Survey of Return-Oriented Programming Defense Mechanisms

Yefeng Ruan, Sivapriya Kalyanasundaram, Xukai Zou*

Department of Computer and Information Science, Indiana University-Purdue University Indianapolis, 723 W Michigan St, SL 280, Indianapolis, IN, USA.

ABSTRACT

A prominent software security violation – buffer overflow attack has taken various forms and poses serious threats until today. One such vulnerability is Return-Oriented Programming (ROP) attack. An ROP attack circumvents the Dynamic Execution Prevention (DEP) which is employed in modern operating systems to prevent execution of data segments, and attempts to execute unintended instructions by overwriting the stack exploiting the buffer overflow vulnerability. Numerous defense mechanisms have been proposed in the past few years to mitigate/prevent the attack – compile time methods that add checking logic to the program code before compilation, dynamic methods that monitor the control flow integrity during execution and randomization methods that aim at randomizing instruction locations. This paper discusses 1) these different static, dynamic and randomization techniques proposed recently and 2) compares the techniques based on their effectiveness and performances.

KEYWORDS

ROP; buffer overflow; control flow integrity; DEP

*__Correspondence__
Department of Computer and Information Science, Indiana University-Purdue University Indianapolis, 723 W Michigan St, SL 280, Indianapolis, IN, USA.
E-mail: xkzou@cs.iupui.edu

Received . . .

1. INTRODUCTION

The fundamental purpose of software attack is to divert the intended program flows and execute unintended instructions. The term software attack appears more generic that includes diverse types of applications ranging from Web and desktop applications to simple programs such as calculator performing addition of two numbers. A notorious form of software attack vulnerability is exploitation of memory. This comes in different forms like buffer overflow, heap overflow [1] and format string vulnerability [1]. Among them, buffer overflow vulnerability [2] is the most prominent one in which the attacker can overflow a buffer's boundary in the process' stack and overwrite the return address of a function with an arbitrary memory address such that he/she can execute any code in this overwritten address when the vulnerable function returns. Buffer overflow attack has attracted many researchers' attentions and numerous defense mechanisms have been proposed and employed to mitigate the attack. These defense methods prevent execution of code that is present in the data regions of the process, for example in stack or/and heap. Modern operating systems employ Data Execution Prevention (DEP) using the W $\oplus$ X security model, in which the memory location can be either be executable or writable but cannot be both. In such a way, the attacker cannot write code and execute it at the same address of stack or heap.

Despite the employment of such methods, attackers are still able to find ways to execute unintended code by making use of already existing code in the process' address space instead of injecting new code. A well known form of this

attack is the return-into-libc attack [3]. In a return-into-libc attack, the attacker overwrites the return address of a vulnerable function with the address of any function in the libc library. For example, the attacker could overwrite with the address of a 'system' function, by which a shell can be opened. A shell is a powerful backdoor which enables performing multiple operations. Return-Oriented Programming(ROP) is a specific form of the return-into-libc attack, in which the attacker tries to execute sections of code scattered across the process address space by linking them with indirect control transfer instructions, typically the 'ret' instruction. Numerous ROP exploits have been demonstrated recently that target at products like AdobeReader, AdobeFlash Player, Internet Explorer and so on. Since its appearance, ROP has being a very hot topic of research with various types of ROP attacks being proposed in parallel with some defense methods that prevent such linking of instruction sequences [4]. Among those proposed approaches, some of them are compile time approaches and some of them are dynamic.

The main contributions of this paper include:

- Briefly discuss about Return-Oriented Programming and some of the defense strategies adopted by defense mechanisms.
- Provides a high-level classification of recently proposed defense techniques and investigate some of them.
- Provides a comparative evaluation of the discussed techniques based on certain identified criteria - General Solution Correctness, ROP specific Effectiveness and Performance Metrics.
- Shows some important aspects to be considered towards developing an effective defense solution against ROP.

2. BACKGROUND

Return-Oriented Programming as mentioned before, is a type of return-into-libc attack [3]. As proposed by Shacham [5] [6], the attacker usually performs the attack in two steps:

1) In the first step, the attacker identifies sequence of instructions useful in performing his/her intended operations. This short sequence of instructions is called a 'gadget' and is typically between 2 and 5 instructions in length. However there is no limitation on the gadgets' length and it has not been proven that it is infeasible to have longer gadgets. Also the attacker can identify more than one gadget.

2) Then in the second step the attacker link these identified gadgets (from the first step) together in such a way that they are executed sequentially.

An overview of the ROP exploit is shown in Figure 1. The attacker first finds a vulnerable function in the process address space, either belonging to the application or included in external or internal libraries, such that the function would be executed during the actual program execution. He/she then overwrites the stack data with the intended return addresses (i.e the start addresses of the gadgets). The gadgets should end in indirect control transfer instruction, mostly the 'ret' instruction serves this purpose. Every time the actual operation (gadget) is executed, the 'ret' instruction at the end of the gadget pops the next value (which is the next gadget's address) in the stack, which becomes the new EIP. The attacker may also insert data values into the stack which can be used as arguments for the unintended instructions to be executed.

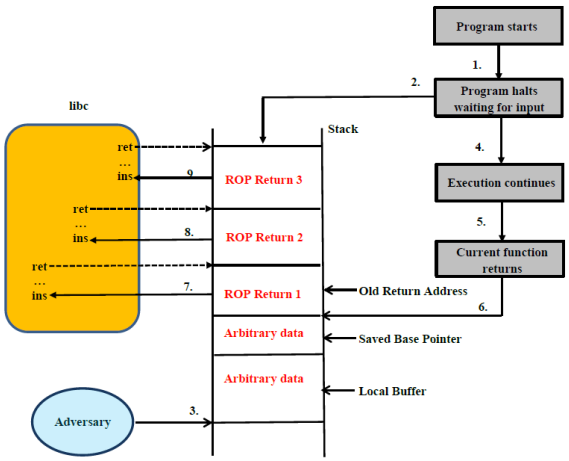

Figure 1. ROP Overview [7]

The technique which was first identified and explained above uses only the 'return' instruction for diverting the control flow of the program and transferring it to the unintended program's flow. However the 'return' instruction is not the only way of transferring control. There are other indirect control transfer instructions like the indirect 'jmp' and indirect 'call' instructions, in which the address of the target instructions are stored in the argument registers of these 'jmp/call' instructions. Therefore the attacker can use such instructions as well to divert the control flow and execute code that he/she intends to. This was demonstrated by Checkoway.et.al [8] in 2010 which proposed Return-Oriented Programming using return like instructions rather than directly using the return instruction. For example, the

; **00**:1–21 © John Wiley & Sons, Ltd.
DOI: 10.1002/
Prepared using

instruction sequence "$pop\ \%eax;\ \ jmp\ \%eax$;" is similar to a 'ret' instruction which pops the top of stack to EIP register and executes the instruction at the address contained in EIP. Following Checkoway.et.al's proposal, a similar technique called Jump Oriented Programming (JOP) was proposed by Bletsch.et.al [9]. JOP is very similar to ROP and it uses gadgets to perform operations which are chained together by a *dispatcher gadget* rather than the 'ret' instruction in ROP. The gadgets are linked through 'jmp' instructions and the *dispatcher gadget* takes care of executing them in sequence. The gadget sequence is stored in a dispatcher table in the process memory (overwritten by exploiting a memory vulnerability) and is independent of the stack. Therefore any ROP defense mechanism that depends only on the stack or the 'ret' instructions can detect only the original ROP attack and are vulnerable to the later proposed variations of the same.

Although a Turing-complete program can be formed using ROP, typically attackers use them targeting at only certain functions, such as 'system' functions, that spawn a shell or those functions which can change the memory settings of the process to undo certain permissions. Especially by changing the DEP setting, the attacker can directly overwrite the stack with the intended code and execute from the stack itself, instead of making a more difficult attempt of formulating a Return-Oriented Program attacks to perform the intended operations. A program binary may contain many gadgets which are helpful for the attacker. Schwartz.et,al in [10] demonstrated the existence of many potential ROP payloads (or gadgets) in more than 80% of programs that are larger than 20KB.

3. DEFENSE STRATEGIES

In this section we discusses certain criteria that many recently proposed approaches concentrate on for detection of ROP. Following is a list of proposed defense strategies:

a) Gadget Elimination: Prevent the formation of gadgets, i.e. sequence of instructions that end with an indirect control transfer instruction that could be potentially used by the attacker. Since gadgets are the backbone of ROP, eliminating them as much as possible would reduce the vulnerability space for exploitation by attackers. Some of the useful instructions that are commonly exploited by the attackers are instructions manipulating registers, storing values into the register (via pop instruction), restoring values from the register (via push instruction) and other similar instructions.

b) Control Flow Integrity (CFI): Prevent the program execution flow transferred from intended and legitimate return addresses. In [11], the authors first proposed a formal research work on how Control Flow Integrity can avoid exploitation of an application by different types of software attacks. Ensuring Control Flow Integrity based on the Control Flow Graph (CFG) of a program (graph depicting the possible paths of execution of the program) prevents any diversion from expected behavior of the program execution. In the case of ROP particularly, enforcing the integrity of control transfer instructions ensures prevention from gadget chain execution.

c) Unaligned Instructions: It is important to understand that program instructions could be aligned or unaligned. Aligned instructions are the original sequences of instructions as intended by the program and compiled by the compiler. Unaligned instructions are those instructions that can be formed from the original instruction sequence although they are not intended to be executed as such. Due to the fact that instructions in certain processor architectures can be of varying length and unaligned memory access is possible, instructions can be executed beginning from random memory addresses and these are the unaligned instructions. For example, consider the following set of instructions [7]:

$$B8\ 13\ \ 00\ \ 00\ \ 00 \quad mov\ \ \$0x13,\ \ \%eax$$
$$E9\ \ C3\ F8\ FF\ FF \quad jmp\ \ 3aae9$$

Since a gadget is a short sequence of instructions typically ending in the 'ret' (C3) instruction, the above instruction sequence contains a potential gadget as shown below which can be exploited by the attacker (for manipulating register value):

$$00\ \ 00 \quad add\ \ \%al,\ \ (\%eax)$$
$$00\ \ E9 \quad add\ \ \%ch,\ \ \%cl$$
$$C3 \quad ret$$

Therefore a complete defense mechanism that either eliminates gadgets or enforces control flow integrity should also be able to handle such unaligned instructions for complete coverage.

In addition to ROP's specific features, following are some of the general defense strategies observed in other defense approaches.

d) Deployment: The technique must facilitate easy deployment being compatible with different operating systems and processor architectures. It is also preferred that the deployment and the execution of the technique depends on minimal or even without side information like source

code, symbolic debugging information etc. since these may not be available with production binaries.

e) Performance: This is an integral feature of any software that primarily includes space overhead, runtime overhead and few other factors. The execution of the technique along with the actual program should not affect users' experiences. It may not be possible to completely eliminate such overheads, however it is expected that the impact is minimal.

4. CLASSIFICATION OF DEFENSE APPROACHES

Based on the trend observed in recently proposed defense mechanisms, these techniques can be briefly classified as shown in Figure 2. It is important to note that this is not the complete classification, rather it is only based on the techniques surveyed by this paper.

Randomization: Randomization includes a) Address Space Layout Randomization (ASLR) – This technique was the first kind of defense mechanisms proposed to defend against ROP. Address Space Layout Randomization [12] randomize the base addresses of parts of the process in the process address space like the stack, heap, external libraries etc., such that the attacker cannot predict the address where these would be loaded, which in turn means he/she cannot predict the addresses of his/her intended gadgets to overwrite the stack. ASLR was used in conjunction with DEP to protect against ROP. However since only the base addresses are re-located and not for each individual address within the components, if the base address can be found, then all other addresses relative to the base address can also be found. Roglia.et.al [13] proposed a method to find the base address of libc if the attacker can find the absolute address of any function in that library. If the base address of libc can be found, then it is easy to compute the addresses of any other functions in libc using the Global Offset Table (GOT). Also some parts of the process which are not compatible with this randomization are unrandomized and attackers can find useful gadgets in them. For these reasons, this technique has not been of researchers' interests in recent proposed defense mechanisms against ROP attacks.

b) Instruction Randomization – This approach primarily aims at randomizing a single or block of instruction(s), thereby affecting the attackers' entry point itself. The attacker needs to know the addresses of the instructions that he/she intends to use beforehand. The fundamental purpose of instruction randomization is to change the layout of the instructions as predicted by the attacker. This can be done by changing instructions' sequence order or replacing an instruction with semantically equivalent instruction, randomizing the addresses of the instruction [14]. Since the modification is done before execution, performance overhead is typically minimal in these approaches. However, in most cases one has to get access to the source code or debug symbols and disassemblers to make the modifications without affecting the control flow graph. Approaches have been proposed without the requirement of access to such side information, however they are not guaranteed to provide complete coverage [15]. Another disadvantage of Instruction Randomization is that position-dependent code cannot be randomized.

Compiler Based approaches: The compiler based solutions modify the code layout of the program at compile time itself. Typical defense strategies employed by these approaches are adding code that can check and validate the control flow integrity at the runtime, particularly by controlling the behavior of free branch instructions. Some of them also try to eliminate gadgets by rewriting instructions with other semantically equivalent instructions. It is apparent that source code is required, as the changes are done at compile time. For this reason, the deployment of these compile time approaches is difficult as it is not feasible to get access to the source code of production binaries.

Dynamic approaches: These approaches perform checking dynamically by monitoring the control flow integrity of the program. Some of the usual patterns observed and handled by dynamic approaches are checking for the source/destination of indirect control transfer instructions – 'ret', 'jmp' and 'call', saving and checking for matching 'call/ret' addresses, checking for the frequency of 'ret' instructions etc. Since the method is dynamic, most of them do not require access to source code/debug symbols, however some approaches are involved in pre-processing stage, where the compiled binary is examined to identify potential gadgets that could be used during runtime for comparison. Also there is a significant and noticeable performance overhead in these approaches owing to runtime monitoring. Few subtypes identified in Dynamic approaches are:

a) Binary instrumentation: ROP checking logic is instrumented to the program during execution using a binary instrumentation framework.

b) Hardware-facilitated: Registers available as part of the processor are used by the ROP checking logic, which is implemented as a kernel module.

; **00**:1–21 © John Wiley & Sons, Ltd.
DOI: 10.1002/
Prepared using

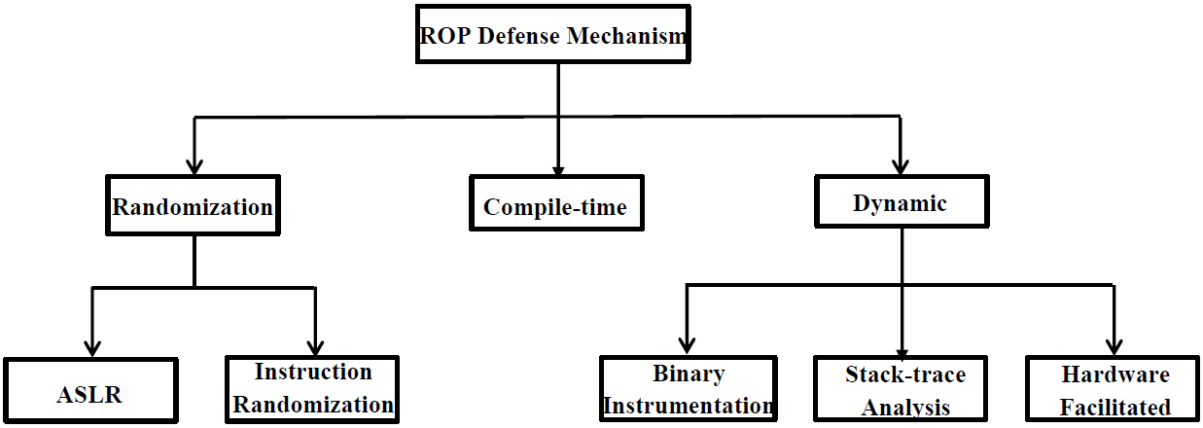

Figure 2. ROP Defense Techniques Classification

c) Stack-trace analysis: The stack trace is analyzed based on certain heuristics and checked for potential ROP exploits.

A brief comparison of the different approach types is given in Table I.

5. PROPOSED DEFENSE TECHNIQUES

In this section we briefly discusses some of the recent proposed approaches under each type (as shown in Figure 2) to detect and mitigate ROP attacks.

5.1. Randomization

5.1.1. In-place code randomization

Pappas.et.al [16] proposed a compile time approach that performs code transformation i.e transform one (or more) instruction(s) to another set of instrunction(s) such that they are semantically equivalent. This transformation is done to either eliminate formation of gadgets or modify them such that the intended (gadget) operation may not be performed completely. This method is designed to work even when there is no symbolic debugging information available with the binary. Disassemblers that would work without debug symbols can be used to analyze the instructions, however they cannot provide 100% disassembly coverage. The authors of [16] used IDAPro disassembler for their prototype implementation.

There are basically two types of modifications of instructions proposed:

- Atomic substitution
- Instruction re-ordering
  - Intra basic block re-ordering
  - Re-ordering of Register Preservation code
  - Register reassignment

Atomic substitution: Instructions that may be involved in formation of gadgets can be substituted with different instructions such that the ‘C3’ byte which can also be executed as a ‘ret’ instruction is removed, while maintaining the semantics of the instruction equivalent.

As shown in Figure 3 there is a gadget formatted from unaligned instructions before transformation. The corresponding instruction can be transformed in to an equivalent form to remove the ‘C3’ byte in the instruction sequence.

Intra basic block re-ordering: Independent instructions within a basic block can be re-ordered such that unaligned gadgets can be eliminated. The dependence graph, which is a graph depicting relationship between the instructions of the basic blocks, can be used to identify independent instructions. An example of re-ordering is shown in Figure 4 and its corresponding dependence graph is shown in Figure 5.

Re-ordering Register Preservation code: Functions sometimes save register values before performing the operations and restore them at the end. These values are usually saved in the stack itself by a series of push and pop instructions. Reordering of the push/pop instructions may modify the expected behavior of gadgets intended to happen by the attacker. This approach modifying the register saving instructions is adopted since instructions with pop/ret pattern are popular in ROP gadgets.

Register re-assignment: Live region of a register is the region between where the register is defined and last used. The approach proposes to re-assign registers within the same boundaries of live regions, such that only the names of the registers are exchanged and the data at the corresponding

Table I. Comparison of ROP defense approaches

Approach	Strength	Weakness
Compiler-based	• Runtime efficiency • Employs information transformation in addition to gadget protection • No binary re-writing	• Need source code • Comparatively deployment is not easier • Recompilation required when program changes
Randomization	• Prevents the attackers' entry point itself • Runtime overhead better than dynamic approaches, but higher than compiler-based approaches • Easy transformation of most instructions	• Need source code or debug symbols for some techniques • Position-dependent code cannot be randomized • Space overhead higher in some cases • Chances of error, due to incorrect randomization
Dynamic approaches	• No source code or debug symbols • Minimal or nil binary re-writing • Does not require re-compilation if program changes	• Often higher runtime overhead • Often dependent on memory limits or consider only few parts of application code • Significant Space overhead in some cases

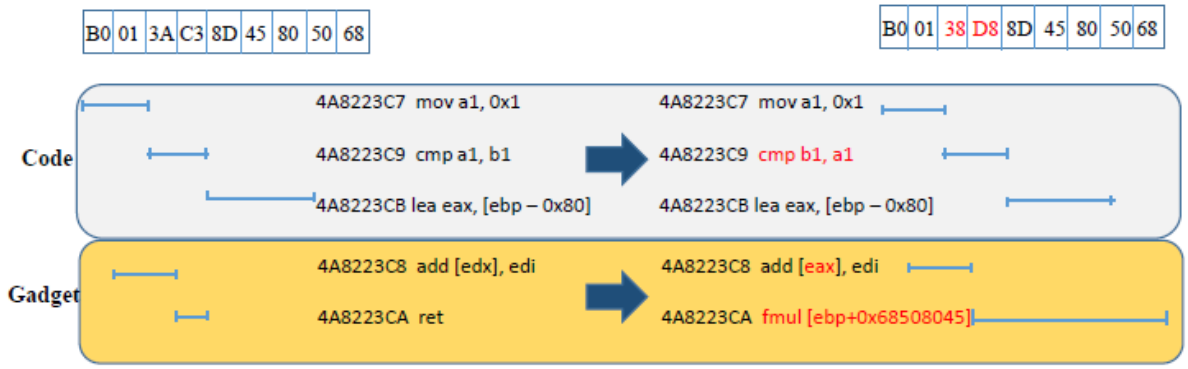

Figure 3. Atomic substitution [16]

Figure 4. Intra basic-block re-ordering [16]

; **00**:1–21 © John Wiley & Sons, Ltd.
DOI: 10.1002/
Prepared using

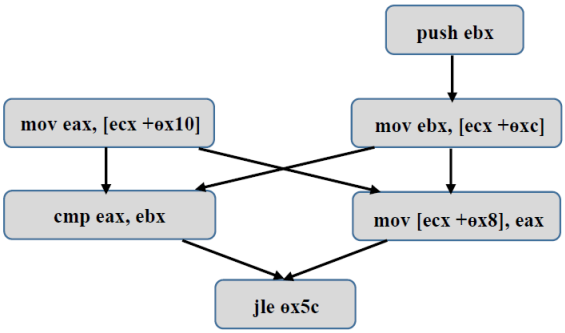

Figure 5. Dependence graph of instructions in Figure 4 [16]

memory locations are not affected. This re-assignment could possibly affect the intended gadget behavior.

This type of randomization has a significant advantage of no dependence on side information, i.e symbolic debugging information. However this is offered at the expense of the inability of complete coverage. Only around 80% partial coverage is obtained on average with 10% being complete elimination of gadgets and the remaining being modification of the gadget behavior. Regarding the entropy, 80% of gadgets seem to have four or more possible ways of randomization and 12% of gadgets have two possible ways.

5.1.2. Instruction Location Randomization

Hiser et.al, [17] proposed a randomization approach to randomize every instruction in the program, to thwart any ROP attack based on the assumption of instruction addresses. A ROP attack is typically carried out by populating addresses in the stack, in which the attacker needs to be aware of the addresses of the instructions in advance. Therefore randomizing the addresses would prevent the attacker's intended behavior. The method consists of two stages:

1) Offline Analysis: The program binary is analyzed for every byte and all possible instructions are identified and inserted into the database. This includes the original intended instructions and the unaligned instructions as well. This method makes no attempt to classify the instructions as intended or unintended. Offline Analysis includes analysis of indirect branch targets and call sites as well. The results of the offline analysis is a set of re-write rules that contain the actual locations of the instructions and the redirection information for the randomized instructions. Therefore the instructions are scattered apart and the expected sequence information is maintained by the re-write rules. However certain addresses cannot be randomized directly since they may be relative to the return addresses or other locations. In such cases, the original instruction is left unchanged, however the branch target contains a rewrite rule which points to a randomized address.

2) Running the ILR protected program: The ILR protected program is run using a per-process virtual machine(VM) like Strata in this case. The virtual machine, before fetching and executing the instructions, applies the translations made by the offline analysis dissembler. It executes an instruction and then to execute the next instruction, looks up at the re-write rules, identifies the location of the next instruction and executes it fetching from that location.

Most of the binary is covered, except for indirect branch targets that are not identified by the disassembler and position-dependent code. Also the external libraries included in program are not randomized by this approach. With the prototype implementation, the average overhead is from 13-16%, almost half of it due to the use of the per-process in VM.

5.1.3. Marlin

Gupta.et.al, [18] proposed yet another randomization approach similar to ILR, but instead of randomizing every instruction, this technique randomizes every function block of the program. The rationale, as claimed by the authors, behind choosing to randomize function blocks is to draw a fine line in the granularity achieved between the higher performance overhead incurred due to randomizing every instruction/every basic block or randomizing only the program segments (like ASLR). A significant advantage of this method is, the randomization is done by a modified loader at load-time. Therefore every execution of the program results in different randomizations, making it difficult for the attacker to predict the code layout. A brief discussion about the steps involved in Marlin is listed below:

1) Pre-processing step: Since this technique randomizes function blocks, it is essential to know the addresses of these blocks in the program binary. For this reason, debug symbols are required. However since many application binaries are not provided with debug symbols, external tools to unstrip the application are used.

2) Randomization: The randomization algorithm generates a random permutation and the function blocks are shuffled according to this permutation. Since there might be 'jmp' instructions within the function blocks and they might be dependent on relative addresses, in order to not affect their behaviors, a *jump table* is maintained during the execution of the algorithm, which maintains the old and new addresses of

the functions. After the shuffling is done, based on the *jump table*, the jump offsets are modified according to the new addresses. This step is called *jump patching*. After that, the *jump table* is deleted, to prevent the attacker from knowing the new location of the instructions.

However, not every function can be randomized, since for some functions if they are changed from their original locations they may be rendered useless or even result in some exceptions, a typical example is the _start function. Therefore only those position-independent functions can be randomized.

5.2. Compiler-based approaches

5.2.1. Return-less kernels

Li.et.al, [19] proposed a compiler based approach to defeat return-oriented rootkits. Return-oriented rootkits as explained and demonstrated by Hund.et.al, [20], is a malicious Return-Oriented Program that can attack the kernel code without exposing the fact that the kernel has been attacked. For example, a 'ps' or 'ls' would not list the malicious or unintended processes in running. The approach proposed to prevent this return-oriented rootkits (ROR) is through return indirection. Instead of directly saving the return addresses in the stack, a return index is stored during the 'call' instruction. This return index points to a return address entry in a centralized return address table, looking up which the actual return address can be retrieved at runtime. Since this return address table contains the return addresses of functions, this can be generated offline. Therefore only valid return indices (i.e legitimate 'ret' instructions that follow 'call' instructions) contain entries in the return address table and random gadgets' addresses populated by the attacker in the stack would not be the valid indices and hence the attack fails.

The method therefore requires compiler support such that it can perform this translation as depicted in the Figure 6 - for every 'call' instruction, the return index corresponding to the return address is pushed into the stack and for the corresponding 'ret' instruction, instead of popping a return address, the return index is used to compute the return address and the control is transferred to that address.

The authors also proposed to remove unintended return opcodes (C3) from the instructions by two techniques :

a) Register re-allocation : If the usage of certain registers' machine code contains the return opcode, rewrite the instruction to replace with another register whose machine code does not contain the 'C3' opcode.

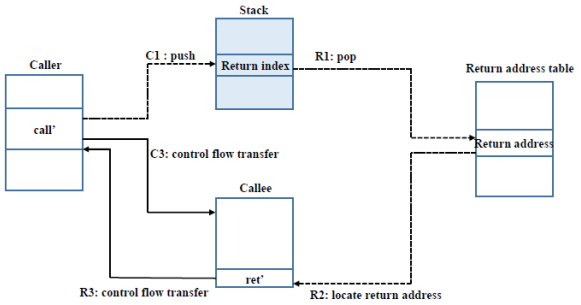

Figure 6. Return-less kernels [19]

b) Peephole optimization : Similar to register re-allocation, if the operands contain the return opcode, replace them with equivalent instructions without changing the semantics but removing the return opcode.

Since every legitimate 'ret' instruction should be preceded by a 'call' instruction, all entries in the centralized table are valid return addresses and hence there is no probable reason for any false alarm.

5.2.2. G-Free

This technique proposed by Onarlioglu.et.al, [21] is the first compiler-based approach proposed to defend against both 'ret' based gadgets and other indirect control transfer like 'jmp/call' based gadgets as well. Similar to the return-less kernel approach, this approach also aims to prevent gadgets from unaligned instructions and protect branches formed from aligned legitimate instructions. The authors' experiments reveal that there are around 18K free branch instructions present in the libc library. This means, the attacker has many ways of forming gadget sequences to perform his/her intended behaviors. Gadgets/free branch instructions in aligned instructions are protected by adding code at the prologue/epilogue of the function calls, that check for legitimate execution of the function block. The functionality performed by the code is – when a function is called i.e. at the function 'call' instruction, it encrypts the return address and pushes it into the stack with a *random key*, and when the function returns i.e at the 'ret' instruction, it decrypts the saved return address with the same random key to compute the original value. If an attacker pushes an arbitrary return address into the stack, when the 'ret' instruction is executed and the decryption results in an invalid value such that the attack is detected. The authors, however, have not given any details about how a computed address is checked for validity/invalidity. Similarly for other

; **00**:1–21 © John Wiley & Sons, Ltd.
DOI: 10.1002/
Prepared using

free branch instructions like 'jmp', a technique called frame cookie is used. When a function/block containing a 'jmp' instruction is executed a frame cookie is computed using the same *random key* and pushes it into the stack. Then just before the free branch instruction, validation is performed to check for the validity of the cookie. If there is no cookie pushed or the computed value does not match with the stored cookie, it is assumed to be an attack. The authors also propose to add alignment sleds (nop instructions) before these code blocks, so that they cannot be executed in an unaligned fashion too.

For the prevention of gadgets formation in unaligned instructions, the occurrence of the corresponding instruction bytes needs to be avoided. For example, the bytes, 'C3, C2, CA, CB' for 'ret' instructions and 'FF' for 'jmp' instruction. The authors proposed different kinds of instruction sequence re-writing techniques to achieve the elimination. They are Register re-allocation (similar to Return-less kernel technique) – replaces register operands to remove the 'C3' byte, Instruction transformation – replaces an instruction that has a free branch byte with an equivalent instruction, Offset Adjustments – adjust or modify the offset of the operands of the 'jmp' instruction if a 'ret' instruction byte is encountered. This offset adjustment can be done by adding nop instructions accordingly. This way, the authors claimed that it is possible to eliminate many gadgets formation.

5.2.3. Control Flow locking

Bletsch.et.al, [22] proposed another compiler time technique to detect Return-Oriented Programming attacks. As discussed in Section 3, ensuring the CFI of the program prevents ROP from happening. The CFL approach targets at the same purpose – ensuring the program execution follows the pre-determined control flow. To achieve this, a mechanism similar to mutex locking is used excluding the waiting/atomicity. A lock is acquired at the entry point and unlocked at the exit point. Acquiring/releasing lock is achieved by updating a variable in memory. The technique, as with other compiler-based techniques, proposes two main ideas to protect aligned and unaligned instruction sequences:

- Remove unintended code : Unintended code can be removed by adding no-op bytes to instructions so that every instruction is aligned to a n-byte boundary. This also means that every target of indirect control flow transfer is also aligned to n-byte boundary.
- Protect intended code : To protect intended code, all indirect transfer control locations are identified from the control flow graph (CFG), such as the 'jmp/call' instructions, call sites, 'ret' instructions etc. Lock/unlock code is then added to these identified locations appropriately such that acquiring 'lock' sets the value of the *control flow key* 'k' to a particular value and 'unlock' validates to check if the 'k' value matches with the *control flow key* value of that particular control path. The authors chose to use unique values for different control paths, rather than a single bit same value for all control paths, since setting 'k' to a single bit value ensures the transfer passed through *any* valid entry point. This can be misused by the attacker to evade the protection mechanism. Therefore to avoid that, the authors chose to use unique values for every control flow path. However, since direct calls cannot be modified by the attacker, they do not require the locking mechanism. An overview of the proposed control flow lock checking at different locations is shown in Figure 7.

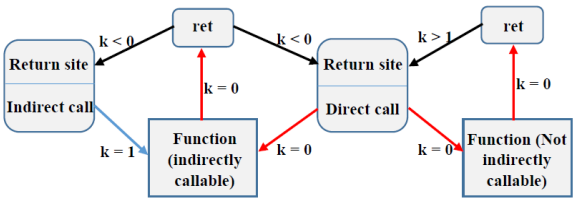

Figure 7. Control-FLow Locking [22]

The protection mechanism is implemented in two phases before execution:

1) Pre-assembly phase: An assembly re-writer removes possibility of unintended code by aligning instructions in the specified n-byte boundary. It then inserts the lock/unlock code before indirect control transfers, with dummy values for 'k' since the complete control flow graph is not yet known.

2) Post-link phase: The assembly re-writer can now extract the CFG and compute the actual 'k' values and update the same in the lock/unlock accordingly.

Since the control flow locking ensures proper value of 'k' of a particular control flow path via the lock/unlock, the technique cannot report false behavior unless the value of 'k' is compromised or the extracted CFG is incomplete or wrong. The correctness also depends on the correct computation/checking of the 'k' values at each control transfer locations.

5.3. Dynamic Approaches

5.3.1. ROPDefender

Davi et.al, [7] proposed a binary instrumentation method to detect ROP attacks based on 'ret' address comparison. The method works by maintaining a shadow stack to keep track of expected return addresses and comparing the actual return addresses with the expected addresses (Figure 8). PIN, a dynamic instrumentation framework is used to implement the method.

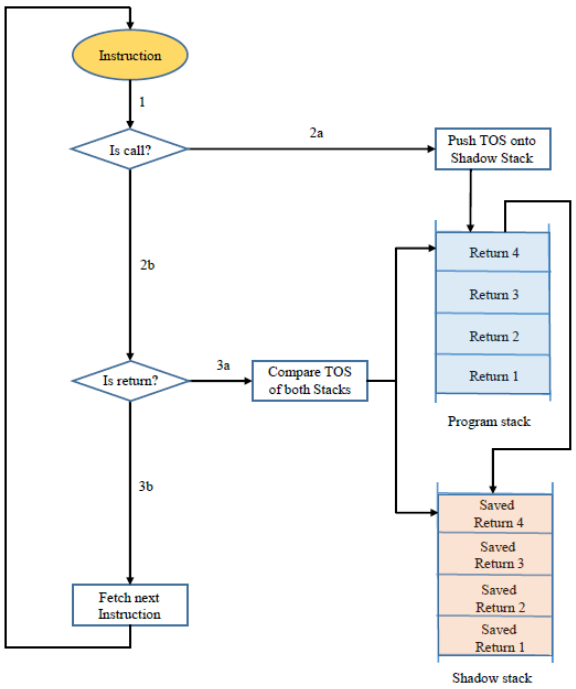

Figure 8. High level approach of ROPDefender [7]

Instrumentation using PINtool: Instrumentation routine checks if the given instruction is 'call' or 'ret' and analysis the routine. For the 'call' instruction, it checks that if it pushes the corresponding return address into the shadow stack. And in a 'ret' instruction checking, the actual return address is compared with the expected address in the stack. These two types of checking are implemented within the PINtool API [23] [24]. When the process which needs to be monitored is executed in the PIN environment, the PIN tool examines the first/last instruction of basic blocks and at the same time appropriate analysis routines are instrumented using the JIT compiler of the PIN framework. For every 'call' instruction the return address is pushed into the shadow stack and for every 'ret' instruction the address at the top of the stack is popped and compared with the actual return address popped from the actual program stack. A mismatch in the return address implies a ROP attack and that process will be terminated. Though earlier proposed approaches take a similar approach, this method handles some of the unusual scenarios like exception and signal handling.

This tool has several advantages over the previous approaches. It handles the following exceptional cases which are not taken into account in the previous approaches:

a) Setjmp/Longjmp: When system calls like 'setjmp/longjmp' are executed, stack frames are bypassed and this may lead to mismatch in the return addresses and then raises false alarms. To handle such type of cases, ROPDefender pops the return addresses continuously from the stack until there is match or the stack is empty. The former case is due to 'longjmp' and the latter implies a ROP attack.

b) Unix Signals/Lazy binding: When signal handler is invoked, there is no call instruction that gets invoked. To prevent false positive, ROPDefender uses signal detector API and pushes the appropriate return address into stack. Similarly when lazy loading happens the return address may not be available until the call instruction is actually executed. Again ROPDefender pushes the computed return address before the 'ret' is actually invoked.

c) C++ Exceptions: In C++ when the function does not handle any exceptions, stack unwinding will happen until a function that handles exception is found in program. ROPDefender pushes the computed return address (to be executed after the exception handling completes) by the library functions during the stack unwinding process.

5.3.2. Control Flow Monitoring

Chen.et.al, [25] proposed the first dynamic method to detect ROP attacks based on different types of gadgets according to three instructions – 'ret', 'call' and 'jmp'. The method depends on ensuring programs' control flow integrity rather than the type of instructions. The authors discussed about monitoring the control flow in three scenarios:

a) 'call' instruction: When a 'call' instruction is executed, the control should flow to an instruction that is the prologue of the function. This could be either a '$sub\ esp,\ value$' instruction for a frame function or '$push\ ebp,\ mov\ ebp,\ esp$' instruction sequence for a non-frame function. Therefore after a 'call', the next instruction is checked against either of these instructions.

b) 'ret' instruction: A 'ret' instruction usually transfers control to the next instruction after the caller, by popping the

; **00**:1–21 © John Wiley & Sons, Ltd.
DOI: 10.1002/
Prepared using

saved address from the stack. Therefore to maintain the 'ret' integrity, the authors proposed to maintain a shadow stack mechanism similar to the ROPDefender approach [7], in which after each 'ret' instruction, the popped return address from the program stack is checked against the saved return address in the shadow stack.

c) 'jmp' instruction: The authors say that the 'jmp' instructions are typically used to jump between locations within the same function and hence propose to deem any jump between different functions/libraries as an ROP attack.

This technique is typically implemented using the PIN instrumentation framework [23] for runtime monitoring. Also for static and dynamic analysis of the libraries and resolution of addresses which is used during the runtime monitoring, the technique uses the IDAPro disassembler and ReadELF.

5.3.3. kBouncer

Pappas.et.al [26] who proposed a binary rewriting method earlier [16], also proposed a dynamic method to monitor the control flow integrity of the program by making use of the Last Branch Record (LBR) feature provided by the Intel(R) processors. LBR is a cache like facility provided by the processor, which includes a set of model-specific registers that stores the source and destination address of the branches in branch/target register pairs. Since a typical ROP exploit always tries to gain control of the victim program, it is usually done via system calls which are wrapped in wrapper functions in Unix-like systems and in API calls in Windows. Therefore instead of checking the LBR stack for every indirect branch instruction, it can be checked only during such API calls that perform system-level operations. The authors have chosen to enforce the LBR stack check at the time the API is invoked itself, since in more than 50% of the API calls (for system call invocation), the number of legitimate branches between the API call and the actual system call exceeds the total limit on the number of the LBR registers. Therefore it is best to check at the entry point itself i.e API call itself to check if the API call was legitimate or not.

The authors proposed to identify illegitimate operations by using two key attributes:

- Illegal 'ret' instructions are not preceded by call sites. Any 'ret' instruction should transfer control to an instruction which is located after the corresponding call site of the caller function. This is the usual flow in a normal legitimate function execution. Since ROP gadgets transfer control from 'ret' to arbitrary locations in the functions, illegal 'ret' instructions can be detected.
- Since ROP code consists of many gadgets chained together, this attribute aims to analyze the LBR stack for the recent instructions and check if a particular target address is followed by uninterrupted instruction sequence (of maximum length 20) ending at the branch address of the next level. If there exists a sequence like this, it is assumed to be a gadget and when a sequence of 8 or more gadgets exists, it is assumed to be ROP exploit.

The kBouncer – which is the prototype developed by the authors of [26] to demonstrate the proposed branch tracing technique consists of three components:

a) Gadget Extraction component : This component examines the executable application and identifies gadgets (intended and unintended) by taking into account different types of indirect control transfers including 'ret', 'jmp', 'call' instructions. The output of the process is stored in two hash tables, one for the call-preceded gadgets (i.e gadgets where 'ret' instructions transfer the control to instructions that are preceded by a 'call' instruction) and the other to store all other kind of gadgets.

b) Interposition component: This component is implemented on a Detours framework – a framework to intercept function calls, to call the kernel module to enable LBR feature. It also sends messages to the kernel module to analyze the LBR stack whenever a sensitive API is called.

c) Kernel Component : The kernel component does the actual check on the LBR stack based on the previously described key attributes. For every 'ret' in the LBR stack, starting from the last but one register pair (ignoring the last pair, since it actually denotes the current API call), the kernel module checks if there is a 'call' instruction that precedes the target. The kernel module also checks for every target address (to also account for branches due to 'call/jmp' instructions), whether it is a part of any gadget by looking for the address in the stored hash tables and identifies those potential gadgets and hence the corresponding illegitimate executions.

Since the technique uses hardware-facilitated registers and the checks are performed only for selective sensitive API calls, the performance overhead is minimal. The authors claimed a worst time overhead of only upto 4% in their

test on Wine test suite, which calls windows API functions extensively.

5.3.4. ROPecker

Cheng.et.al [27] proposed a sliding window oriented dynamic method to detect ROP attack based on the past and future execution flow of the program. The method makes use of the LBR registers available in the hardware similar to the kBouncer technique. The method consists of two stages: offline pre-processing and runtime detection of gadgets.

a) Offline pre-processing: The application binary is analyzed for potential gadgets and the details are stored in a instruction database. The pre-processor disassembles and analyzes the first six instructions starting from each byte of the code segment in the binary and stores information about the instruction sequence for every byte in the code segment. If the instruction sequence ends in a indirect branch, the tool considers the sequence as a potential gadget and stores information accordingly in the database.

b) Runtime Detection: Initially all pages of the code segment are marked as non-executable (by setting the No eXecute bit) and a sliding window is maintained at all times. Beginning from the first page, for every new page accessed an exception is raised and ROP checking logic is triggered. If the gadget chain formed so far is above the threshold, the program is terminated. Else the page is set as executable by ROPecker and the window is slided to this new page. The same process is repeated throughout the program execution, by raising exceptions whenever code outside the sliding window is accessed. Although the code segments are protected by the sliding window mechanism, some system calls like 'mmap' and 'mprotect' which sets execution permissions can disable the DEP protection such that no exception will be raised when the code outside the sliding window is executed.

The runtime detection is therefore triggered under two scenarios – when the code outside sliding window is executed and risky system calls like 'mprotect' and 'mmap' are executed. The runtime detection is implemented as a kernel module (in Figure 9) which checks the previous and future execution flow of the program for potential gadgets. For the past execution, the module checks the recent branches in the LBR registers if the source is an indirect branch instruction and destination is a gadget start address (using the instruction database). If a gadget chain of such gadgets is found (with a minimum gadget chain length), then it is assumed as a ROP attack and the program is terminated. However LBR registers may not only contain branch instructions of the same program. In such cases, the formed gadget chain may be less than the threshold value. For efficient detection, the authors proposed to also check the future execution flow, by examining the next instructions in the stack for potential gadget addresses and comparing them with the addresses stored in the database. For 'jmp'-based targets, instruction emulation is performed since the targets are determined dynamically and the database may not contain that information.

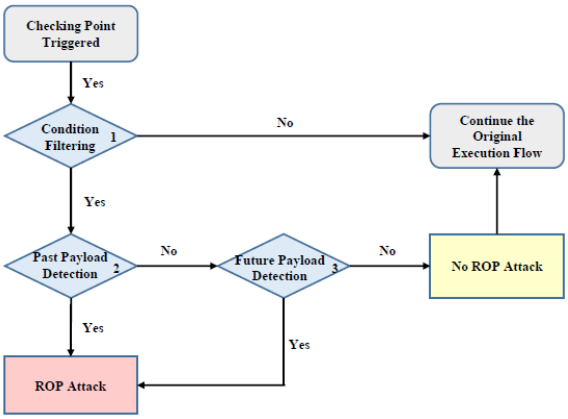

Figure 9. Overview of ROPecker [27]

The method although has a minimal performance overhead, there are some scenarios when the method may not be effective and efficient. The tool works on information from a database which is formed assuming gadgets of six instructions in length. In case that the adversary can form gadgets less or more than six, and insert them in between regular gadgets, the gadget chain length may never reach the threshold and the tool would not work. Also when the gadgets fall within the sliding window, they may be executed without any exceptions raised.

5.3.5. ROPGuard

This is yet another dynamic method proposed by Ivan [28] which uses a technique of monitoring the execution flow during the program execution. The execution flow, instead of being checked for every instruction, is checked when certain 'critical functions' are executed. Critical functions as assumed by the author, are functions that try to change memory permission settings or create new processes from the current process that may not be protected by DEP. The author suggested the following series of checks whenever such critical functions (details or specifics of the critical function

; **00**:1–21 © John Wiley & Sons, Ltd.
DOI: 10.1002/
Prepared using

are stored in a configuration file which the ROPGuard uses at runtime) are called to ensure ROP detection.

- Check the stack pointer: Since the stack needs to be modified with required return addresses, the attacker may sometimes modify the stack pointer itself to point it to a different memory region than the original program stack. Therefore when a critical function is called, the stack pointer has to be checked whether or not it is within the stack boundaries of that particular thread or process.
- Check for the address of critical function on stack: If the critical function is not called via the 'call' instruction (which is the expected behavior), then the address of the critical function should be present in the stack as a return address.
- Check for the return address from the critical function: Check if the instruction preceding the target of the return address from the critical function is a 'call' instruction and the address that the 'call' instruction transfers control to is the critical function itself. If this is not the case, it could be a potential target.
- Check the stack frames: If EBP is used as the stack frame pointer, check if the EBP points to a location outside the stack region, and if it does it means a corruption and possible ROP exploitation.
- Simulate the execution flow: If the EBP is not used as the stack frame pointer, then the return target after the critical function returns may not be directly available. In such a case, the instructions after the 'ret' from the critical function are simulated to check the stack behavior. If the 'ret' instruction directs to an instruction that is not preceded by a 'call' instruction, it can be an ROP attack.
- Function specific check: Apart from the above mentioned checks, some checks could be specific to the function. The author claimed that currently ROPGuard provides function specific checks for the functions – VirtualProtect and LoadLibrary, by preventing their execution.

6. COMPARISON OF APPROACHES

In this section we provide comparative evaluations of the detection techniques discussed in the previous sections. Tables listed here compare the techniques on different dimensions – general evaluation (Table II), ROP specific Effectiveness (Table III) and Performance metrics (Table IV).

6.1. General Evaluation

In this part we focus on the general aspects of the defense techniques like:

- Type – Type of the approach, is it executed or included at compile-time/run-time or use a randomization technique?
- Coverage – How much percentage or parts of the application does the corresponding technique cover?
- Correctness – With the proposed implementation logic, does the technique always provide correct results or incorrect results like false positive, false negative and true negative ?

6.1.1. Coverage

Considering the coverage achieved, almost every compile-time technique achieves complete coverage, owing to the availability of the source code. Dynamic approaches also typically has full coverage with few exceptions like kBouncer [26] and ROPGuard [28] that focus only on certain API calls that are presumably the usual targets of ROP exploits. Also the coverage may not be complete for those techniques that depend on disassemblers (without debug symbols). The accuracy and completeness of the coverage of these techniques is proportional to the efficiency of the latter in recovering the instructions. There is a case in IPR [26] that only achieve 80% coverage on average. The authors claimed that this is sufficient to randomize the popular gadgets identified in today's ROP exploits. However this claim is based on their existing experiments and may not hold for other or future exploits. On the other hand, even with efficient disassembly some randomization approaches cannot achieve complete coverage. Like in the case of Marlin [18], some functions if changed from their original location may be rendered useless/result in exception, for example, the _start function. Hence only those position-independent functions are randomized. Similarly ILR [17] also depends on disassemblers and position-independent code for randomization. Therefore the amount of position-dependent code is crucial in randomization approaches for achieving complete coverage.

6.1.2. Correctness

Considering the correctness of the approaches, though most of them are claimed to be false-positive proof, there

could be certain conditions in which they would result in false-positive, false-negative or true-negative cases. Some of these potential conditions are shown in Table II. For example, in those techniques that depend on the accuracy of the output of pre-processing stages (offline stages), there is a chance for incorrect results. For example, consider the cases of CFL [22], ROPecker[27] and ILR [17]. In the cases of CFL which entirely depends on the Control Flow Graph of the program, it will result in incorrect results if the CFG is not computed properly. Another insignificant cause of error is the improper computation of the 'k' value used in lock/unlock verification. In ROPecker, the runtime ROP checking logic depends on the gadget database extracted offline prior to the execution based on the results from a disassembler. Therefore the correctness of the solution considerably depends on the correctness of the disassembler used. Secondly, every instruction sequence ending in an indirect control transfer is termed as a potential gadget. Though the technique checks for chain of gadgets rather than for one or two gadgets before concluding an ROP attack, there is still possibility that execution of some legitimate instruction sequences ending in indirect control transfer can be termed as ROP. Another scenario that would result in false negative is when the ROP gadget chain is less than the pre-defined threshold value. Another approach that can result in false-positive is ILR, which depends on its offline indirect branch analysis routines for randomizing indirect control transfer instructions.

In some cases, the possibility of incorrect behavior is due to the limitations in the execution setup. For example, in the case of kBouncer, the ROP checking logic only checks for the authenticity of the branches in the LBR registers. If the number of LBR registers is 16, and the attacker exploits an API that has the previous 16 branches to be valid, but is still not an intended function or contains indirect branches after these 16 instructions, the tool cannot identify such attack and results in false negative since the checking logic is triggered only at the beginning of the API. Though no such instances have been reported yet, it is not infeasible. Also regarding the gadget chaining check, though current tested applications did not report legitimate instruction sequences higher than the threshold, there is no guarantee it will be the case always. In such cases, legitimate code would be termed as ROP, being a false-positive. Therefore it is essential that the defense technique caters to such exceptions by including additional checking logic to protect its correctness.

Another potential cause of false-positive is the ROP checking logic itself. It is the case in Control Flow Monitoring [25] which terms any 'jmp' across functions/libraries as ROP, since the technique assumes jumps typically happen only within same functions. This is not the case always, there could be legitimate jump outside of the functions usually as a result of some compiler optimizations.

Let alone the ROP detection, normal program execution itself can be affected sometimes due to the methods adopted in the techniques. For example, consider the techniques – IPR, G-Free and Return-less kernel that perform instruction re-writing. Instructions are re-written to remove indirect control transfer instruction bytes, especially the opcodes of the 'ret' instruction ('C2', 'C3' etc), present in unaligned instruction sequences.

6.2. ROP Specific Effectiveness Evaluation

In this part we focus our comparison on some ROP specific features, including gadgets, defense strategies and effectiveness as well, of the proposed defense techniques (also shown in Table III):

- Type of gadgets that could be detected by the corresponding technique – Only 'ret' based or 'ret' and 'jmp/call' based gadgets.
- Defense strategies that are employed in the defense techniques (as discussed in Section 3 – especially the first two).
- Possible limitation on its availability or potential evasion techniques.

6.2.1. Type of gadgets

As we can see almost in all the discussed approaches protect against the three major types of indirect control transfer instructions – 'ret', 'jmp' and 'call'. Some of the earlier approaches targeted only 'ret' based gadgets which served as a serious limitation among those techniques. However the later proposed approaches have been designed in a wiser way of handling any indirect control transfer.

6.2.2. Defense strategy

Considering the defense strategy undertaken, the trend observed is nearly all dynamic and most of the compile-time approaches target to enforce control flow integrity. This is safer than gadget prevention as often times it is not possible to identify every potential gadget by analysis. It is also the case that not every gadget can be eliminated, as it is difficult

; **00**:1–21 © John Wiley & Sons, Ltd.
DOI: 10.1002/
Prepared using

Table II. Comparison of ROP defense techniques - General Evaluation

Technique	Type	Coverage	Correctness
Control Flow Locking [22]	Compile-time	Complete binary	• Possible false positive,false negative if the Control Flow Graph is incorrect
Control-Flow Monitoring [25]	Dynamic	Complete binary	• False positive for valid 'jmp' across different functions/libraries • False positive in exception/signal handling
G-Free [21]	Compile-time	Complete binary	• Possible error due to wrong computation of encryption/decryption
ILR [17]	Randomization	Excludes external libraries	• Incorrect branch target analysis may result in false positives
IPR [16]	Randomization	Covers around 80% on average	• Possible error in instruction re-writing or re-ordering
k-Bouncer [26]	Dynamic	Only certain API calls that performs certain system-level operations	• False positive when legitimate code has instruction sequences greater than threshold • False negative if the LBR has valid recent instructions
Marlin [18]	Randomization	All Functions identified	• Possible error due to incorrect symbol resolution
ROP Defender [7]	Dynamic	Complete binary	• No potential cause of false positives. Covers scenarios including exception and signal handling
ROPecker [27]	Dynamic	Complete binary	• False negative if gadget chain is less than threshold or longer gadgets inserted between gadget chain
Return-less kernel [19]	Compile-time	Complete binary	• Possibility of error affecting normal execution, due to incorrect instruction re-writing
ROPGuard [28]	Dynamic	Only critical function calls	• Series of checks performed for every invocation, so less probability of incorrect results

to rewrite some instructions with semantically equivalent instructions. However some approaches like G-Free [21] and Return-less kernels [19] focus on both the strategies by eliminating gadgets as much as possible at pre-processing stage and ensuring CFI at the runtime (ensured by the code added at compile-time). Few other approaches target only at gadget elimination like the IPR [16]. It is interesting however that regardless of the defense strategies adopted, all these approaches handle the execution of unaligned instructions.

6.2.3. Availability/Evasion

Regarding the availability of the approach, a possible cause for limitation is the design itself. Techniques like kBouncer [26] and ROPGuard [28] that are designed to handle only certain API/function calls cannot detect ROP attack in other execution paths that do not invoke the selected API. The attackers can then take advantage of this limitation and use any gadgets that may be present elsewhere

Table III. Comparison of ROP defense techniques - ROP specific Effectiveness Evaluation

Technique	Defense Strategy	Gadgets Covered	Availability/Potential evasion techniques
Control-Flow Locking [22]	Enforce CFI	ret,jmp,call	None identified
Control Flow Monitoring [25]	Enforce CFI	ret,jmp,call	• Cannot be used with Position Independent code
G-Free [21]	Enforce CFI and eliminate gadgets	ret,jmp,call	• Access to the random key used for encryption, may cause evasion
ILR [17]	Hinder prediction of addresses	ret,jmp,call	• If attacker can get any unrandomized address, may exploit and launch the attack
IPR [16]	Eliminate gadgets	ret,jmp,call	• Does not eliminate every gadget, depends on semantics and coverage
k-Bouncer [26]	Enforce CFI	ret,jmp,call	• Can check only LBR registers number of branches • • Gadgets in other instructions excluding the selected APIs can be evaded
Marlin [18]	Hinder prediction of addresses	ret,jmp,call	• Strength depends on number of functions randomizable
ROP Defender [7]	Enforce CFI	ret	None identified
ROPecker [27]	Enforce CFI	ret,jmp,call	• If sliding window size is larger, possible evasion by including gadgets that all fit within one window • Checks gadgets of length only upto 6
Return-less kernel [19]	Enforce CFI and eliminate gadgets	ret	• If the attacker can acquire knowledge of the return indices, he can use it directly to populate the stack
ROPGuard [28]	Enforce CFI	ret	• Gadgets in other instructions excluding the critical calls can be evaded

excluding the considered API to escape detection from these techniques.

Few other approaches can be evaded due to the factors the approach itself is dependent upon. For example, considering the kBouncer approach again, the efficiency of the approach depends on LBR registers. The registers may not only contain the instructions of the program in concern, it may as well contain instructions pertaining to context-switch and other running processes. Therefore more number of registers would help in capturing more branches of the concerned program. In some processors the number is 4 or 8 only. Also when the number is small, there is more probability for an attacker to circumvent the technique by forming gadgets (in the API) with LBR number of legitimate branches. An improvement to this would be to not entirely depend upon the LBR registers but also include additional logic to conclude an ROP. This is what is proposed in ROPecker [27] which checks for past execution in LBR registers and future execution by examining the stack trace and comparing with gadget database. However this still has possibilities for evasion. The approach checks for gadgets with length of only upto a pre-defined length instructions (6 proposed in [27]. Although the pre-defined length can be any number, it may not be easy to predict the right choice for all applications) in its pre-processing stage of creating gadget database. The attacker can evade this approach by either

; **00**:1–21 © John Wiley & Sons, Ltd.
DOI: 10.1002/
Prepared using

choosing only gadgets of length greater than 6 (in this case) or inserting these longer gadgets in between shorter gadgets (as identified in database) to break the chain length such that it does not exceed the threshold. The authors of ROPecker argued that using only long gadgets is less probable as they usually serve little purpose in achieving useful operations typically performed by ROP exploits. Though the proposed length 6 may work well in today's exploits it may not necessarily be effective always. For the introduction of long gadgets between shorter gadgets, the authors of ROPecker proposed the method to count the gadget chain length across sliding windows. For example, the chain length is considered together in three windows and if the length exceeds this threshold, then it is considered as ROP. Again, the number of windows to take into account cannot be a constant value for different applications and it is difficult to predict the required number before execution of the application.

In a similar manner Marlin's [18] efficiency is dependent on the number of function blocks in the program. If an attacker can find the address of any function by brute-force methods and if the function contains a potential gadget, then the technique is rendered ineffective. Therefore the number of attempts for brute-force methods depends on the number of functions that are available for randomization. The attacker's effort is directly proportional to the number of functions identified. The authors claimed for a binary with n randomizable functions, the number of required attempts (brute-force methods) is $n!$.

Considering from a different perspective, evasion can also happen due to the leakage of information which is crucial to the approach. To cite an example, in G-Free [21] if an attacker can gain access to the *random key*, then he/she can easily populate the stack with the encrypted/decrypted addresses and circumvent the check. Since the random key is only stored in a special file, leakage is not impossible. As discussed by the authors too, the *random key* must be generated at runtime. Otherwise an asymmetric encryption could add more protection than a symmetric one, since encryption/decryption depends on different keys. Likewise in Return-less kernel [19], if the attacker can get access to the centralized return index table, he/she can use them to populate the stack with correct indices corresponding to the gadgets intended for exploit.

6.3. Performance Metrics

In this section we focus on the performance metrics observed in the above defense techniques. In particular the following criteria are studied:

- Space overhead.
- Runtime overhead.
- Dependence on side information, i.e. source code and debug symbols.
- Dependency of the technique on any other external framework/libraries.

6.3.1. Space Overhead

Most of the overhead is incurred from storing the results of pre-processing stages in databases. In the case of ILR [17], database is required for storing the identified instructions from the binary and re-write rules. In ROPecker [27] and kBouncer [26], information about potential gadgets identified at offline stages are stored in database. Databases for shared libraries can be shared across processes. Likewise in Marlin [18], database is required for storing the function symbols extracted that are used for randomization. In Return-less kernel [19], the overhead is from the centralized return index table. If the size of the table is limited, and the number of functions (valid 'ret' instructions) exceed the size of the table, then other techniques like double indirection may need to be employed. Sometimes the overhead is from additional frameworks involved like the PIN framework in ROPDefender [7] and Control Flow Monitoring. The approach that has the minimal overhead among all mentioned mechanisms is IPR [16]. Since the randomizations in IPR are in-place there is no requirement for additional information storage.

6.3.2. Runtime overhead

The runtime overhead is usually high for dynamic approaches in general owing to the actual work done during the program execution. ROP defense techniques are no exception. Particularly ROPDefender [7] and Control Flow Monitoring [25] have reported upto $2X$ and $3.5X$ average total execution time respectively. Both these approaches use the PIN framework, which is the major cause of such high overheads. Though the framework has a code cache to save frequently repeated instructions, if the number of these repeated instructions is much larger compared to the code cache size, then there are chances of increased performance overhead. There are few other dynamic approaches (not

discussed in this paper) like DROP [29], TRUSS [30] which incurs worser overheads of upto $5X$ on average. On the other hand, though kBouncer [26] and ROPGuard [28] are dynamic approaches they have comparatively low overheads due to their ROP checking logic being triggered only for specific API calls. In the case of ROPecker the sliding window size greatly affects the runtime overhead. Small windows means more exceptions which would in turn mean increased overhead and vice versa. However larger window sizes have the disadvantage of the possibility of being able to fit all ROP gadgets within one window to escape detection. Another source of runtime overhead in ROPecker is from the instruction emulation performed to check the future execution flow of the program at the time of checking.

Compile-time approaches usually do not have higher runtime overheads, as most of the work is performed compile-time as suggested by their name itself. This minimal overhead is caused by the ROP checking logic added to the code like the encryption/decryption in G-Free, return index computation in return-less kernel and lock/unlock in CFL.

Randomization approaches' overhead is somewhere between the compile and dynamic time approaches. Since randomization is done for the instruction locations, there is not too much work involved during the execution of the actual application in concern. This holds well especially in the case of IPR [16], where the very minimal overhead could be caused by some re-writing instructions that may be less optimal than the compiled version. However an approach like ILR [17] that randomizes every instruction (position-independent), the next instruction to be executed is determined at runtime only by the per-process virtual machine employed for this specific purpose. In such cases, the runtime overhead is obviously higher. In the case of Marlin [18], the overhead is at load-time when the randomization is performed based on the symbol resolution database. The authors of Marlin also proposed to reduce the effort of pre-processing stage by making it a one-time step, rather than repeating for every execution. In fact Marlin tries to achieve a fine trade-off between the performance overhead (randomizing function blocks rather than individual instructions) and effectiveness (better compared to ASLR which randomizes only parts of process like stack and heap).

6.3.3. Dependence on side information

As already discussed in Section 4, compile-time approaches require source-code. Most of the other approaches do not require any side information, with few exceptions that depend on debug symbols. Although those approaches that do not depend on such information use disassemblers to retrieve the instructions (without symbolic debugging information), they cannot be guaranteed for complete coverage always. The dependence on side information for each approach is shown in the Table IV.

6.3.4. Dependence on framework/hardware

Almost every approach depends on disassemblers to retrieve the instruction sequences from program binary for gadget analysis. Typically IDADiassembler is used for windows applications and ReadELF for Unix applications. Few dynamic approaches like ROPDefender [7] and Control FLow Monitoring [25] that maintain a shadow stack and compare every indirect control transfer instruction uses the PIN dynamic binary instrumentation framework for the runtime monitoring. ILR [17] depends on per-process virtual machine for fetching the next instructions. These dependencies are kind of external that needs to be integrated with the technique. Some techniques depend on built-in support (i.e. hardware) that is provided with the processor itself like in the cases of kBouncer [26] and ROPecker [27]. These techniques depend on the LBR registers that is present in the processor itself.

7. DISCUSSION - TOWARDS EFFICIENT ROP DEFENSE

This section aims at discussing some of the essential factors and insights towards efficient detection and prevention of ROP attacks. Based on the observations made on surveyed defense techniques (discussed in the previous sections), the following properties are considered essential for a robust and effective approach:

- **Defense strategy:** As discussed before, ROP Defense could adopt either gadget elimination or control flow integrity or sometimes combine both. Considering only gadget elimination or modification, it may not be enough to remove every available gadget. As a result there are still chances that the attacker's targeted instructions are not affected by the process. Also some of the attacker's operations could be in such a way that, in a gadget of many number of instructions, modification of one or some part of the gadget may not have any significant effect on the gadget's actual intended operations. Considering the

; **00**:1–21 © John Wiley & Sons, Ltd.
DOI: 10.1002/
Prepared using

Table IV. Comparison of ROP Defense techniques - Performance Metrics

Technique	Space Overhead	Runtime Overhead	Side Information	External dependencies
Control-Flow locking [22]	Lock/unlock code	Minimal	Yes	None
Control-Flow Monitoring [25]	PIN (including shadow stack)	Upto 3.5X on avg.	No	Disassembler, ReadELF, PIN framework
G-Free [21]	Encrypt/Decrypt code, increased by 26% on avg	3% on avg.	Yes	None
ILR [17]	Instruction database, re-write rules, per-process VM	Mainly from VM, 13-16% on avg.	Yes	Disassembler
IPR [16]	None significant	Minimal	No	Disassembler
k-Bouncer[26]	Offline analysis result tables, detours frwk	Minimal, only for API calls	No	LBR registers, Detours framework
Marlin [18]	Symbol resolution database	None	No	Disassembler, ReadELF
Return-less kernel [19]	Centralized Index table	Minimal	Yes	None
ROP Defender [7]	PIN (including shadow stack)	Upto 2X on avg.	No	PIN framework
ROPGuard [28]	None significant	Minimal, only for critical functions	No	None
ROPecker [27]	Instruction gadget database	Mainly from ROP check and emulation, depends on sliding window size	No	LBR registers

control flow integrity strategy, though this is more effective in ensuring the program that is not diverted from its expected behavior, it incurs higher overheads owing to the large number of indirect control transfer instructions in a typical application. To improve such overheads, it would be beneficial to implement both the strategies – gadget elimination and enforcing CFI.

- **Gadget types:** Though the attack started with ROP i.e only 'ret' instruction based, as put forth by Checkoway.et,al [8] and Bletsch.et.al, [9], it could be directed via the 'jump/call' instructions as well. Therefore an efficient defense technique must account for all types of indirect control transfer instructions – 'ret', 'jmp' and 'call'. It is also important that these gadget types are covered in both aligned and unaligned instructions.
- **Gadget length:** In the process of gadget elimination, most approaches consider short gadgets only, of length upto 5 or 6 instructions. This is based on the observations from current ROP exploits. Though this may be sufficient for detecting today's exploits, it may not be correct invariably. With the extent of research advancing in the area of ROP, future defense techniques need to be robust enough to handle gadgets of varying length, independent of assumptions. kBouncer [26] is good in this aspect, it considers gadget length of upto 20 instructions.
- **Other Parameters:** Besides the gadget length discussed before, some other parameters that need to be well chosen are threshold values in looking for gadget chains and frequency or number of indirect branches executed. Poor choice of these values without proper rationale may result in increased false-positives.
- **Dependency:** It is not unusual or unnecessary for a defense technique to be dependent on external factors like hardware, frameworks and libraries. But it is important that the dependency does not affect the efficiency or performance of the technique. Like in the case of kBouncer which is dependent on LBR registers, its efficiency is directly proportional to the number of available registers. Considering performance impact, some approaches that requires an instrumentation framework, have very high overheads making their usage difficult.
- **Security for ROP logic:** Almost every approach is based on the assumption that the tool/ROP checking logic itself is protected from any misuse by attackers. It is important that this expected integrity and security is guaranteed to prevent evasion.

8. CONCLUSION AND FUTURE WORK

Return-Oriented Programming is a recent type of software attack that exploits the buffer overflow vulnerability. Recent advancements in this subject has resulted in various types of both – ROP exploits and ROP defense mechanisms. This paper focuses on the defense techniques particularly those proposed in recent few years. This paper provides a brief classification of the defense mechanisms and discusses some of them in details. A comparative evaluation of the techniques is also provided focusing on ROP specific parameters and certain other general parameters. This paper also provides some insights into the important aspects to be considered for future developing an efficient defense mechanism.

Besides the techniques discussed there are certain other techniques also which though not specific to ROP, can still be used in detecting such type of code re-use attacks. Also few hardware-based virtualization approaches have been proposed to mitigate ROP recently. An extended survey including these approaches as well is considered to be included in future work.

REFERENCES

1. Chen S, Xu J, Sezer EC, Gauriar P, Iyer RK. Non-control-data attacks are realistic threats. *Usenix Security*, vol. 5, 2005.
2. One A. Smashing the stack for fun and profit. *Phrack magazine* 1996; **7**(49):14–16.
3. Designer S. return-to-libc attack. *Bugtraq, Aug* 1997; .
4. Soderstrom EK. Analysis of return oriented programming and countermeasures. PhD Thesis, Massachusetts Institute of Technology 2014.
5. Shacham H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). *Proceedings of the 14th ACM conference on Computer and communications security*, ACM, 2007; 552–561.
6. Roemer R, Buchanan E, Shacham H, Savage S. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)* 2012; **15**(1):2.
7. Davi L, Sadeghi AR, Winandy M. Ropdefender: A detection tool to defend against return-oriented programming attacks. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ACM, 2011; 40–51.
8. Checkoway S, Davi L, Dmitrienko A, Sadeghi AR, Shacham H, Winandy M. Return-oriented programming without returns. *Proceedings of the 17th ACM conference on Computer and communications security*, ACM, 2010; 559–572.
9. Bletsch T, Jiang X, Freeh VW, Liang Z. Jump-oriented programming: a new class of code-reuse attack. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ACM, 2011; 30–40.
10. Schwartz EJ, Avgerinos T, Brumley D. Q: Exploit hardening made easy. *USENIX Security Symposium*, 2011.
11. Abadi M, Budiu M, Erlingsson Ú, Ligatti J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 2009; **13**(1):4.
12. Team P. Pax address space layout randomization (aslr) 2003.
13. Roglia GF, Martignoni L, Paleari R, Bruschi D. Surgically returning to randomized lib (c). *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, IEEE, 2009; 60–69.
14. Sovarel AN, Evans D, Paul N. Where's the feeb? the effectiveness of instruction set randomization. *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, USENIX Association: Berkeley, CA, USA, 2005; 10–10. URL `http://dl.acm.org/citation.cfm?id=1251398.1251408`.
15. Gupta A, Habibi J, Kirkpatrick M, Bertino E. Marlin: Mitigating code reuse attacks using code randomization. *Dependable and Secure Computing, IEEE Transactions on* May 2015; **12**(3):326–337, doi: 10.1109/TDSC.2014.2345384.
16. Pappas V, Polychronakis M, Keromytis AD. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. *Security and Privacy (SP), 2012 IEEE Symposium on*, IEEE, 2012; 601–615.
17. Hiser J, Nguyen-Tuong A, Co M, Hall M, Davidson JW. Ilr: Where'd my gadgets go? *Security and Privacy (SP), 2012 IEEE Symposium on*, IEEE, 2012; 571–585.
18. Gupta A, Kerr S, Kirkpatrick MS, Bertino E. Marlin: A fine grained randomization approach to defend against

; **00**:1–21 © John Wiley & Sons, Ltd.
DOI: 10.1002/
Prepared using

rop attacks. *Network and System Security*. Springer, 2013; 293–306.

19. Li J, Wang Z, Jiang X, Grace M, Bahram S. Defeating return-oriented rootkits with return-less kernels. *Proceedings of the 5th European conference on Computer systems*, ACM, 2010; 195–208.
20. Hund R, Holz T, Freiling FC. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. *USENIX Security Symposium*, 2009; 383–398.
21. Onarlioglu K, Bilge L, Lanzi A, Balzarotti D, Kirda E. G-free: defeating return-oriented programming through gadget-less binaries. *Proceedings of the 26th Annual Computer Security Applications Conference*, ACM, 2010; 49–58.
22. Bletsch T, Jiang X, Freeh V. Mitigating code-reuse attacks with control-flow locking. *Proceedings of the 27th Annual Computer Security Applications Conference*, ACM, 2011; 353–362.
23. Reddi VJ, Settle A, Connors DA, Cohn RS. Pin: a binary instrumentation tool for computer architecture research and education. *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, ACM, 2004; 22.
24. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices* 2005; **40**(6):190–200.
25. Chen P, Xing X, Han H, Mao B, Xie L. Efficient detection of the return-oriented programming malicious code. *Information Systems Security*. Springer, 2011; 140–155.
26. Pappas V, Polychronakis M, Keromytis AD. Transparent rop exploit mitigation using indirect branch tracing. *Proceedings of the 22nd USENIX Conference on Security*, 2013.
27. Cheng Y, Zhou Z, Yu M, Ding X, Deng RH. Ropecker: A generic and practical approach for defending against rop attacks. The 21th Annual Network and Distributed System Security Symposium (NDSS14), 2014.
28. Fratric I. Runtime prevention of return-oriented programming attacks 2012.
29. Chen P, Xiao H, Shen X, Yin X, Mao B, Xie L. Drop: Detecting return-oriented programming malicious code. *Information Systems Security*. Springer, 2009; 163–177.
30. Sinnadurai S, Zhao Q, fai Wong W. Transparent runtime shadow stack: Protection against malicious return address modifications 2008.