LBR records are passed to *ret-check()*, and it starts to control each pair of the LBR. Each LBR pair consists of a source and target address of each branch. In this step, we just control branches that are related to protected program and we ignore branches which occur after a context switch. Next, we should filter the return instructions from other LBR entries.

E. Filtering return instructions from LBR entries

After retrieving all LBR records whenever an interrupt occurs due to the page fault, we should filter return branches from other kinds of the branches (call and jump branches). First of all, we get the source address of the branch from LBR. Next, we refer to the source address and control the memory word to find binary opcode related to return instructions. In x86 platforms, return from procedure is specified by assembly opcodes of *0xC2*, *0xC3*, *0xCA* and *0xCB*. In 32-bit systems, each memory word contains 32 bits and the first 8 bits used for instruction opcode, so the bit's number 0 to 7 in each memory address define the type of the instruction. In this work, we filter return instruction by use of *copy_instruction()* function. If the hex value returned from the first 8 bits of the memory word equals to one of the assembly opcodes value mentioned above, the LBR record is considered as a return instruction, and the destination address should be checked out. In all other cases, the module ignores the LBR record and proceeds to the next branch.

F. Checking out the destination address of the return branches

When kernel module finds and filters the return branches from other branches in LBR entries, the module has to control the destination address of the filtered branches to check if the assembly opcode value located in the memory word before the destination address contains the *call* instruction or not. To satisfy this need, the module should check the 8 first bit used for instruction code to find assembly opcodes value *0xE8* and *0xFF* that indicates *call* procedure instruction. In this step, if the module can find any assembly opcode mentioned above, the branch is legal and the program execution continues. Otherwise the module terminates the program execution and announces a ROP attack.

V. Evaluation and Experimental results

In this section, we first evaluate the security features of ROPK++ and compare it with ROPecker and other significant defense approaches. Then we compare the performance overhead of ROPK++ with ROPecker. To do the comparison, we chose a number of important measures that are vital in all defense approaches. First, to have a fair comparison we

launched ROPK++ and ROPecker modules in the same environment and configured them both to protect a simple vulnerable C program. Then we designed a code reuse attack (ret-to-libc) with a chain of 9 gadgets which its length is more than the permitted threshold (6 gadgets) and another with a length of 4 which is lower than the detectable threshold of ROPecker. We evaluated both methods by launching various attacks such as flashing gadgets and long-NOP gadgets. The results are depicted in Table 1.

Table 1: ROPK++ vs. ROPecker – Comparison of security features

Type of Attack	ROPK++	ROPecker
Attack with chain of 9 gadgets	✓ Detect	✓ Detect
Attack with chain of 4 gadgets	✓ Detect	× No Detection
Attack using long-NOP gadgets	✓ Detect	× No Detection
Attack using flashing gadgets	✓ Detect	× No Detection

As depicted in Table 1, considering a code reuse attack that contains more than 9 gadgets, both methods prosperously detect the attack. Next, we try the exploit of [19] which has a short gadget chain. In this case, the ROPK++ successfully detects the attack, however the ROPecker does not detect the attack because the exploit uses a gadget chain with a length less than the defined threshold of 6.

In the case of an attack that uses long-NOP gadgets between normal gadgets, ROPecker is bypassed because the long gadget reset the ROPecker counter. However, ROPK++ successfully detects the attack.

Considering the case of flashing gadgets [18], there is a payload which uses a gadget to load the potential page into the sliding window and then uses flashing gadgets to clear the LBR entries. ROPK++ successfully detects this kind of attack because when the exploit is going to load a potential page by jumping out of the sliding window, a page-fault is triggered and ROPK++ can detect the non-call-preceded gadget and stop the exploit.

A. Comparison with other code reuse defense approaches

In this section, we do a comparison of ROPK++ with four typical ROP defense approaches. The first measure for comparison is the type of code-reuse attack that can be detected by each defense method. Attacks can be either ret-based, jmp-based, call-based or a composition of them. The next measure is the kind of indirect branch that can be detected by each defense method. The other measure is whether the mechanism supports behavioral-based attack detection. In the behavioral-based approach, ROP attacks are detected by side effects on the victim system, like gadgets chain length. Finally, we state whether the defense mechanism needs to perform any binary rewriting on the program's binary code. Some defense approaches do binary rewriting on the protected program to trace their protection method. This may have undesirable side effects since the program's integrity is violated. The result of the comparison is depicted in Table 2.

Table 2: ROPK++ vs. other code reuse defense approaches

	Code-reuse attack type	Indirect branch detection	Behavioral Based detection	Binary rewriting?
ROPGuard[21]	***Ret-based***	***Ret branches***	**-**	***YES***
CCFIR[22]	***Ret/JMP/Call-based***	***All branches***	**-**	***YES***
KBouncer[20]	***Ret/JMP/Call-based***	***Ret branches***	***Gadgets chain control***	***YES***
ROPecker[7]	***Ret/JMP/Call-based***	**-**	***Gadget chain control***	***NO***
ROPK++	***Ret/JMP/Call-based***	***Ret branches***	***Gadget chain control***	***NO***

B. Performance Overhead

In this section, we evaluate the performance overhead of ROPK++ with ROPecker. There are two important factors, which play a significant role on the overhead caused by a defense mechanism. The first factor is the frequency of checkpoints used during protection. The less the number of checkpoints is, the less the overhead of the system becomes. The second factor is the ROP attack detection function and the complexity of which these functions detect the attacks.

Considering the first factor, because both methods use sliding window and also critical function calls to trigger checkpoints, there is no extra performance overhead for ROPK++ comparing to ROPecker. Regarding the second factor, note that we have added a new function to the kernel module to check the return branches directly. This function controls and validates return branches by use of LBR entries. Since most of the LBR burden is handled inside the CPU, there is no significant overhead by adding this function to the system. To prove this point, we use a benchmark tool called UnixBench [23] to evaluate and compare ROPK++ performance with ROPecker. The results are depicted in Table 3.

Table 3: ROPK++ vs. ROPecker – Performance comparison

	Maximum load with ROPecker	Maximum load with ROPK++	Overhead
dhry2reg	***27631195***	***27565586***	*0.2*
whetstone-doubl	***3702***	***3693***	*0.2*
excel	***1051***	***1046***	*0.7*
Pipe	***1352676***	***1343245***	*0.7*
syscall	***2211321***	***2191423***	*4.7*

In Table 3, "maximum loads" indicates total load per second (LPS) on the system per some Linux tools, which have been put under protection by ROPecker and ROPK++. This analysis shows that the new method has less than 1% overhead than prior ROPecker.

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed ROPK++ which is an enhanced version of ROPecker, a well-known defense system against code reuse attacks. This is accomplished by adding a new security function to the ROPecker kernel module code in order to check whether the return instructions are call-preceded. The real-world exploits are usually a combination of different types of gadgets (*ret*, *jmp* and *call*) and in most of these exploits ret-based gadgets play an important role. So not only ROPK++ can handle

ROPecker limitations, but it offers a more effective defense mechanism against general code-reuse attacks.

Using experimental results, we showed that ROPK++ can detect a wider range of attacks and its performance overhead is not significant comparing to ROPecker.

Regarding shortcomings and future work, it should be mentioned that no code reuse defense mechanism is 100% bullet-proof and ours is not an exception. For example, if an attacker can find suitable gadgets in the code that are all call-preceded, and then create a chain of gadgets that is shorter than the defined threshold, our proposed method can also be bypassed. Although this attack scenario is very hard and may not be feasible in practice, it may be an interesting future work that leads to more secure defense systems.

VII. Acknowledgement

This work is supported by APA research center (*http://apa.aut.ac.ir*) at Amirkabir University of Technology, Tehran, Iran.

VIII. Refrences

[1] Microsoft. (2006). *Data execution prevention (DEP)*. Available: http://support.microsoft.com/kb/875352/EN-US

[2] S. Designer. (1997, Non-executable stack patch. Available: http://lkml.iu.edu/hypermail/linux/kernel/9706.0/0341.html

[3] 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*, 2007, pp. 552-561.

[4] 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*, 2011, pp. 30-40.

[5] 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*, 2010, pp. 559-572.

[6] L. V. Davi. (2015). *Code-Reuse Attacks and Defenses*. Available: http://tuprints.ulb.tu-darmstadt.de/4622/

[7] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[8] S. Designer. (1997, lpr LIBC RETURN exploit. Available: http://insecure.org/sploits/linux.libc.return.lpr.sploit.html

[9] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming," in *NDSS*, 2015.

[10] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, 2004, pp. 298-307.

[11] L. Liu, J. Han, D. Gao, J. Jing, and D. Zha, "Launching return-oriented programming attacks against randomized relocatable executables," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, 2011, pp. 37-44.

[12] F. J. Serna. (2012, CVE-2012-0769, the case of the perfect info leak. Available: http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf

[13] A. Sotirov and M. Dowd, "Bypassing browser memory protections in Windows Vista," *Blackhat USA,* 2008.

[14] C. Tzi-Cker and H. Fu-Hau, "RAD: a compile-time solution to buffer overflow attacks," in *Distributed Computing Systems, 2001. 21st International Conference on.*, 2001, pp. 409-417.

[15] M. Frantzen and M. Shuey, "StackGhost: Hardware Facilitated Stack Protection," in *USENIX Security Symposium*, 2001.

[16] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *USENIX Security*, 2015, pp. 28-38.

[17] 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.

[18] N. Carlini and D. Wagner, "ROP is Still Dangerous: Breaking Modern Defenses," in *USENIX Security*, 2014.

[19] A. Sadeghi, F. Aminmansour, and H. R. Shahriari, "Tiny Jump-oriented Programming Attack (A Class of Code Reuse Attacks)," in *Information Security and Cryptology (ISCISC), 2015 12th International Iranian Society of Cryptology Conference on*, 2015, pp. 52-57.

[20] V. Pappas. (2012, kBouncer: Efficient and transparent ROP mitigation. Available: http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf

[21] I. Fratrić. (2012, ROPGuard: Runtime prevention of return-oriented programming attacks. Available: http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf

[22] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant*, et al.*, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*, 2013, pp. 559-573.

[23] (2016). *UnixBench*. Available: https://github.com/kdlucas/byte-unixbench