International Journal of Digital Crime and Forensics
Volume 13 • Issue 2 • March-April 2021

Automatic Generation of ROP Through Static Instructions Assignment and Dynamic Memory Analysis

Ning Huang, National University of Defense Technology, China
Shuguang Huang, National University of Defense Technology, China
Chao Chang, National University of Defense Technology, China

ABSTRACT

W⊕X is a protection mechanism against control-flow hijacking attacks. Return-oriented programming (ROP) can perform a specific function by searching for appropriate assembly instruction fragments (gadgets) in a code segment and bypass the W⊕X. However, manual search for gadgets that match the conditions is inefficient, with high error and missing rates. In order to improve the efficiency of ROP generation, the authors propose an automatic generation method based on a fragmented layout called automatic generation of ROP. This method designs new intermediate instruction construction rules based on an automatic ROP generation framework Q, uses symbolic execution to analyze program memory states and construct data constraints for multi-modules ROP, and solves ROP data constraints to generate test cases of an ROP chain. Experiments show that this method can effectively improve the space efficiency of the ROP chain and lower the requirements of the ROP layout on memory conditions.

KEYWORDS

Automatic Generation, Fragmented Layout, Intermediate Instruction Construction, Multi-Modules ROP, Return-Oriented Programming, Solve Constraint, Symbolic Execution, W⊕X

INTRODUCTION

Mining and exploitation of software vulnerabilities have become popular issues with the development of information technology. Many protection mechanisms for different types of exploit technologies are also emerging. However, many methods can be used to bypass the mechanisms given the limitations of these mechanisms.

Control-flow hijacking attack occurs due to the vulnerability of an overflow because computers cannot distinguish whether a binary number in the memory page is a code or data, thus resulting in the injected data being executed as a code (Shao & Gao, n.d.). Linux system first introduced the W⊕X mechanism in 2000 to address the abovementioned problem (Executable Space Protection, 2018); then, Windows system introduced Data Execution Prevention in XP SP2 and its subsequent products. The basic principle of this mechanism is to distinguish a code segment from a data segment by marking

DOI: 10.4018/IJDCF.2021030104

the memory pages as executable/non-executable. The shellcode located in the data segment cannot be executed with the W⊕X (Wei et al., n.d.).

In 2000, a solar designer proposed the ret2libc. The ret2libc hijacks the program control flow and controls the program that jumps to an existing system function because W⊕X does not intercept the code that is located on an executable page. Schacham (Shacham, 2007)[6](Roemer et al., 2012) proposed the ROP based on ret2libc. Compared with ret2libc, ROP uses a smaller assembly instruction fragment called gadget, which improves the generality of the method. Checkoway (Checkoway et al., 2010) proposed an ROP construction method without return instruction and extended the use of the ROP. Lu (Lu et al., 2011) proposed a squeezable, printable, and implementable method on the basis of RIX method and improved the flexibility of an ROP payload.

Various automatic analysis and test case generation techniques for binary program vulnerabilities have emerged in recent years with the development of program analysis techniques (Chipounov et al., 2012). Avgerinos proposed an automatic exploit generation (AEG) (Avgerinos et al., 2012) to determine whether the input can trigger the unsafe state of the program through program verification techniques. Cha proposed the Mayhem for an automatic generation of an exploit (Sang et al., 2011). The Mayhem uses a symbolic execution technology to automatically mine vulnerabilities and generate an exploit. However, these methods do not consider the impact of the W⊕X.

Huang proposed an automatic exploit generation method based on symbolic execution called CRAX (Huang et al., 2012). This method uses a selective symbolic execution, utilizes binary files that can be directed to the vulnerability point as an input, boots the program to run and triggers control-flow hijacking, and generates an exploit. The CRAX method uses the ret2libc with the influence of the W⊕X. CRAX checks the controllable space in the memory and injects the address of a system function while analyzing the state of the control-flow hijacking. The hijacked program reads the address, and its control flow is directed to the system function. However, this method cannot achieve the automatic construction and layout of the ROP, and the exploit can only perform limited functions under the W⊕X.

Schwartz proposed an ROP automatic construction method Q (Schwartz et al., 2011) (Brumley, D., Jager, I., & Avgerinos, T. 2011). This method implements an automatic search of gadgets and automatically constructs ROP chains through gadget-oriented programming languages. Its workflow is as follows: First, an executable program or library file is provided to Q, and a gadget set with a specific function is searched; second, the program used to build intermediate statement sequences is analyzed; finally, the sequence of intermediate statements is evaluated, and a suitable set of gadgets is assigned for each intermediate statement to form an ROP chain. Q (He & Su, 2016) has the following limitations: (1) The payload generated through this method only continue on the basis of the perspective of a functional implementation and disregard the requirements of the ROP layout on the controllability conditions of the memory. (2) In the process of allocating gadgets for the intermediate statement sequence, this method has a record of the program stack location modification, which resulting in a low space efficiency of the ROP.

This paper proposes an ROP automatic generation method on the basis of the fragmented layout to address the problems of ROP automatic generation technologies. The contributions of this papers are shown as following:

1. This paper designs a new static instruction assignment rule which can reduce the length of ROP payload. This rule is designed on the basic of Q and optimizes the static instruction assignment of ROP modules switching. In the case of implementing a similar functionality, the ROP chain generated by the new assignment method uses few gadgets and a short binary code.
2. This paper propose a dynamic memory analysis method for automatic fragment ROP layout. This method uses the generated ROP payload as an input, utilize the selective symbol execution to direct the program execution to the control-flow hijacking point, analyze the program memory layout, and construct the data constraints of the fragmented ROP chain. An ROP chain that satisfies

the memory layout conditions is generated by solving the fragmented ROP data constraint, thus alleviating the problem of poor practicability caused by the requirement of the ROP for memory controllability.

BACKGROUND

W⊕X

Figure 1 illustrates the process of stack overflow exploitation and the scenario of shellcode execution with and without the W⊕X. The grey blocks in Figure 1 are the memory blocks of registers which are tainted by input data; the white blocks the memory blocks that are not tainted. In the case of stack overflow, the overflow data in stack may overwrite some important data illegally, such as function address, which would lead to the overwritten of IP register and control-flow hijack. Generally, arbitrary code (shellcode) execution is the most dangerous consequence of control-flow hijack.

Shellcode must be injected into the memory to achieve control-flow hijacking attacks. Jumping to the address of the shellcode and executing the shellcode are the solutions when the control flow of a program is hijacked. Therefore, current operating systems have introduced the W⊕X protection mechanisms to address the defect to prevent computers from identifying the data as a code and limit the execution rights of the data (Gao et al., 2013).

For the W⊕X, current major bypass technologies (Stojanovski et al., 2007) include the following three types. (1) The W⊕X is turned off. Normally, the system will set the function to turn on/off the W⊕X. W⊕X disablement of part or all of the memory areas of the program can be achieved by operating these functions. (2) Processes that are not enabled are used with the W⊕X. Under certain conditions, W⊕X may cause process abnormalities. Several processes do not involve the W⊕X to avoid this situation. (3) Existing code is used. This method performs a specific function, such as ret-to-libc (or ret2libc) and return-oriented programming (ROP), by searching for the existing code in a memory.

Return Oriented Programming

ROP is developed on the basis of ret2libc. This technology aims to address the defect that the W⊕X does not limit the execution permission of the existing code in the code page and bypass the W⊕X. The main principle is to construct a gadget set that ends with a retn instruction by searching the code page. The qualified parts from the gadget set are filtered and combined to produce an ROP chain with a specific function (Prandini & Ramilli, 2012). Figure 2 depicts the code execution sequence of an ROP chain and its stack structure.

An ROP chain for implementing a program in Code 1 in a 32-bit system has an effective length (excluding pad characters) of 424 bytes by using an automatic ROP generation tool, namely, ROPC. If the ROP chain is arranged in the actual program, then the program is required to satisfy the stack

Figure 1. Stack overflow exploitation and shellcode execution

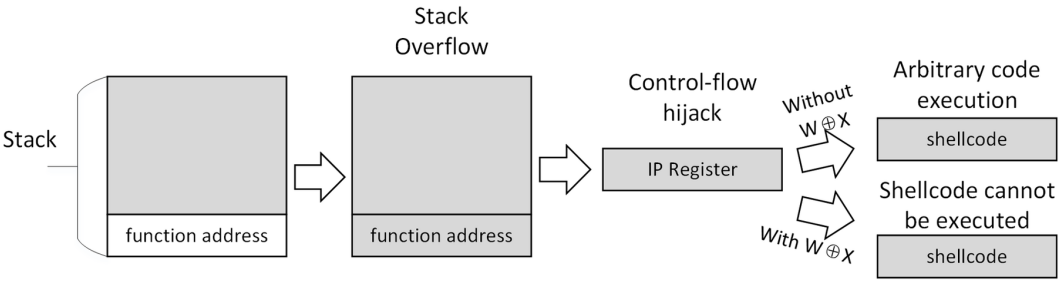
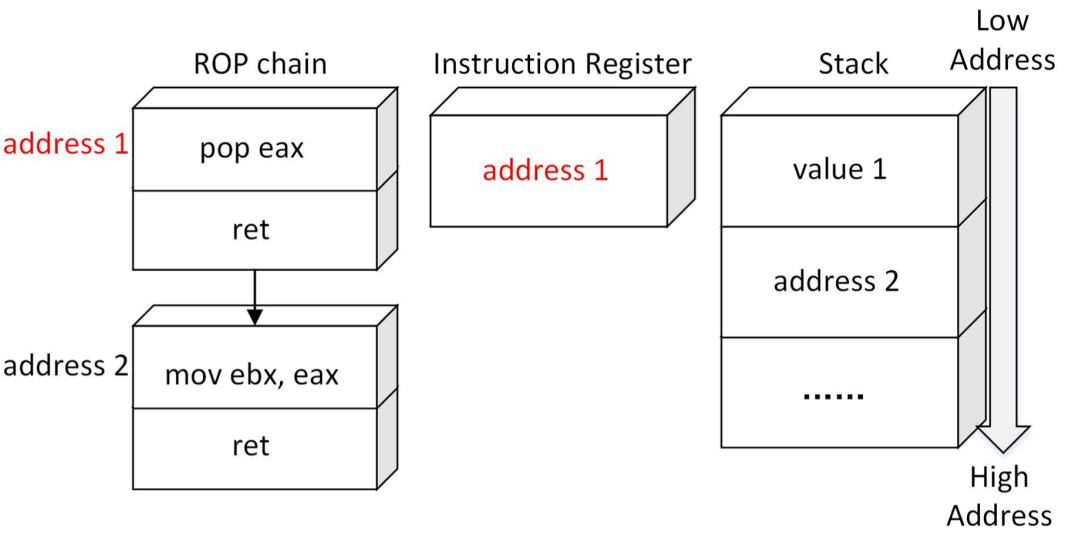


Figure 2. ROP chain and the structure of a stack



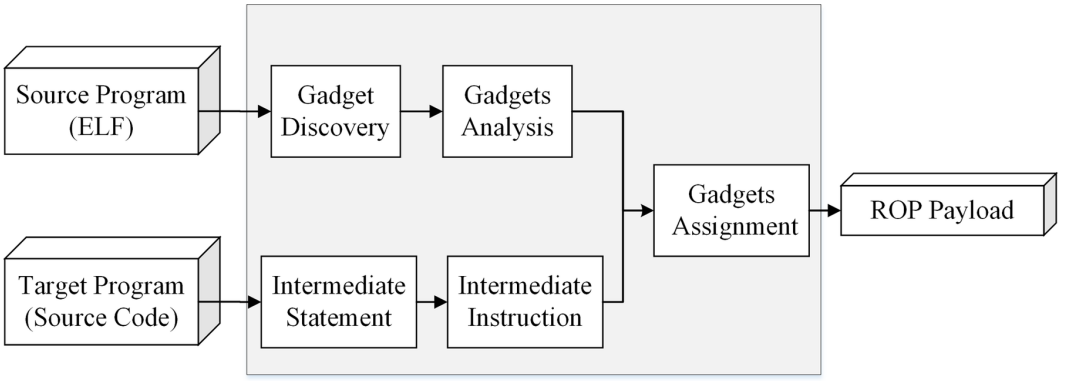
with a controllable space of at least 424 bytes when the control flow is hijacked. This requirement limits the application of the ROP in the actual program.

```
Code 1
fun main()
{
    i = 0
}
```

Static ROP Construction of Q

Figure 3 illustrates the structure of Q. ROPC is a practical tool for automatic ROP generation based on Q. This paper uses ROPC to analyze the working details of automatic ROP generation. The main working steps of ROPC includes.

Figure 3. Framework of Q



Gadget Discovery and Analysis

The following attributes are recorded in the candidate gadget set when these attributes are satisfied.

Jump controllability. A gadget can jump exactly to the next specified gadget after this gadget has been executed. Generally, the candidate gadget must end with a jump instruction, such as retn, to control the jump process. ROPC selects the gadget that ends with the Ret instruction.

Controllability of the target register/memory. ROPC uses the /ROPC/verify component to verify candidate gadgets. The source and destination operands of the candidate gadgets must be controllable registers, pointers to the memory, or memory areas.

Operability of the target register/memory. ROPC combines gadgets to an ROP chain that can achieve a specific function through data operations on registers or memory areas. ROPC will filter candidate gadgets that do not operate on any register or memory during the construction of the candidate set.

Intermediate Statement and Instruction Analysis

The allocation of gadgets to intermediate instructions must satisfy the following conditions.

Same semantics. Each intermediate instruction consists of at least one candidate gadget. A gadget sequence must perform the same function as its matching intermediate instruction.

Register is free. Gadget assignment reuses the register conflict detection rule of the ROPC. According to this rule, the registers in the gadgets allocated for an intermediate instruction must not conflict with registers that were previously called and not freed.

Operator matching. For intermediate instructions with operator parameters, the sequence of candidate gadgets must contain at least one gadget with the same operator. Translation process from ROPL to gadgets.

The target program is written using the high-level language ROPL, which is defined in the ROPC. The ROPL target program can realize functions, such as function calling, variable assignment, condition judgment, branch jump, and loop operation. ROPC realizes the functions of the target program through the ROP payload.

The process of constructing an ROP chain for an ROPL target program analysis is as follows. First, an intermediate statement sequence is generated through the ROPL target program analysis. Second, an intermediate instruction sequence is produced. Finally, the gadget sequence is allocated. Especially, the intermediate statements are the program symbols generated after analyzing the target program. Figure 4 shows the translation process from ROPL to assemble instructions (gadget).

SYSTEM OVERVIEW

Existing ROP automatic generation technologies address the problems of gadget search and classification, high-level language semantic analysis, and assignment and arrangement of gadgets to intermediate instructions. However, based on the actual application effect, the ROP generated by existing technologies cannot satisfy the constraints of the memory state in most scenarios. Thus, this paper proposes a method called automatic generation of ROP (AGROP) based on the ROP fragmented layout to alleviate the abovementioned problem. The main working framework of the AGROP is demonstrated in Figure 5.

Figure 4. Translation process from ROPL to gadgets

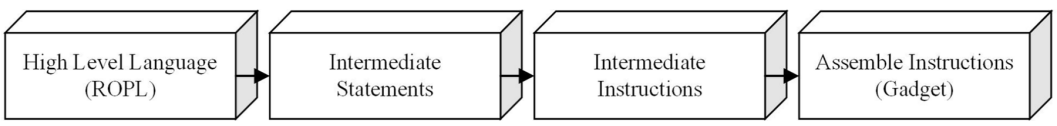
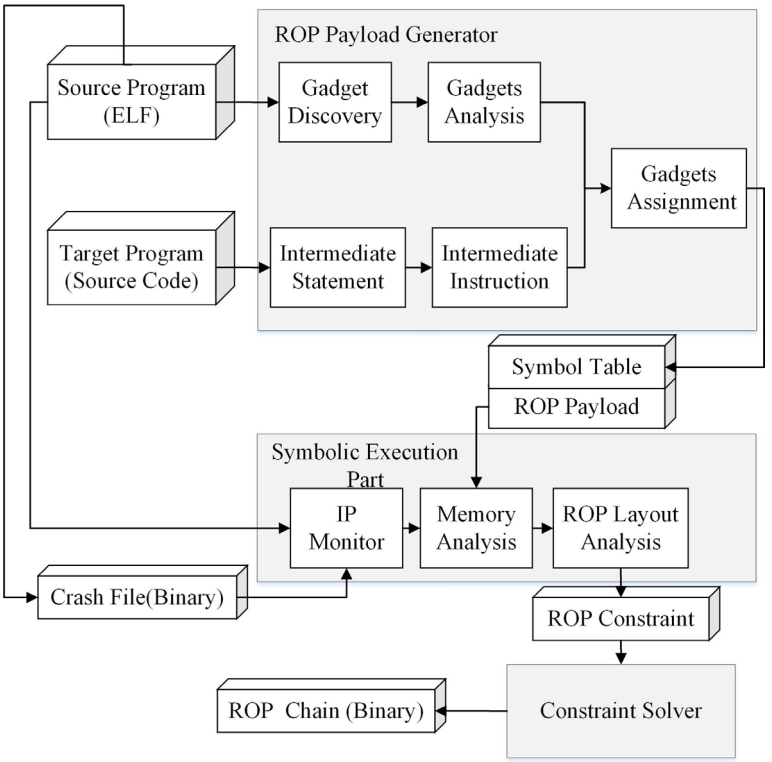


Figure 5. Overview of the AGROP design



This method is composed of two parts, namely, ROP payload generator and symbol execution part.

The inputs to the ROP payload generator include the source program (ELF) and the target program (source code).

The source program is a vulnerable program. The ROP payload generator uses Q to perform a gadget search and classification of a source program and generate a gadget candidate set.

The target program is the target function program of the ROP. On the basis of the source code analysis of the target program, this paper constructs a sequence of intermediate statements and instructions that are suitable for the functions of a target program. The gadgets that can match the functions of each intermediate instruction are found from the candidate gadget set through gadget assignment, and the ROP payload is generated.

A target program is composed of a high-level language. A symbol table is a record of source code analysis files of the target program and is composed of intermediate statement sequences. This table records the information of inter-module calls, variable application, and several other operations of the target program. This paper uses the target program in Code 2 as an example. The symbol table of intermediate statements for this program is displayed in Code 3.

```
Code 2
fun main()
x = 1
foo(x)
fun foo(x)
y = x
```

```
Code 3
main: Enter(0)
Assign(x, Const 1)
Call(foo, args: x)
Ret(main)
foo: Enter(1)
Assign(y, Var x)
Ret(foo)
```

On the basis of ROPC, AGROP analyzes the behavior of each gadget, defines the corresponding gadget type, and categorizes it. Table 1 summarizes the semantic definition of these gadgets.

Table 2 presents the ROPL intermediate statements and their semantic definitions.

Intermediate instruction is an instruction that expresses a specific function between the intermediate statements and the gadget. AGROP establishes the intermediate statement sequence by analyzing the ROPL source code. Then, for each intermediate statement, an intermediate instruction sequence that satisfies the semantics of the statement is allocated. Table 3 displays parts of the intermediate instructions and their semantic definitions.

The inputs to symbolic execution parts are as follows: source program, crash file, and ROP payload.

A crash file is input data that can trigger the control-flow hijacking state of a source program. The AGROP marks all the tainted data that are passed into the source program as symbolic data. All memory spaces or registers that are tainted with symbolic data are marked as symbolic memory/registers during the dynamic running of a source program. Currently, a variety of automatic exploit

Table 1. Classification of gadgets according to the semantic definition

Name	Parameters	Semantic Definition
LoadConst	reg * int	reg ← int
CopyReg	reg1 * reg2	reg1 ← reg2
BinOp	reg1 * reg2 * op * reg3	reg1 ← reg2 op reg3
ReadMem	reg1 * reg2 * int	reg1 ← [reg2 + int]
WriteMem	reg1 * int * reg2	[reg1 + int] ← reg2
ReadMemOp	reg1 * op * reg2 * int	reg1 ← reg1 op [reg2 + int]
WriteMemOp	reg1 * int * op * reg2	[reg1 + int] ← [reg1 + int] op reg2
OnEsp	op * reg	esp ← esp op reg

Table 2. Semantic definitions and intermediate statements of the ROPL

Name	Parameter	Semantic Definition
Assign	id * exp	id← exp
WriteMem	id * exp	[id]← exp
Branch	cond * id	id: cond
Enter	int	Initialize the current module
Ret	id	Return to a module whose name is id
Call	id * List	Call for a module whose name is id

Table 3. Parts of the semantic definitions of the intermediate instructions

Name	Parameter	Semantic Definition
Rawhex	int	int
OpStack	op * reg	esp ← esp op reg
BinO	reg1 * reg2 * op * reg3	reg1 ← reg2 op reg3
LoadRegConst	reg	reg ← [esp]
ReadMConst	reg * int	reg ← [int]
WriteMConst	int * reg	[int] ← reg

generation tools determine whether a program enters the control-flow hijacking state by checking the symbolic attributes of an IP register (Avgerinos et al., 2012)[16].

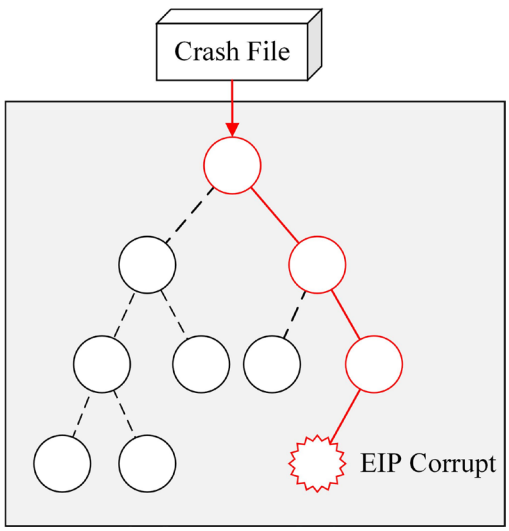
The symbolic execution part uses an optimized symbolic execution with a path-guided algorithm to reduce the time overhead detection of control-flow hijacking through symbolic execution (Huang et al., n.d.)[20]. The crash file is used as the input of the source program, and the source program is guided to run dynamically along the determined program path until the control-flow hijacking state is triggered. Figure 6 exhibits the process of triggering the flow hijacking state through an optimized symbolic execution by using a crash file.

The AGROP collects the states of the stack and controllable memory area during the optimized symbol execution. Combined with the ROP payload, the AGROP analyzes whether these states satisfy the layout condition of the ROP chain, and a corresponding data constraint of the ROP called ropConstraint is constructed. A fragmented ROP test case can be automatically generated by solving ropConstraint.

AUTOMATIC GENERATION AND FRAGMENTED LAYOUT FOR MULTI-MODULES ROP

This paper calls the ratio of the statement sequence length of the target program and the length of memory space occupied by ROP chain as the space efficiency of ROP by referring to a section of

Figure 6. Path selection of a source program with a path-oriented symbolic execution



the ROPL source code. In Formula (1), Len_{stmts} indicates the number of intermediate statements in the sequence, Len_{memory} indicates the number of bytes of memory space required by the ROP chain, and S is the space efficiency.

$$S = \frac{Len_{stmts}}{Len_{memory}} \tag{1}$$

Additional bytes in the memory occupied by the ROP chain indicates a low space efficiency, and minimal bytes occupied by the ROP chain denotes a high space efficiency when the ROPL statement sequence length is fixed.

Static Instructions Assignment Rules For Module Switching

The existing ROP auto-construction tool ROPC, records and modifies the stack pointer by statically evaluating the execution of the gadgets during the ROP construction and ensures that the stack structure is complete and accurate during the ROP execution. However, the records and modifications of the stack pointer require the support of a certain number of gadgets. This part of the gadgets is indirectly related to the function implementation of the ROPL target program.

To solve the problem above, this paper designs a new translation rule of intermediate statements for the module calling in the multi-module ROP. In the process of static ROP function analysis and instruction sequence construction, the new rule uses special characters to replace the stack pointer that must be modified in module switching, rather than using the gadgets. In Code 2, the target program contains two modules, namely, the main and the foo. In the main module, an intermediate statement Call is used to call the foo module.

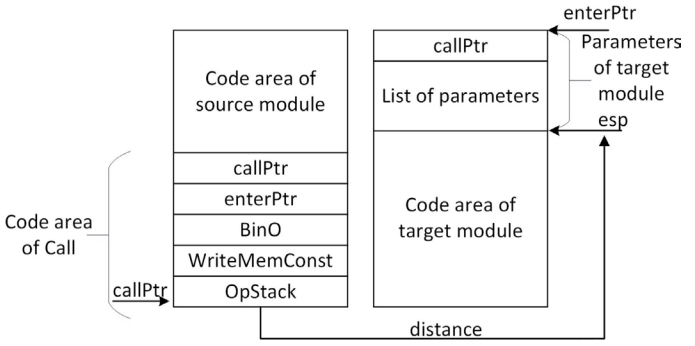
According to the semantic definition of statements for module switching in Table 2, the new rules for translating the intermediate statement to intermediate instruction designed in this study are as follows.

(1) The rule for translating the Call statement to intermediate instruction sequence is expressed in $Rule_{Call.}$ The $Rule_{Call}$ represents the set of intermediate instructions included in the Call statement. The set consists of a series of intermediate instructions that are arranged in a certain order. The semantics of the intermediate instructions in the set are listed in Table 3:

$$Rule_{Call}:$$
$$Rawhex(presentPtr)$$
$$Rawhex(targetPtr)$$
$$WriteMemConst(targetPtr, args)$$
$$BinO(distance = presentPtr - targetPtr)$$
$$OpStack(esp = esp - distance)$$

Figure 7 shows the structure of Call statement. Certain key data in RuleCall consist of special values in the static ROP construction process and are filled with concrete values during the dynamic analysis. These data include the present module pointer callPtr, target module pointer enterPtr, and distance between the present and the target module pointers. The module calling process will first realize the inter-module parameter transfer, and args represents the parameter list of the incoming target module. In the ROPL, two types of parameters can be passed as variables and constants. Then,

Figure 7. Structure of call statement



the AGROP calculates the distance and implement the ROP module switching by changing the value of the esp register.

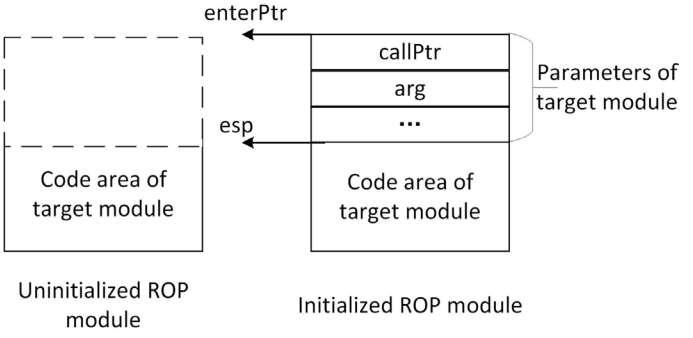
(2) The rule for translating the Enter statement to intermediate instruction sequence is presented in $Rule_{Enter}$:

$$Rule_{Enter}:$$
$$Rawhex(callPtr)$$
$$Rawhex(args)$$

The Enter statement is used to initialize the memory space of the return address and the incoming parameters of the target module. The callPtr represents the return address of the target module, and args is the argument list of the incoming target module. The memory layout of the target module after initializing the Enter statement is similar to the layout displayed in Figure 8.

(3) The rule for translating the Ret statement to intermediate instruction sequence is presented in $Rule_{Ret}$:

Figure 8. Layout of target module after the Enter instruction is executed



$Rule_{Ret}$:

$$BinO(distance = callPtr - retPtr)$$
$$OpStack(esp = esp + distance)$$

In the new rule, the Ret statement reads the return destination address callPtr, and the AGROP calculates the distance between the current pointer retPtr and callPtr. The module switches back to the previous module by modifying the value of the stack pointer register esp. Figure 9 illustrates the process of a module returning call by using the Ret statement.

Dynamic Memory Analysis

In Figure 2, the ROP chain consists of a sequence of gadgets in a specific order. The target address and operand of the gadget must be stored in the program's stack because each gadget ends with the Ret instruction to prevent the source program from jumping to the address of the next gadget. The stack has a sufficient controllable space for the ROP and can determine whether the ROP is suitable for the source program when the source program is in the control-flow hijacking state (i.e., the value in the instruction register EIP is a symbol value). Therefore, the AGROP initially analyzes the source program memory state for the ROP layout.

The key data must be collected in the memory analysis of the AGROP and are defined as follows:

symbolicBlock< symbolicAddr, symbolicSize >: This map records the information of all symbolic memory areas, except the present stack frame of the initial control-flow hijacking state, where symbolicAddr represents the starting address of the area, and symbolicSize represents the length of the area.

stackPtr: This function indicates the current stack pointer at the first control-flow hijacking state.

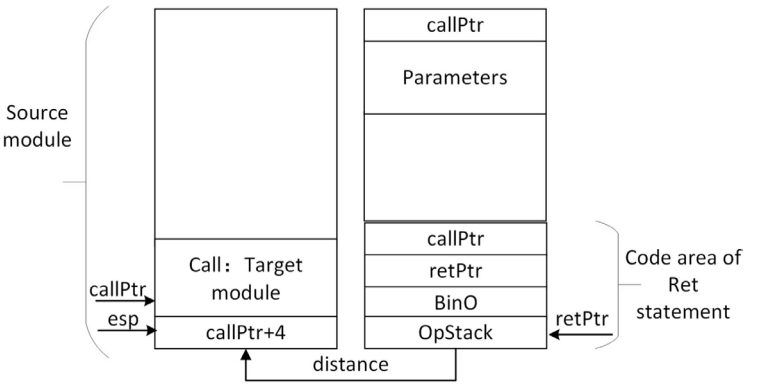
stack_symbolicLength: If the top position of the stack is in the symbolic area, then this value represents the length of a continuous symbolic memory starting from stack_ptr.

$\mathbf{mLen_{id}}$: This function indicates the memory length occupied by the ROP module called id. The length of each module is recorded in the symbol table.

If the controllable space of the current stack frame is insufficient to satisfy the layout requirements of the ROP, then the memory analysis algorithm will continue to search and record the remaining symbolic area information in other memory blocks. The process is expressed in Algorithm 1.

```
Algorithm 1. Searching for symbolic memory blocks
Input: Memory space of source program memory
Output: All symbolic memory area memSet
foreach byte  ∈ memory
if  byte is symbolic value
    if  previous byte is symbolic value
        symbolicAddr ← address of byte
        symbolicSize ← 1
    end if
    else
        symbolicSize ← symbolicSize + 1
    end else
end if
if  byte is not a symbolic value
    if  previous byte is symbolic value
        insert symbolicBlock to memSet
    end if
end if
end foreach
```

Figure 9. Return process by executing the Ret statement

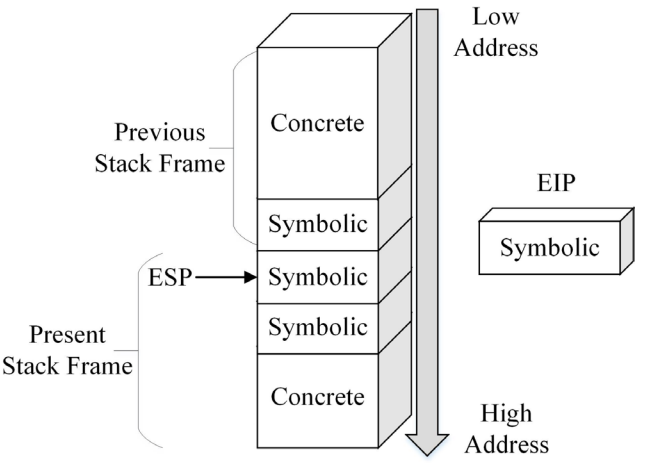


In general overflow vulnerabilities (e.g., stack overflow), the starting address of the tainted data that covers the top of the current stack is typically located in the last function stack frame. However, in implementing a gadget sequence, the initial control-flow hijacking state of the program and the stack pointer are the conditions that must be addressed. Therefore, the AGROP will check the symbolic data from the top of the current stack at the time of the first control-flow hijacking. If the data at the top of the stack is not symbolic, then the source program cannot jump to the second gadget, and the memory analysis algorithm exists. If the data at the top of the stack is symbolic, then the length of the symbolic area starting from the top of the stack is calculated. Figure 10 depicts the stack structure of the initial control-flow hijacking, which satisfies the ROP layout conditions.

Fragmented Layout of ROP

The AGROP searches for an element that satisfies the ROP module layout conditions from the set of symbolicBlocks. The candidate symbolic area length symbolicSize must be able to accommodate at least one ROP module, and the range composition in this area must not conflict with the top pointer of the stack in the initial control-flow hijacking. The AGROP performs the first round of filtering on the elements in the set symbolicBlock, selects several symbolicBlock elements that conform to the

Figure 10. Structure of the stack of the initial control-flow hijacking



length condition of the ROP module, and adds these elements to the set of candidate regions. The set of candidate regions is defined in Formula (2), where lenBlocks represents a set of candidate regions whose ROP module name is id.

$$lenBlocks_{id}: \\ (symbolicSize > mLen_{id}) \wedge [(stackPtr < symbolicAdd) \vee \\ (stackPtr > symbolicAdd + symbolicSize)] \tag{2}$$

A second round of filtering is performed on the set lenBlocks. Then, the set of conBlocks for candidate regions is defined in Formula (3).

$$conBlocks_{id}: \\ (areaLen \geq ropModule_{id}.Size) \wedge Eq(area, module_{id}) = true, \tag{3}$$

The area represents the continuous controllable memory area in the candidate area block; area. add, areaLen, and area.dataConstraint correspond to the start address, length, and data controllability constraints of the area; moduleid.Size represents the length of the ROP module id; and moduleid. dataConstraint represents the data constraint of the module. Constraint conditions A and B are compatible when the constraint comparison function Eq(A, B) returns true. However, A and B are incompatible when the function is false. The relationship between the compatibility judgment result of the $module_{id}$.dataConstraint and the area.dataConstraint and the executable ROP module id are expressed in Formula (4).

$$Eq(area.dataConstraint, module_{id}.dataConstraint) = true \tag{4}$$

The compatibility comparison process for area.dataConstraint and moduleid.dataConstraint is presented in Algorithm 2. The algorithm generates the module data constraint mConstraint.

```
Algorithm 2. Construct Module Data Constraint
Input: area, module
Output: mConstraint, isAvailable
foreach mByte  in module
foreach aByte  in  area
  if  mByte and aByte are compatible and bytes in area < bytes in
module
     mConstraint ←mConstraint ∧ mByte
  end if
  else
     mConstraint ← false
  end else
end foreach
isAvailable ← solve(mConstraint )
```

In Algorithm 2, if the isAvailable is true, then the area satisfies the layout condition of the ROP module id. A set of symbolicBlocks for the other controllable memory area is reconstructed after the AGROP marks the area as an uncontrollable area. For the remaining ROP modules, the first and

second rounds of filtering are repeated until all ROP modules are allocated for the controllable areas. This process is presented in Algorithm 3.

```
Algorithm 3 Construct ROP Data Constraint
Input : ropModules, symbolicBlock
Output : ropConstraint
foreach module in ropModules
lenBlock ← The first round of filter(ropModule, memSet)
conBlock ← The second round of filter (ropModule, lenBlock)
foreach  block  in  conBlock
    (isAvailable, newConstraint) ← Execute Algorithm 2
    if  isAvailable == true
         mark the block as concrete block
        ropConstraint ← ropConstraint ∧ newConstraint
    end if
    memSet ← Execute Algorithm 1
end foreach
end foreach
```

The ROP data constraint ropConstraint is solved using the constraint solver to generate fragmented ROP test cases. Figure 11 demonstrates the execution of a fragmented multi-module ROP.

EXPERIMENTAL RESULT

Space Efficiency of Module Switching in The AGROP

This paper selects the ROP static analysis and automatic generation tool ROPC to verify the effectiveness of the statement translation rule in improving the ROP space efficiency in the multi-module ROP switching process (); since the ROPC is also based on the Q framework for comparison.

Figure 11. Execution of a fragmented multi-modules ROP

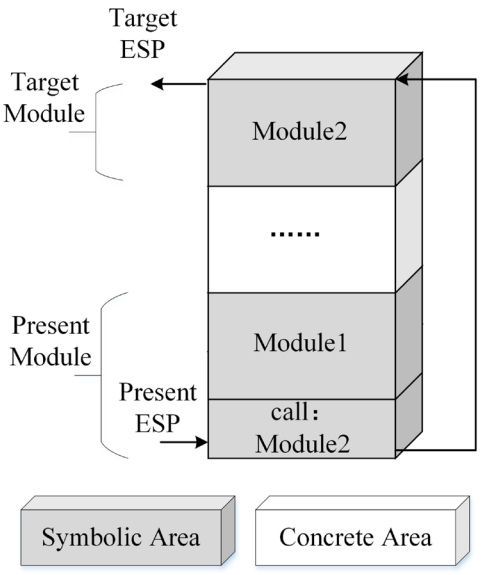


Figure 12 exhibits the amount of memory occupied by the Call function during module switching. In this figure, the abscissa is the number of parameters that must be passed in the module calling, and the ordinate is the number of bytes of memory occupied by the ROP.

According to the data in Figure 12, the gap between the memory spaces of the Call statement translated using the AGROP and the ROPC rules increases with the number of module parameters. This difference is caused by the method of modifying the stack pointers during the construction of the ROP chain between the AGROP and the ROPC rules.

The ROPC rule constructs the ROP chain on the basis of the static analysis. The ROPC must record the values of the top and bottom pointers of the stack during the calling of the module by using the Call statement, and the stack pointers are updated in a static environment during the transfer of each parameter. Under this rule, the relationship between the number of bytes required and the number of paras must be passed, as expressed in Formula (5).

$$bytes = 80 \times paras + 76 \tag{5}$$

The AGROP rule implements the ROP chain construction on the basis of the static instruction sequence construction and dynamic data filling. Key data involved in the module calling, such as target module address, top stack pointer, and bottom stack pointer, are filled with special characters in the ROP instruction sequence. Then, the key values are obtained to replace the placeholder characters by inputting the ROP into the source program of the symbol execution. This procedure omits the steps of statically analyzing the key data and improves the space efficiency of the ROP module calling. With AGROP rule, the relationship between the number of bytes and the number of paras is defined in Formula (6).

$$bytes = 20 \times paras + 44 \tag{6}$$

Figure 13 displays the comparison of the memory spaces occupied by the Enter function under the ROPC and AGROP rules.

According to the data presented in Figure 13, the AGROP rule exerts advantages in space efficiency over the ROPC rule when the number of parameters is less than 10 given the difference in the initial operation of the target module of the two rules.

Figure 12. Comparison of the lengths of Call statement in the memory between ROPC and AGROP

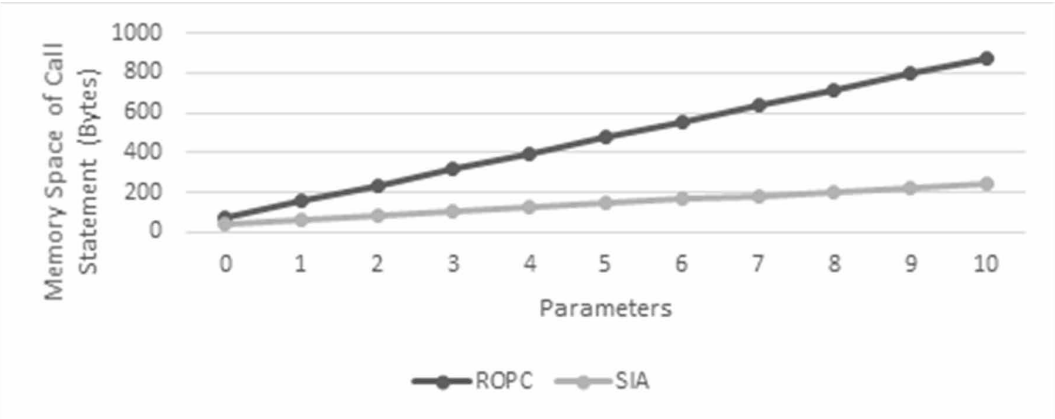
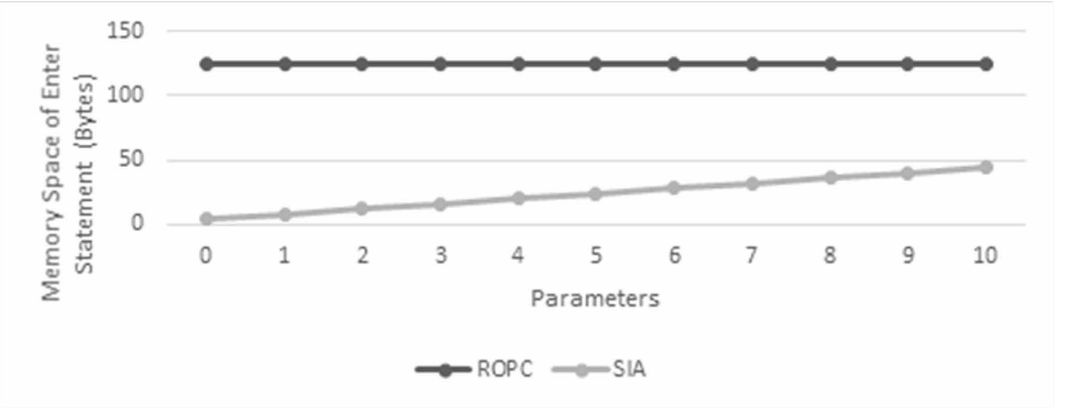


Figure 13. Comparison of the lengths of the Enter statement in the memory between ROPC and AGROP



ROPC initializes the target module and updates the stack pointers. The layout of each ROP module in the memory must be adjacent because the ROPC disregards the concept of the multi-module ROP fragmented layout. The ROPC can obtain the concrete values for parameters through a direct memory address location. Therefore, the space occupied by the Enter statement in the ROPC is constant at 124 bytes.

Similar to the Call construction, the Enter statement in the AGROP does not require an update to the stack pointers during the static construction of ROP. Owing to the influence of the ROP fragmented layout, the AGROP must initialize the target module space and set the placeholder characters for the module return address and passed parameters. According to the AGROP rule, the relationship between the bytes and the paras is expressed in Formula (7):

$$bytes = 4 \times paras + 4 \tag{7}$$

For the translation of the Ret statement, the ROPC and AGROP rules exclude the steps for inter-module parameter transfer. The memory spaces of Ret in the ROPC and AGROP are 60 and 28 bytes, respectively.

Figure 14 illustrates the comparison of the memory spaces required for the AGROP and ROPC to implement a complete ROP module switching (except for the main module).

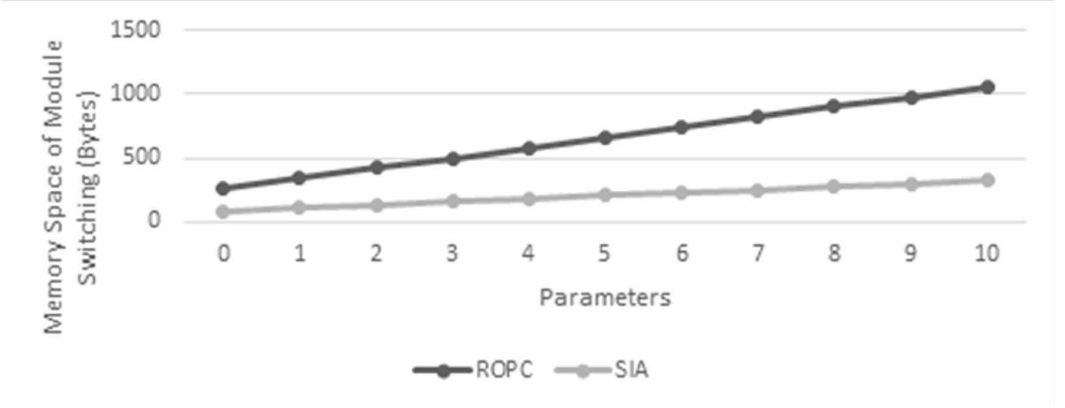
According to Definition 3 and Figure 14, this paper assumes that the number of parameters to be passed during the module switching is n, and the space efficiency of the ROP module switching under the ROPC rule is:

$$S_{ROPC\_switching} = \frac{3}{80 \times n + 260}$$

The space efficiency of the ROP module switching under the AGROP rule is:

$$S_{AGROP\_switching} = \frac{3}{24 \times n + 76}$$

Figure 14. Comparison of the lengths of module switching between the ROPC and the AGROP



The ROP space efficiency ratio of the module switching between AGROP and ROPC when the parameter number n≤10 is:

$$\frac{S_{AGROP\_switching}}{S_{ROPC\_switching}} \approx 3.4$$

According to this result, the space efficiency is approximately 3.4 times higher in the AGROP than in the ROPC. Therefore, the length of memory required by the ROP chain constructed using the AGROP rule is only approximately 29.2% of the ROPC when executing the same ROP module switching.

Analysis of The Multi-Module ROP Fragmented Layout

```
Code 4
function main( )
f1( )
function f1( )
foo( )
function foo( )
x = 1
```

The ROPL program is used in Code 4 as the target program. The symbol table generated by the AGROP through the semantic analysis of the target program is presented in Code 5.

Code 5

```
0x00  Enter(0): main
0x04  Call( f1, args:)
0x30  Ret(main)
0x4C  Enter(0): f1
0x50  Call(foo, args:)
0x7C  Ret(f1)
0x98  Enter(0): foo
0x9C  Assign(x, Const 1)
0xC4  Ret(foo)
0xE0  Global_end
```

The length of the ROP chain generated for the target program in Code 5 is 224 (0xE0) bytes. This paper selects seven source programs with control-flow hijacking vulnerabilities for experimental verification to verify the effectiveness of the fragmented layout ROP constructed using the AGROP rule in the actual program. The AGROP triggers the control-flow hijacking state of the source program through the crash file. This paper analyzes the memory state of the initial control-flow hijacking state. Table 4 displays the controllable tainted data in each memory area.

In Table 4, the lengths of the tainted data introduced in several programs are not exactly equal to the lengths of the controllable data in each part of the memory during the initial control-flow hijacking state. The main reasons for this condition are as follows:

(1) The tainted data is propagated to other memory areas, thereby causing the data constraints of certain tainted data to overlap. The vulnerability that falls into this category includes the CVE-2014-9707. This vulnerability is a heap overflow that causes a target memory to be overwritten by modifying the chunk pointer when the chunk is free. In particular, the symbolic values that overwrite the target memory are tainted by the symbolic value in the chunk.
(2) The tainted data cover key data in other memory segments. The vulnerabilities that fall into this category include CVE-2014-0322, CVE-2014-6332, and CVE-2015-5122. Arbitrary memory reads and writes are performed by reading and writing arrays across boundaries, thus leading to an overwritten function address and program control-flow hijacking. The function address is not within the scope of our analysis of the ROP layout because the function address is in the code segment.
(3) The loss of symbolic or tainted data is uncontrollable. The vulnerabilities that fall into this category include CVE-2010-3333 and CVE-2014-9707.

CVE-2010-3333 is a stack overflow vulnerability, but the controllable tainted data in the stack frame are only in the range of 16 bytes down from the top of the stack. Furthermore, this paper does not perform statistics and analysis for uncontrollable tainted data.

In the process of copying the chunk data of CVE-2014-9707, compression mapping of data based on memory addresses results in the loss of tainted data.

Based on the analysis of the controllable tainted data of the source program, the layout of each module in the memory space after the ROP chain constructed for Code 4 is fragmented is listed in Table 5.

The stack of MS06-055 only satisfies the layout condition of the main module. Numerous heap blocks in the source program are arranged through the heap spray, and the data written in the heap

Table 4. Layout of the controllable tainted data in the memory of the initial control-flow hijacking

Vulnerability	Stack*	Heap	Initial Length of Symbolic Data
MS06-055	92 Bytes	200M Bytes	200M Bytes
CVE-2010-3333	16 Bytes	100M Bytes	400 Bytes
CVE-2012-0158	400 Bytes	0 Bytes	400 Bytes
CVE-2014-0322	4 Bytes	380M Bytes	4608 Bytes
CVE-2014-9707	4 Bytes	2000 Bytes	2048 Bytes
CVE-2015-5119	4 Bytes	380M Bytes	2000 Bytes
CVE-2015-5122	4Bytes	380M Bytes	4608 Bytes

Table 5. Layout of the fragmented ROP chain

Vulnerability	main	f1	foo
MS06-055	Stack	Heap	Heap
CVE-2010-3333	Heap	Heap	Heap
CVE-2012-0158	Stack	Stack	Stack
CVE-2014-0322	Stack	Heap	Heap
CVE-2014-6332	Stack	Stack	Stack
CVE-2014-9707	--	--	--
CVE-2015-5122	Stack	Heap	Heap

block is marked as tainted data. The heap memory satisfies the layout conditions of the f1 and foo modules through memory analysis.

The stack of CVE-2010-3333 does not satisfy the layout condition of any module. This paper manually adjusted the vulnerability experiment and placed simplified stack forgery instructions in its stack memory; thus, its stack pointer will point to the specified data segment. The controllable area in the heap satisfies the layout conditions of all modules in Code 4 through the analysis.

The stack layout of CVE-2012-0158 and CVE-2014-6332 satisfies the layout conditions of all the modules of Code 4; thus, no modules are arranged in other memory areas.

The f1 and foo modules are arranged in the heap to verify the fragmented layout method and compare them with the layouts of the two vulnerabilities, although the stacks of CVE-2014-0322 and CVE-2015-5122 also satisfy the layout conditions of all modules in Code 4. The experimental results confirm that the memory of CVE-2014-0322 and CVE-2015-5122 satisfies the layout conditions presented in Table 5.

The ROP module layout conditions of Code 4 is unsatisfied in CVE-2014-9709 because the controllable tainted data in the stack do not satisfy the layout condition of jump instructions.

CONCLUSION

In this study, a multi-module ROP automatic generation method is proposed on the basis of the fragmented layout to address the problems of low space efficiency and high memory layout requirements in the ROP generation. Based on the static construction of the ROP chain and dynamic memory analysis, this method obtains the key data, such as stack pointer, during the dynamic execution of the ROP and designs a new intermediate statement translation rule in module switching. Compared with existing technologies, the new rule omits the steps of modifying the stack pointer during the static construction of the ROP and improves the space efficiency of the ROP chain.

Simultaneously, this paper proposes a multi-module ROP fragmented layout method. This method uses an ROP module as a unit, dynamically analyzes the distribution of controllable memory blocks of the source program and realizes the fragmented layout of each module to reduce the requirements of the ROP layout conditions.

However, the multi-module AGROP method based on the fragmented layout proposed in this study also has limitations. First, the process of ROP module switching disregards the influence of ASLR. Second, the loss of tainted data during the symbolic execution affects the result of the analysis of the memory states. The reduction of the impact of these issues on the AGROP is the goal of future studies.

REFERENCES

Avgerinos, T., Rebert, A., & Brumley, D. (2012). Unleashing Mayhem on Binary Code. *IEEE Symposium on Security and Privacy*, 19, 380-394.

Brumley, D., Jager, I., & Avgerinos, T. (2011). BAP: a binary analysis platform. *International Conference on Computer Aided Verification*, 463-469. doi:10.1007/978-3-642-22110-1_37

Buchanan, E., Roemer, R., & Shacham, H. (2008). When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. *ACM Conference on Computer and Communications Security*, 27-38.

Checkoway, S., Davi, L., & Dmitrienko, A. (2010). Return-oriented programming without returns. *ACM Conference on Computer and Communications Security*, 559-572.

Chipounov, V., Kuznetsov, V., & Candea, G. (2012). The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*, *30*(1), 1–49. doi:10.1145/2110356.2110358

Executable Space Protection. (2018). Available: https://en.wikipedia.org/wiki /Executable_space_protection#Windows

Gao, Y., Zhou, A., & Liu, L. (2013). *Data-Execution Prevention Technology in Windows System*. Information Security & Communications Privacy.

He, L., & Su, P. (2016). Research Progress on automatic Exploit of Software Vulnerabilities. *China Education Network*, (z1), 46–48.

Huang, H., Lu, Y., & Liu, L. (n.d.). A research on Control-flow taint information directed symbolic execution. *Journal of University of Science and Technology of China*, *46*(1), 21-27.

Huang, S. K., Huang, M. H., & Huang, P. Y. (2012). CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations. *IEEE Sixth International Conference on Software Security and Reliability*, 78-87.

Lu, K., Zou, D., & Wen, W. (2011). Packed, Printable, and Polymorphic Return-Oriented Programming. *International Conference on Recent Advances in Intrusion Detection*, 101-120.

Prandini, M., & Ramilli, M. (2012, November). Return-Oriented Programming. *IEEE Security and Privacy*, *10*(6), 84–87. doi:10.1109/MSP.2012.152

Roemer, R., Buchanan, E., Shacham, H., & Savage, S. (2012, March). Return-Oriented Programming:Systems, Languages, and Applications. 2*012*. *ACM Transactions on Information and System Security*, *15*(1), 1–34. doi:10.1145/2133375.2133377

Sang, K. C., Avgerinos, T., & Rebert, A. (2011). *Unleashing Mayhem on Binary Code*. Security and Privacy.

Schwartz, E. J., Avgerinos, T., & Brumley, D. Q. (2011). Exploit hardening made easy. *Usenix Conference on Security*, 25-25.

Shacham, H. (2007). The Geometry of Innocent Flesh on the Bone. *ACM Conference on Computer and Communications Security*, 552-561. doi:10.1145/1315245.1315313

Shao, S., & Gao, Q. (n.d.). Progress in Research on Buffer Overflow Vulnerability Analysis Technologies. *Chinese Journal of Software*, *29*(5), 1179−1198.

Stojanovski, N., Gusev, M., & Gligoroski, D. (2007). Bypassing Data Execution Prevention on Microsoft Windows XP SP2. *International Conference on Availability, Reliability and Security*, 1222-1226. doi:10.1109/ARES.2007.54

Wei, Q., Wei T., & Wang, J. (n.d.). Evolution of Exploitation and Exploit Mitigation. *Journal of Tsinghua University*, *51*(10), 1274-1280.

Xiao, Q., Chen, Y., & Qi, L. (n.d.). Detection and analysis of size controlled heap allocation. *Qinghua Daxue Xuebao. Ziran Kexue Ban*, *55*(5), 572–578.