ROP Defense Using Trie Graph for System Security

Alex Yao Chu Zhu, Auckland University of Technology, New Zealand
Wei Qi Yan, Auckland University of Technology, New Zealand
Roopak Sinha, Auckland University of Technology, New Zealand

https://orcid.org/0000-0001-9486-7833

ABSTRACT

Most intrusion detection systems (IDS)/intrusion prevention systems (IPS) cannot defend the attacks from a return-oriented program (ROP) that applies code reusing and exploiting techniques without the need for code injection. Malicious attackers chain a short sequence as a gadget and execute this gadget as an arbitrary (Turing-complete) behavior in the target program. Lots of ROP defense tools have been developed with satisfactory performance and low costs overhead, but malicious attackers can evade ROP tools. Therefore, it needs security researchers to continually improve existing ROP defense tools because the defense ability of target devices such as smartphones is weak, and such devices are being increasingly targeted. The contribution in this paper is to propose an ROP defense method that has provided a better performance of defense against ROP attacks than existing ROP defense tools.

KEYWORDS

Code-Reuse Attack, Cyber-Attack, Network Security, Return-Oriented Programming

INTRODUCTION

It is less feasible for relying on code injection of a cyberattack now because Intrusion Detection Systems/Intrusion Prevention Systems (IDS/IPS) provide systems with a stronger defense ability. Therefore, malicious attackers tend to apply a code-reuse attack to circumvent IDS/IPS detection. Return-Oriented Programming (ROP) is a variant of code-reuse-based security exploitation (Shacham, 2007), ROP has become a favored methodology to compromise computer security because it is more concealing and has a stronger penetrating ability by bypassing the IDS/IPS. ROP has been evolved for more than a decade and has become the mainstream of malicious exploitation now.

An ROP attack needs to overwrite a stack in memory for buffer overflow, heap overflow to manipulate the control flow of a program. Malicious attacks will scan the memory to search the desired sequence instructions, construct gadgets from the existing code of the program or linked libraries. They will chain these gadgets to accomplish an ROP attack without injecting any malicious codes.

Pertaining to defend against ROP, even though there are already various ROP defence tools based on address space layout randomization (ASLR), Data Execution Prevention (DEP), frequent return...
instructions, frequent gadgets, shadow stacks, and slip window. Unfortunately, the newly evolved ROP attacks have utilized multiple techniques to complete gadgets chaining, malicious attackers can evade these ROP defence tools and the performance overhead (computational cost) of some defence tools is too high (Wang, Xie, Wang, & Rong, 2018). In this paper, the ROP attack and existing ROP defence algorithms are summarized and a novel ROP attack defence countermeasure is proposed. The contribution of this paper is to propose a novel ROP attack defence methodology: (1) To apply Trie graph to present the protected program procedure to monitor control flow integrity; (2) To monitor the turn-up number of gadgets comparing with data of database for each node of Trie graph in the benign program to discover abnormally chained gadgets for ROP attack.

RELATED WORK

ROP attacks were firstly documented in 2007 by Shacham, (2007) for the Intel x86 architecture. Among the state-of-the-art cyberattacks, ROP attack is a most effective cyberattack (Ding et al., 2012) illustrated by recent ROP attacks that have compromised Harvard architecture-based platforms, i.e., Apple iPhone and Sequoia AVC advantage that is impossible to be achieved by using traditional cyberattacks based on injection code (Checkoway, et al., 2010).

An attacker deliberately reuses code snippets inside the program to accomplish the execution of malicious codes (Li, et al, 2018). The defined steps of this process chain sequence instructions together as a basic block within the memory address space (Hund, Holz, & Freiling, 2009). Gadgets are integrated together to replace the in action calls in Return-to-LIBC (McClure, Scambray, Kurtz, & Kurtz, 2009). There are six sorts of ROP evolution: 1). Return-to-LIBC; 2). Jump-Oriented Programming; 3). String-Oriented Programming; 4). Blind Return-Oriented Programming; 5). Signal Return-Oriented Programming; 6). Function-Oriented Programming.

Return-to-LIBC

Return-to-LIBC is one of ROP without a non-executable stack. It typically targets the process of returning address pointing to where the shared library, such as a C library that is loaded and accessible through UNIX processes (Shacham, 2007). The existing codes from the linked library and text segment can be leveraged to escalate privilege. A Return-to-LIBC changes a return address to direct to a new location of RAM to control program process flow. The requested arguments to the shared library function are stored in a stack where malicious attackers can access, modify them, and invoke other library functions. Return-to-LIBC attack allows a malicious attacker to execute arbitrary code. This is extremely difficult to be detected by using IDS/IPS as it neither does it modifies existing codes, nor injects malicious codes.

Jump-Oriented Programming (JOP)

JOP (Bletsch, Jiang, Freeh, & Liang, 2011) utilizes jump instruction to replace return instruction in program code. It eliminates the return function method and applies return-like instructions such as POP+JMP. This exploitation dispatches and executes gadgets via a dispatcher gadget. It has been proved successful in GNU Lic Library. JOP only requests EIP when registers or memory locations are utilized to process the dispatcher gadget. The initialized gadget can fulfill this task by firstly directing control flow. The initialized gadget may find relative register via the obtained value from memory or by logic or arithmetic operation. Once this process is completed, the initialized gadget will move to the dispatcher and start the Jump-Oriented program.

String-Oriented Program (SOP)

SOP takes advantage of format string exploitation as a parameter and injects a dynamic dispatcher into the running process(Payer, 2013). Format string attack happens when a malicious attacker crafts
an input string and lets an application treat it as a normal command so that the malicious attacker can write arbitrary code in an arbitrary memory location, even writing NUL byte. Malicious attackers scan and read stack, heap, exploit to make a segmentation processing fault, execute malicious code, and even run shellcode to obtain root privilege.

**Blind Return-Oriented Program (BROP) Attack**

A crafted string is inputted into a system monitor whether a process crashed or not (Bittau, Belay, Mashtizadeh, Mazières, & Boneh, 2014) so as to deduce a single bit. Secondly, the memory is scanned byte-by-byte to discover the malicious attacker’s desired gadgets and obtain writing ability in the program. Then, enough binary is dumped to establish a shellcode and attack the root. However, it still requires stack vulnerability, and the server restarts after a crash. BROP can defeat ROP defense of no-execute protection (NX), address space layout randomization (ASLR), and stack canaries.

**Counterfeit Object-Oriented Program (COOP)**

COOP starts to hijack one of the target objects in C++ applications, malicious attackers repeatedly invoke virtual functions existing in applications on counterfeit C++ objects (carrying attackers’ choice of VPTR and data fields) showing benign control flow without exposing the malicious characteristics (Schuster et al., 2015).

**Function-Oriented Programming (FOP)**

It is a new evolved ROP that does not alter the return address, a chained gadget must jump to the beginning of the function, which will enhance the difficulties of detection with abnormal control flow integrity processing execute in the C program. FOP must firstly find a dispatcher function gadget that can glue gadgets into a chain and dispatch gadgets to launch ROP (Guo, Chen, & Shi, 2018).

**Signal Return-Oriented Programming (SOP)**

SOP loads values to the registers of the stack via the SIGRETURN Linux signal invokes the SIGRETURN syscall to exploit the root and inserts very stealth proof of concept backdoor in UNIX like program system (Mabon, 2016). It controls program processing flow behavior by embedding a “weird” malicious program.

**The Existing Defence Countermeasures of ROP**

Various ROP defence strategies are normally employed for gadget elimination and/or control flow integrity. It is impossible to eliminate all gadgets that can be used for potentially malicious purposes because some of them support program functionality. Control flow integrity assures the program processes in accordance with program design to execute. It will trigger a security alarm when the control flow integrity is violated. Some defence countermeasures apply to analyze run-time traces on-the-fly to monitor control flow integrity, but the high detection latency may too late to detect ROP attack (Cheng et al., 2014). There are some defence countermeasures that have a very good performance of defence ROP, but their performance overhead is too high to be applied practicable.

The following methodologies are existing main ROP defence countermeasures: 1). ASLR makes the address of the stack, heap, and external libraries random for each execution (Shacham et al., 2004). 2). Data Execution Prevention (DEP) makes the memory location that cannot be simultaneously writable and executable (Gao, Zhou, & Liu, 2013). 3). Frequent return instructions are proposed to use dynamic binary instrumentation to count the number of consecutive instruction sequences between two return functions. An alarm is triggered if there are successive five instructions ending with the return. 4). Frequent gadgets between two return opcodes. An alarm is triggered if there are successive five chained gadgets between two return opcodes. 5). Shadow stack stores a copy of the return address at function entry on a separate shadow stack where saved return addresses are duplicated and later compared with the value in the original stack at function exits so as to guarantee the program control...
flow integrity (Davi, Sadeghi, & Winandy, 2011). Sliding window detects ROP by monitoring the presence of a sufficiently long chain of gadgets at run-time in checking points. It sets the most recent visited code regions of the protected application as executable and leaves the application code outside of the window as non-executable (Cheng et al., 2014).

**ROP Attack Defence Evasion**

ASLR can be defeated by a brute-force attack (Cheng, Zhou, Miao, Ding, & Deng, 2014). The defence mechanism will be inefficient if any function in memory addresses can be discovered by a brute-force attack. DEP can be evaded by String-Oriented programming (Payer & Gross, 2013). Frequent return instructions can be easily bypassed by JOP without return (Checkoway et al., 2010). Most ROP attacks contain less than five consecutive instruction sequences to evade the threshold warning positive attack of some of the defence countermeasures. There are lots of return-like effect instruction sequences to be applied to ROP (Checkoway et al., 2010). Therefore, malicious attackers easily evade IDS/IPS by applying various return-like instruction sequences. Frequent gadgets can be defeated by gluing gadgets (Checkoway et al., 2010). Malicious attackers deliberately glue several consecutive instruction sequences with mingling long and short gadgets, chain many segmented gadgets together so as to reduce the number of gadgets to pass the threshold of alarm. Shadow stack happens with false positives when the system-calls “setjmp” and “longjmp”, two functions permit to bypass multiple stack frames. Another evasion is to enlarge the length chained gadget and disperse the chained gadgets in several successive sliding windows so that the part of the whole chained gadget may pass the threshold the detection of ROP in a sliding window (Cheng et al., 2014).

Defence systems normally check the Last Branch Record (LBR) stack to decide whether there is malicious code and mark a checkpoint, but it can be circumvented by jumping over through defence ROP tools. Some ROP defence countermeasures employ binary code on-the-fly to exam selective key functions to defeat ROP, but they do not fully defend all LBR approaches. Therefore, they are unavoidable to be bypassed if ROP attacks do not apply such approaches. There are other circumventing techniques, it is a stack pivoting attack that the malicious payload is hidden in a heap, a global data region, or other memory regions to control the stack pointer direct to the malicious payload in suitable time. Defence countermeasure tools of ROP, such as RAP, IFCC, and CFG may be defeated by this evasion. ROPguard and KBouncer do not supervise all paths, only selected important paths execution control flow of API and or other function calls are not able to defend ROP. It is inevitably to be bypassed through the path that is not supervised. There are four to eight number registers. It is potential for malicious attackers to evade defence mechanisms by forming gadgets in API with LBR number of legitimate LBRS if the number of registers is small. ROPecker examines the past processing in LBR registers and future processing by comparing with the gadget of the database and inspecting the stack trace. However, there is a loophole that an attacker can execute gadgets of monitored applications to bypass detection (Cheng, Zhou, Miao, Ding, & Deng, 2014). If the random key of G-free defence tool is leaked, the defence is useless. Malicious attackers may circumvent ROP defence countermeasures by adopting non-control data attacks. Even though some defence countermeasures enforce control flow integrity and identify anomalies instruction jump in the memory address, it is unfortunate that they can be circumvented by changing periodically dispatchers or arranging for processing long-running functions.

Summary: ROP attack may cause computer security serious harm and it has already been the mainstream cyberattack now. ROP attack is tough to be detected by IDS/IPS because it reuses the existing codes of the targeted programs without injecting malicious codes from outside of the targeted program. Even though many ROP attack defence tools have been developed, previous works either still have some drawbacks that malicious attackers are able to evade them and/or the performance overhead is too high and need to reduce lower.
THE PROPOSED ALGORITHM

To the best of the authors’ knowledge, this is the first time that this novel methodology is proposed. The design of this algorithm is based on two essential components of ROP attack:

- To deviate from control-flow integrity.
- To chain gadgets as malicious code.

Authors proposing algorithm of defence ROP attack is different from ASLR, data execution prevention, frequent return instructions, frequent gadgets, shadow stack, and sliding window defence countermeasure of ROP in related work, the novel algorithm of this paper is proposed to represent the protected program procedure as a Trie graph. Each node may be used to store node ID of a database, object name or function name, relations among nodes, return number, gadget, gadget length, the number turn up of gadget, dispatcher, BROP gadget number, stop gadget number, “strcmp” gadget number, “vfgadget” number, “sigreturn-call” number, FOP Dispatcher, etc in a mathematical matrix of data structure. The authors allocate all objects or functions of the program to nodes from the root of the Trie graph to all descendant nodes according to the protected program character. The connections with the uni-directional line between nodes indicate benign program routines. Each function is disassembled into gadgets. The data is stored in the database in a static state. When a program starts to execute in running time, in Figure 1, the algorithm monitors whether a coming child node is related to the relevant parent node, all invoked gadgets whether they have stored in the database for each node and their turn up times does not exceed the recorded maximum times that have been calculated in a static state. If the number of returns is less than the recorded return number, it means that the control flow has been altered. It is also to check whether there are successive five chained gadgets or return instructions. This algorithm selects the maximum length of gadgets to store, it proposes to avoid gluing gadget attacks. Attackers may chain system-call in a shared library or construct system-call by chaining gadgets. The proposed algorithm checks whether an executing system-call with authority. The proposed algorithm monitors occurred Dispatcher gadget, BROP gadget, stop gadget, “strcmp” gadget and “vfgadget”, “sigreturn-call”, FOP Dispatcher gadget turn up number whether they exceed the maximum turn up number. If the entry of the Global Offset Table (GOT) is requested to alter, it should be examined whether the request is originated from the Procedure Linkage Table (PLT) or with authority. This algorithm monitors executing codes whether they contain malicious sub-strings for format string commands to detect string-oriented programming attacks, whether SetJMP buffer (JOP), VTable Pointer (ROP without return), Function Pointer (ROP without return), Last Branch recording, or SHE pointer is overwritten without authority. It is an ROP attack if the occurred data is different from relative data in the database.

The proposed algorithm regards that the control flow integrity program shall be unidirectional moving from the root of Trie to its descendant nodes according to benign processing routes. If the invoked gadget or the invoked function does not belong to its parent as children node, that means, the control flow has deviated. If a program user switches his/her mind to jump from one node to another that does not obey the unidirectional route, i.e., retrolessly jumps back to parent or ancestor node, then it goes down to other descendants. This algorithm will forcefully terminate the thread from the starting node to the ancestor node that is the closest one to the root to avoid ROP attacks. This methodology is applied to accomplish tasks via the thread restarts from the ancestor node where it stops and executes the rest of the unidirectional route.

EVALUATIONS

To implement the proposed algorithm, a simulation environment is set up for virtual cloud computing in Oracle VM VirtualBox, the configurations are X86 64bit Ubuntu 12.04 with kernel 3.2.0-29-general-
paed, MySQL database version 5, Kali; ROP Emporium, Pwnlib.rop, Pwntool, Radare2, Gdb-peda, ROPGadget, GNU Binutils, Objdump, Ropper, ROPgadget, GNU Binutils, ROP Emporium, Exploit Exercises, GNU C Library and GNU LIBC, YaSSL library.

Experiment Stage 1: authors select 110 vulnerable programs. First of all, the authors use “gdb-peda $ pattern create n (n is a positive integer)” command in gdb-peda application to check whether the targeted program contains buffer flow or heap overflow bugs. The authors can find buffer flow or heap overflow memory addresses in EIP. In the case of a targeted program is vulnerable to buffer flow or heap overflow, to read arbitrary memory address by appending a different number of %#08x in “$./program name %#08x” running program command. The pattern of the whole memory addresses can be revealed via this format string exploitation. Then to write malicious code into arbitrary memory address, even shellcode: “$./program name (printf “desired compromising memory address”)%#08x.%08x.%n ”. This is prepared to chain gadgets and execute shellcode later. To use “gdb-peda $info function” discover all function names and their memory addresses, to apply “gdb-peda $disasm” command to disassemble code for the main function, to utilize “gdb-peda $disasmname of function” command to reveal invoked gadgets, memory address and binary code information of functions. Next, to employ “$ ROPgadget -- binary function name --only “pop |mov |ret” to discover all available gadgets with “pop” or “mov” or “ret” in the protected program, “$ objdump -x program name|grep -A1 ‘.data’ ” to find out “.data” and “.bss” memory address, etc. To create a malicious payload in Python based on the memory address of buffer overflow or heap overflow, gadgets, “.data” and “.bss”, etc. Finally, to utilize various ROP attack methodology in Table 1 to compromise these programs.

Stage 2: 87 out of 110 programs have been successfully exploited by ROP, authors install the proposed algorithm program to defence a vulnerable targeted program and exploit them with all ROP attack methodologies. Successful defence percentage earns 1 score if the author proposing the algorithm program can defeat one time cyberattack, otherwise, the proposed algorithm program fails, earns 0 scores. The database of the proposed algorithm program stores relative control flow data and the maximum turn-up number of gadgets for all benign routines, the security alarm will be triggered no matter the cyberattack derives from either the control flow integrity is breached, or the turn-up number of gadgets exceeds the stored maximum turn-up number of the gadget. The experimental result is shown in Table 1. In Table 1, most of the results for the authors proposing defence algorithm exceeds 90%, except Glue_two_or_More-Gadgets-bypass because lots of gadgets are very short, even gluing several gadgets, its length does not longer than the maximum gadget. Another is the Blind Return-Oriented Program attack, it is difficult to detect BROP as its concealment because attackers may deduce a single bit and scan the whole memory pattern via byte-by-byte. Attacks may have more opportunities to gain attack approaches and gadgets to circumvent detection. The most successful defence percentage is 99%, General ROP is easily defeated by frequent return instructions and frequent gadgets. It is not difficult to impede System_Call_Invoked without authority. Function-Oriented Programming will be detected if a dispatcher function gadget repeatedly operates. To detect Return-to-LIBC, we apply a recording procedure location to link a shared library and exit the shared library in a benign program. For ROP_Without_Ret, we monitor the return-like instruction whether they are deliberately to replace return instruction and bypass frequent return instructions detection. In order to detect Jump-Oriented Programming, not only do we utilize frequent return instructions and frequent gadgets, but also monitor the number turn up of gadget to impede dispatchers to chain and dispatch gadgets. It is easy to detect the malicious format string commands of String-Oriented Program (SOP), they have not been forged as benign codes. However, the detection may be bypassed if various evading techniques are applied. Counterfeit Object-Oriented Program (COOP) can be defeated by monitor repeatedly invoke virtual functions existing in applications on counterfeit C++ objects. To detect Signal Return-Oriented Programming (SOP), Global_Offset_Table_entry__alter, Setjmp_Buffer_Overwrite, Vtable Pointer Overwrite, and Function_Pointer_Overwrite, we focus on
monitoring the invoked whether they have authority. The average of successful defence percentage for various types of ROP and evasion is 96%, it shows that the authors’ defence algorithm is effective and high accuracy to defeat ROP attack.

Stage 3: authors launch ROP attacks against 87 vulnerable ROP programs in Stage 2 that are protected by each of the other ROP defence tools in Table 2. In Table 2, all results of the Successful defence percentage for existing defence tools exceed 90%. Even though CFIMon hardware-based defence tool is the lowest Successful defence percentage, the authors think it is still a very useful complement defence tool for software-based defence tools. Cyberattack vector is multiple approaches, so defence certainly needs multiple approaches. The authors’ penetration testing shows that ILR tool and the proposed algorithm are the highest successful defence percentage 96% without access source code. If a defence algorithm does not require to access source code, it has more advantages to be accepted by end-users, especially commerce security issues. Performance overhead (computational cost) is the increased extra computing execution time after a defence program is installed. Authors proposing the algorithm’s performance overhead (computational cost) is only 1%-2%, which is much lower than the ILR performance overhead, 13%-17%. Therefore, the authors proposing algorithm has better performance than other ROP defence tools for defence ability and performance overhead (computational cost).

The software computer languages of target include Javascript, C, C++, C#, .net, PHP, Python, and Ruby, etc. The tested software programs with ROP vulnerability include in Windows and in Linux. The defense performance will be excellent if source code or binary code is available. The proposed algorithm still works well if neither of the two is unavailable. The evaluation is decided by whether the targeted program that has been installed the authors’ defense countermeasure can detect and prevent Return-oriented programming attacks, e.g., escalate privilege to root, access and control a database, etc.

The advantages of the proposed algorithm: 1). Detection is effective and high accuracy because the proposed algorithm can impede most of the evasions for other ROP defence tools. The proposed performance overhead is lower than other defense countermeasures because it directly jumps to the data of each node of the Trie graph, it will not accumulate the time of checking routes as exponent explosive increment while the program grows, it is only linear workload. 2). It is transparent to most existing computer programs, does not require computer programs to be modified. It can be installed in most user-friendly tool kits. Transparency is an important factor in whether the product can be accepted by end-users without any modification to their system. 3). It does not require the information about either source code or binary, etc., side information, customized compiler support, or no binary rewriting.

The shortcomings of the proposed algorithm: 1). Authors proposing algorithm can be bypassed and the negative fault is not easy to be discovered in case of attackers have reconnoitered how it works. The number of chained gadgets does not exceed the threshold of turn-up number in benign execution or the control flow graph of the targeted program is deep enough for attackers to chain all malicious gadgets along with benign program route without deviating control-flow integrity. 2). The spawn shellcode or other crucial privilege escalation of system-calls is examined only.
Figure 1. The flowchart of ROP detection
### Table 1. The performance of the authors’ algorithm against various ROP and evasions

| Types of ROP                                           | Successful defense percentage |
|--------------------------------------------------------|------------------------------|
| General ROP                                            | 99%                          |
| Return-to-LIBC                                         | 98%                          |
| ROP_Without_Ret                                        | 95%                          |
| Glue_Two_Or_More_Gadgets_bypass                        | 20%                          |
| System_Call_Invoked without Authority                  | 99%                          |
| Jump-Oriented Programming (JOP)                         | 98%                          |
| String-Oriented Program (SOP)                          | 95%                          |
| Blind Return-Oriented Program (BROP)                    | 86%                          |
| Counterfeit Object-Oriented Program (COOP)              | 91%                          |
| Signal Return-Oriented Programming (SOP)                | 92%                          |
| Function-Oriented Programming (FOP)                     | 99%                          |
| Global_Offset_Table_entryAlter                         | 98%                          |
| Setjmp_Buffer_Overwrite and Vtable Pointer Overwrite and Function_Pointer_Overwrite | 97%                          |
| Successful average defence rate for various types of ROP and evasions | 96%                          |

### Table 2. Comparisons of the existing main ROP defence countermeasure tools

| Defence countermeasure tool                        | Access source code required | Successful defence percentage | Performance overhead |
|-----------------------------------------------------|----------------------------|--------------------------------|----------------------|
| kBouncer (Pappas, Polychronakis, & Keromytis, 2013) | No                         | 94%                           | 4%                   |
| ROPguard (Fratrić, 2012)                           | Yes                        | 92%                           | 11% - 13%            |
| ROP defender (Davi, Sadeghi, & Winandy, 2011)      | No                         | 91%                           | 3% - 5%              |
| Drop (Chen et al., 2009)                           | No                         | 93%                           | 10% - 15%            |
| Return-less Kernel (Li et al., 2010)               | Yes                        | 92%                           | 9% - 14%             |
| G-Free (Onarlioglu, et al., 2010)                  | Yes                        | 93%                           | 1.09%                |
| CFLocking (Bletsch, et al., 2011)                  | Yes                        | 94%                           | 0% - 5%              |
| Binary stirring (Wartell, Mohan, Hamlen, & Lin, 2012)| No                         | 91%                           | 1.4% - 1.6%          |
| ILR (Hiser, Nguyen-Tuong, Co, Hall, & Davidson, 2012)| No                         | 96%                           | 13% - 17%            |
| Compact Control Flow Integrity & Randomization (CCFIR) (Zhang et al., 2013) | No | 95% | 3.6% - 8.6% |
| ROPPecker                                           | No                         | 95%                           | 9.72%                |
| CFIMon hardware-based (Harrison & Allwein, 2018)   | No                         | 90%                           | 5.0% - 7.0%          |
| The authors’ defence algorithm                      | No                         | 96%                           | 1% - 2%              |
CONCLUSION AND FUTURE WORK

ROP attacks are the mainstream of cyberattacks now because ROP re-uses the existing codes inside the targeted program. ROP is more concealing and can even evade IDS/IPS. Even though many ROP countermeasure further improvements are needed to address the increasing sophistication and volume of ROP attacks, while also reducing the inherent performance overhead. We propose a novel ROP attack defence methodology in this paper that applies the Trie graph to present a protected program procedure. It is utilized to monitor control flow integrity and the turn-up number of gadgets for each node of the Trie graph to discover abnormally chained gadgets for ROP by comparing it with data of database in the benign program. The proposed algorithm employs multiple defence techniques to achieve high accuracy, better performance, and defence against a wider range of ROP attacks than other existing ROP defense methods. It is also transparent to most existing computer programs and does not require the protected computer program to be modified.

The limitation of authors proposing methodology: It must take some time to test a protected program to establish the relative Trie graph and the turn-up number of gadgets in each node and stored into a database in a static state before actual execution. If the authors proposing algorithm has been reconnoitered how it works by attackers, malicious gadgets do not exceed the threshold of turn-up number, and the control flow graph of the protected program is deep enough for attackers to chain all malicious gadgets along with benign program route, ROP bypasses the detection.

Authors will continuously improve the current version proposing the algorithm to better defensive performance against ROP and reduce performance overhead further by setting proper thresholds of gadget through more experiments and inserting gadget monitor in proper checkpoints to prevent that attackers chain gadgets into malicious codes along with benign control flow if the protected program is large. In the future, the proposed algorithm will be developed more suitable for smartphones against ROP attack with its characteristics being considered.
REFERENCES

Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., & Boneh, D. (2014). Hacking blind. In IEEE Symposium on Security and Privacy (pp. 227-242). IEEE.

Bletsch, T., Jiang, X., Freeh, V. W., & Liang, Z. (2011). Jump-oriented programming: a new class of code-reuse attack. ACM Symposium on Information, Computer, and Communications Security. doi:10.1145/1966913.1966919

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A. R., Shacham, H., & Winandy, M. (2010). Return-oriented programming without returns. In ACM Conference on Computer and Communications Security (pp. 559-572). ACM.

Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., & Xie, L. (2009). DROP: Detecting return-oriented programming malicious code. In International Conference on Information Systems Security (pp. 163-177). Springer. doi:10.1007/978-3-642-10772-6_13

Cheng, Y., Zhou, Z., Miao, Y., Ding, X., & Deng, R. H. (2014). ROPecker: A generic and practical approach for defending against ROP attack. NDSS Symposium 2014: Proceedings of the 21st Network and Distributed System Security Symposium, 1-14. doi:10.14722/nds.2014.23156

Davi, L., Sadeghi, A. R., & Winandy, M. (2011). ROPdefender: A detection tool to defend against return-oriented programming attacks. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (pp. 40-51). doi:10.1145/1966913.1966920

Ding, B., Wu, Y., He, Y., Tian, S., Guan, B., & Wu, G. (2012). Return-oriented programming attack on the Xen hypervisor. International Conference on Availability, Reliability, and Security (ARES).

Fratrić, I. (2012). ROPGuard: Runtime prevention of return-oriented programming attacks. Technical report. doi:10.1109/ARES.2012.16

Gao, Y. C., Zhou, A. M., & Liu, L. (2013). Data-execution prevention technology in the windows system. Information Security & Communications Privacy.

Guo, Y., Chen, L., & Shi, G. (2018). Function-Oriented Programming: A New Class of Code Reuse Attack in C Applications. IEEE Conference on Communications and Network Security (CNS). doi:10.1109/CNS.2018.8433189

Harrison, W. L., & Allwein, G. (2018, December). Language abstractions for hardware-based control-flow integrity monitoring. In 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig) (pp. 1-6). IEEE. doi:10.1109/RECONFIG.2018.8641707

Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., & Davidson, J. W. (2012). ILR: Where’d my gadgets go? In 2012 IEEE Symposium on Security and Privacy (pp. 571-585). IEEE. doi:10.1109/SP.2012.39

Hund, R., Holz, T., & Freiling, F. C. (2009). Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In USENIX Security Symposium (pp. 383-398). Academic Press.

Li, J., Wang, Z., Jiang, X., Grace, M., & Bahram, S. (2010). Defeating return-oriented rootkits with “return-less” kernels. Proceedings of the 5th European conference on Computer systems. doi:10.1145/1755913.1755934

Li, W., Li, M., Ma, Y., & Yang, Q. (2018). PMU-extended Hardware ROP Attack Detection. In 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID) (pp. 183-187). IEEE.

Mabon, R. (2016). Sigreturn oriented programming is a real threat. Informatik.

Maunero, N., Prinetto, P., & Roascio, G. (2019, September). CFI: Control Flow Integrity or Control Flow Interruption? In 2019 IEEE East-West Design & Test Symposium (EWDTS) (pp. 1-6). IEEE.

McClure, S., Scambray, J., & Kurtz, G. (2009). Hacking exposed: network security secrets and solutions. McGraw-Hill Companies, Inc.

Onarlioglu, B., & Lanz, B. (2010). G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries. Proceedings of the 26th Annual Computer Security Applications Conference. doi:10.1145/1920261.1920269

Pappas, V., Polychronakis, M., & Keromytis, A. D. (2013). Transparent (ROP) exploit mitigation using indirect branch tracing. In 22nd USENIX Security Symposium (USENIX Security 13) (pp. 447-462). Academic Press.
Payer, M., & Gross, T. R. (2013). String oriented programming: when ASLR is not enough. In Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (pp. 1-9). doi:10.1145/2430553.2430555

Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A. R., & Holz, T. (2015). Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In IEEE Symposium on Security and Privacy (pp. 745-762). doi:10.1109/SP.2015.51

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Proceedings of the 14th ACM conference on Computer and communications security (pp. 552-561). doi:10.1145/1315245.1315313

Shacham, H., Page, M., Pfaff, B., Goh, E. J., Modadugu, N., & Boneh, D. (2004). On the effectiveness of address-space randomization. In Proceedings of the 11th ACM conference on Computer and communications security (pp. 298-307). doi:10.1145/1030083.1030124

Wang, J., Xie, P., Wang, Y., & Rong, Z. (2018). A Survey of Return-Oriented Programming Attack, Defense, and Its Benign Use. In 2018 13th Asia Joint Conference on Information Security (AsiaJCIS) (pp. 83-88). IEEE.

Wartell, R., Mohan, V., Hamlen, K. W., & Lin, Z. (2012). Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 157-168). doi:10.1145/2382196.2382216

Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., & Zou, W. et al. (2013, May). Practical control flow integrity and randomization for binary executables. In 2013 IEEE Symposium on Security and Privacy (pp. 559-573). IEEE. doi:10.1109/SP.2013.44