IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

ARG: Automatic ROP chains Generation

YUAN WEI[1], SENLIN LUO[1], JIANWEI ZHUGE[2,3], JING GAO[1], ENNAN ZHENG[1], BO LI[1], LIMIN PAN[1]

[1]Information System and Security & Countermeasures Experimental Center, Beijing Institute of Technology, Beijing, 100081, China
[2]Institute of Network Science and Cyberspace, Tsinghua University, Beijing, 100084, China
[3]Beijing National Research Center for Information Science and Technology (BNRist), Beijing, 100084, China

Corresponding author: Limin Pan (e-mail: panlimin2016@gmail.com).

This work is supported by the Beijing National Research Center for Information Science and Technology (BNRist) Network and Software Security Research Program under Grant No. BNR2019TD01004.

ABSTRACT Return Oriented Programming (ROP) chains attack has been widely used to bypass Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) protection. However, the generation technology for ROP chains is still in a state of manual coding. While, current techniques for automatically generating ROP chains are still insufficiently researched and have few successful applications. On the other hand, the existing methods are based on using Intermediate Language (IL) which is in order to translate the semantics of original instructions for symbolic execution, and then fill in a predefined gadget arrangement to automatically construct a gadget list. This kind of methods may bring following problems: (1) when converting semantics of original to IL, there is a large amount of overhead time, critical instructions may be discarded; (2) the process of populating a predetermined gadget arrangement is inflexible and may fail to construct ROP chains due to address mismatching. In this paper, we propose the Automatic ROP chains Generation (ARG) which is the first fully automatic ROP chains generation tool without using IL. Tested with data from 6 open-source international Capture The Flag (CTF) competitions and 3 Common Vulnerabilities & Exposures (CVE)s, this technology successfully generated ROP chains for all of them. According to the obtained results, our technique can automatically create ROP payloads and reduce up to 80% of ROP exploit payloads. It takes only 3-5 seconds to exploit successfully, compared to manual analysis for at least 60 minutes, as well as it can effectively bypass both Write XOR Execute (W⊕X) and ASLR.

INDEX TERMS AMOCO, Automatic Exploit Generation, Return Oriented Programming, Satisfiability Modulo Theories, Z3 solver.

I. INTRODUCTION

EXPLOIT is one of the most common ways to attack the computer system. How to find and analyze the vulnerabilities quickly is the key problem of exploit. Traditional exploit is mainly generated manually, which requires not only comprehensive system underlying knowledge, including knowledge about file system, assembly language, operating system, processor architecture, etc., but also in-depth, careful studies and analysis of the attacking principles of exploit. Only in this way can we achieve the purpose of the attack.

With the emergence of defense technologies such as W⊕X [1] and ASLR [2] which make control-flow hijacks difficult to exploit, attackers turn to execute elaborately chosen machine instruction sequences that are already existing in the machine's memory. This approach allows an attacker to perform arbitrary operations on a machine which deployed defenses. Thus, in 2007, Shacham [3] proposed a new code reuse technology named Return-Oriented Programming (ROP).

Currently, ROP is the most basic and popular attack technology. However, this attack still uses manual analysis and construction, which is low efficient and high costly. In the trend that software functions are increasingly complex, and vulnerabilities are increasingly diverse, traditional approaches have been difficult to cope with these challenges. Therefore, we need to find new ways.

It is an inevitable trend to introduce automation technology into the exploit, which is called Automatic Exploit Generation (AEG) [4] [5]. AEG is rarely considered in terms of bypassing defenses, but it is still widely used due to its high efficiency and low cost in finding vulnerabilities and generating exploits. After decades of development, AEG has become a mature subject and produces a lot of practical solutions. Currently, AEG includes the following three aspects:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2937585, IEEE Access

Patch-Based Exploit Generation. Brumley et al. [6] proposed an automatic exploit generation method based on the comparison between the original program and the patched program. Automatic Patch-Based Exploit Generation (APEG) is the first attempt to generate exploit automatically. Although the idea is relatively simple, it has strong operability, which can cause the collapse of the original program while the control-flow hijacking does not succeed.

Data-Oriented Exploit Generation. Hu et al. [7] proposed FlowStitch, an automatic generation method for data-flow exploit. Although the generated samples cannot directly execute any malicious code, it is still of great practical value because it can leak sensitive data on the target host. Later, Hu et al. [8] proposed a Data-Oriented Programming (DOP). At the same time, they presented a method based on data-flow attacks code blocks and instruction scheduling allocation code blocks for the actual application. The results indicated that DOP is Turing-complete, can execute the arbitrary function, and also can be used to bypass both W⊕X and ASLR.

Control-flow Oriented Exploit Generation. Most of the exploits are control-flow hijacking attacks which have become the most widespread and popular attacks today. Presently, the work has been able to automatically generate exploits under certain constraints, but there are still many limitations. ROP is one of the most popular ways of control-flow hijacking attacks [9] [10]. Schwartz et al. [11] showed a highly reliable automatically generating ROP chains method named Q. Q is by far the most classical technique and accepted widely. The key idea is to collect the gadgets in the target program and then automatically generate ROP chains [12]. The steps to Q's approach are as follows:

1. Q finds special gadgets from unrandomized binary or library file;

2. The gadgets are translated into the semantics of instruction sequences through Q's IL — QooL;

3. The above gadgets are assigned in the gadget arrangements that Q generated. Finally, Q builds ROP payloads.

Q hardened 9 real-world Linux and Windows exploits, enabling attackers to automatically bypass defenses deployed by the industry for those programs. Q proved that defenses as currently deployed can be bypassed with new techniques for automatically creating ROP payloads from small amounts of unrandomized code.

The AEG method represented by the Q scheme, that has high ability to solve the problem of ROP chains generation and is widely used, but the problem of the IL has consequences [13] [14]. For example, meaningful instructions may be discarded in the process of being converted into IL, which leads to a small number of available instructions. Moreover, it reduces the performance of generating ROP chains, which directly causes control-flow hijacking failure because no critical instructions can be found. This problem exists in AEG. In addition, Q has been basically mature and used for commercial purposes but is not open source. Q also has some limitions. Firstly, Q still use the ROP payload that ends with the *ret* instruction. Secondly, Q focuses on practical exploitation instead of Turing-completeness. Thirdly, although Q can automatically generate ROP chains, it uses the method of gadget arrangement rather than the method of constraint solving, which is not smart enough.

In this paper, we propose a new technology — ARG, which efficiently combines the generation capability of AEG with the ability of ROP chains to bypass defenses, and uses AEG to increase the running speed and reduce the cost of program analysis. In other words, we introduce ROP into AEG to bypass both W⊕X and ASLR. ARG mainly aims at the long time-consuming and a large number of available instructions discarded in the existing process of automatically generating ROP chains. Moreover, its strong compatibility and usability, can effectively reduce ROP chains coding and provide Python interfaces that analysts can directly call. In addition, ARG breaks the limitation of ROP exploits payloads ending with *ret* instructions, which is Turing-complete. Through 9 real-life vulnerabilities on multi-processor architectures, such as i386, AMD64, ARM, MIPS, we show that we can automatically bypass existing defenses. In particular, ARG only takes 3-5 seconds to successfully exploit, compared to manual analysis for at least 60 minutes, where Q did not achieve our effeciency. In this way, it solves the very complex and difficult problem that is for professionals to exploit a vulnerability. The demo and source code have been released [15] [16]. The main contributions of this paper are summarized below:

ARG is based on the characteristics of the Directed Acyclic Graph (DAG), and uses symbolic execution technique for automatically generating ROP chains. It can successfully bypass DEP and ASLR protection. ARG symbolizes instructions solves the preconditions of constraint solving without using IL. Moreover, this method can transform the instructions into an algebraic model, which greatly improves the available number of gadgets. In addition, ARG first attempts to automate the link gadget by using DAG, and uses the topological sorting method to avoid overwritten of register value, which is the phenomenon of side effects in Q when the register is reused. In order to improve the efficiency of automatically generating ROP chains, ARG uses the Z3 solver to solve the ROP chain algebra model.

II. RELATED WORK AND BACKGROUND KNOWLEDGE

This section provides background and state-of-art about ARG and ROP.

A. ROP

ROP is a system security exploit technology, where through it, an attacker can control the execution stack to complete control-flow hijacking, and then executes small computer instruction sequences which already exist in the computer memory, called gadgets [17]. By combining these gadgets, an attacker can execute arbitrary malicious code on a computer that uses defenses. Each gadget usually ends with a return instruction. Fig. 1 shows the model of ROP payload, where the value indicates the data to be controlled, and the *ret*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2937585, IEEE Access

indicates the jump to the next instruction. This technology was first proposed by Solar Designer [18] in 1997, and was later extended to unlimited chaining of function calls. Our work is to make the ROP chains automatically generated through the computer.

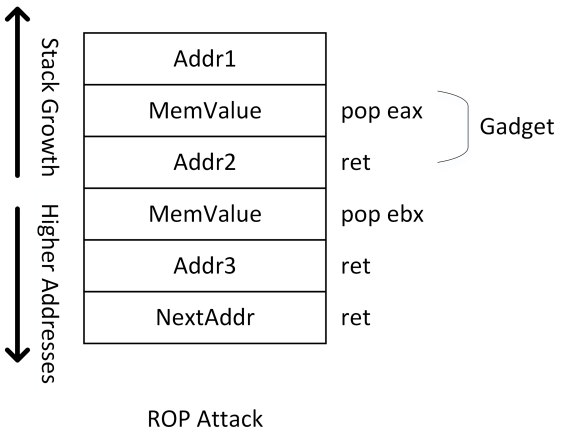


FIGURE 1. The memory layout of the ROP attack.

B. W⊕X AND ASLR

W⊕X is a memory protection policy in operating systems, which means that each page in the memory address space may be either writable or executable, but not both. Without W⊕X protection, the program can write data in the memory area and then run the data. ASLR is also a system security technology involved in protection of preventing buffer overflow attacks. As a security patch for Linux systems, ASLR was first presented in 2001 but was not widely used until 2007 [19]. It randomly arranges the address space positions of key data areas of a process, including the positions of the stack, heap, libraries and the base of the executable, in order to prevent an attacker from reliably jumping to, such as a particular exploited function in memory. In addition, ASLR does not randomize all address spaces, because it causes unnecessary system performance overhead and makes it harder to predict target addresses. In this paper, ASLR refers to randomizing the start address of dynamically linked library, which usually means randomization of default settings. Currently, modern operating systems use W⊕X and ASLR together to prevent control-flow hijacking [20]. This is also two system protections that are mainly bypassed in this paper.

C. AMOCO

AMOCO [21] [22] is a Python package dedicated to the static analysis of binary files. At the same time, it is a common framework for decoding instructions, and is designed to reduce the time required to implement new architectures support. Moreover, AMOCO can calculate the functional representation of instruction blocks and describe the semantics of each instruction. It also can provide an abstract memory model to transparently handle concrete or symbolic values, as well as other system-dependent functions. In addition, various classes in AMOCO are used to implement techniques, such as recursive traversal, linear sweep, path-predicate, etc., which relies on Satisfiability (SAT) / Satisfiability Modulo Theories (SMT) solvers to discover the control flow diagram. In this paper, AMOCO plays a key role because it replaces the IL for symbolization, which is the first time to introduce the automated construction ROP chains. Compared with IL, symbolizing instructions by AMOCO has no side effects and increases the efficiency of automation greatly. We will describe in detail in the section of analyzing gadgets automatically.

D. DAG AND TOPOLOGICAL SORTING

DAG is a finite directed graph without directed cycles. It consists of a finite number of vertices and edges, with each edge directed from one vertex to another. Topological sorting is a topological ordering of a directed graph, and a linear ordering of its vertices. For each directed edge UV from vertex U to vertex V, U is ahead of V in the sorting. DAG is used to construct the delivery relationship between gadgets, and multiple gadget's connections are key part of the rop chain. However, topological sorting can guarantee the correct sequence of gadgets. In order to address the automated combination of gadgets, this paper is also the first time introducing these two techniques for the automated generation of ROP chains, and it will be explained in the next section.

E. SMT AND Z3

SMT [23] is a decision problem of logical formulas, which is a combinations of background theory expressed in classical first-order logic and equation. Moreover, SMT can be considered as a form of constraint satisfaction problem, and therefore as a certain formal method of constraint programming. Z3 [24] is a new and efficient SMT solver provides free of charge by Microsoft Research. It is used to solve problems in various software verification and analysis applications, so it integrates support for various theories. We choose Z3 to complete the automated solution, and the specific approach will be introduced in the next section.

F. CONTROL-FLOW ORIENTED EXPLOIT GENERATION

The main idea of Control-flow Oriented Exploit Generation is analysis programs based on binary analysis techniques, such as program verification, dynamic taint analysis [25], and concolic execution [26]. It constructs an execution path that hijacks program control flow by inputs, thereby allowing an attacker to run arbitrary code. This technique is mainly used to detect the exploitability of vulnerabilities and generating exploits. Currently, in this aspect, the related work has already been able to automatically generate exploits under certain constraints. In 2016, the program called Mayhem [27] automatically defended against cyber attacks. It is a fully autonomous system that combines online and offline execution to find and fix system security vulnerabilities. Moreover, Mayhem introduces an index-based memory model as a prac-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2937585, IEEE Access

tical method to handle symbolic memory loads. In addition, Wang et al. [28] proposed a PolyAEG, a complete system that automatically generates multiple exploits for control-flow hijacking vulnerabilities. It can diversify the combination of different trampoline instructions and shellcode to generate polymorphic exploits, and it is able to identify all possible hijacking points. Although the performance of Mayhem has improved, most of Mayhem's work is focused on exploitable bug finding. Also, Mayhem makes no effort to bypass OS defenses such as DEP and ASLR, which will likely protect systems against exploits Mayhem generate. PolyAEG is based on Qemu [29], which has an interruption in tracking and passing values. The effect of PolyAEG is not ideal and some exploits may fail. It can produce exploits only under specified conditions, and makes limited effort to bypass DEP and ASLR. ARG is good at bypassing system defenses and there are no interruptions when using the simulator.

Currently, AEG has gained great achievement in the field of automatically finding vulnerabilities and generating exploits, but it is much less concerned with bypassing defenses. We combine efficient generation capability of AEG with the ability of ROP chains to bypass defenses in order to help security personnel to improve the efficiency of the analysis.

III. ARCHITECTURE DESIGN AND IMPLEMENTATION

A. ARCHITECTURE OVERVIEW

This paper proposes a new technology that bypasses DEP and W⊕X protection to automatically generate ROP chains — ARG. Fig. 2 shows the end-to-end workflow of ARG. Firstly, ARG finds available gadgets as input variables. Secondly, it uses AMOCO to analyze the semantics of the input (available gadget set). Thirdly, in order to construct DAG to extract the transitive relation between registers, it traverses the DAG tree to find all reachable paths, and then uses Z3 to back-calculate these paths. Finally, ROP chains are automatically generated. In the following, we will introduce each step of ARG in detail.

B. DISCOVERING GADGETS AUTOMATICALLY

The first step is to discover gadgets automatically. Instead of using ROPgadget tool [30] to extract the gadget directly, we take the search algorithm in ROPgadget and redevelop it to meet our needs. The principle of extracting the gadget is shown in Algorithm 1. We define the discovery rules for gadgets, filter them according to these rules, and then store them in Zope Object Database (ZODB) [31] after preprocessing. The number of gadgets here is huge. It is related to the size of the program, where the larger the program is, the more gadgets it contains.

When extracting gadgets automatically, we compared the performance of three different tools in searching for gadgets. Table 1 shows the performance of the three tools. It can be seen that the overall performance of the ROPgadget is better than others. ROPgadget can find almost as many gadgets as Binary Analysis and Reverse engineering Framework (BARF) [32], but the time cost is much lower than BARF. Al-

Algorithm 1 Extract Gadget

Input: Binary File
Output: Gadgets

```
MAX_SIZE = 200
if (data_len >= MAX_SIZE * 1000) then
    NeedFilter = True
end if
for Segment to BinaryExecutableSegments(Binary) do
    gadgets←FindAllGadgetsMultiProcess(Segment,Binary)
end for
if NeedFilter then
    gadgets ← Simplify(gadgets)
end if
gadgets ← DelDuplicate(gadgets)
```

though the overall performance of ROPgadget is superior, the analysts cannot stand such a long-time searches for gadgets in large-sized programs. Therefore, we must solve the following problems. (1) It takes a long time to extract gadgets, and searching for gadgets in large-sized programs makes the time longer. Even a byte-by-byte fast search algorithm cannot completely solve the problem of costing a long time. In response to this problem, we proposed a method for multi-process searching gadgets and only searching for segments with executable permissions. This approach significantly reduces the search time by introducing a process pool and iteration parameter. The number of generated processes is related to the number of executable segments in the libc and source programs. (2) Table 1 shows that 24,812 available gadgets are found in the 873k libc. These gadgets have a huge performance overhead when generating DAG. We remove duplicate gadgets and simplify the storage format of gadgets in order to improve performance. In addition, we set a limit that when the size of stored gadgets is greater than 200k, ARG will immediately set *need_filter* to reduce performance loss.

TABLE 1. Comparison of ROP-tool, ROPgadget, and BARFgadget.

Techniques	ROP-tool	ROPgadget	BARFgadget
Time	21.78s	91.13s	703.816s
Amount	1,229	21,282	24,812

C. ANALYZING GADGETS AUTOMATICALLY

Automated analysis of gadgets is an important part in the process. In order to implement the automated analysis of the semantics of gadgets, we need to classify and symbolize these gadgets.

Table 2 shows the types of gadgets. We define five types of gadgets: move register, arithmetic, load memory, store memory, do nothing, and undefined. Through the mutual combination of gadgets, which can implement basic data transmission, arithmetic operation, logical operation, branch statement, system call, and function call. Different types

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2937585, IEEE Access

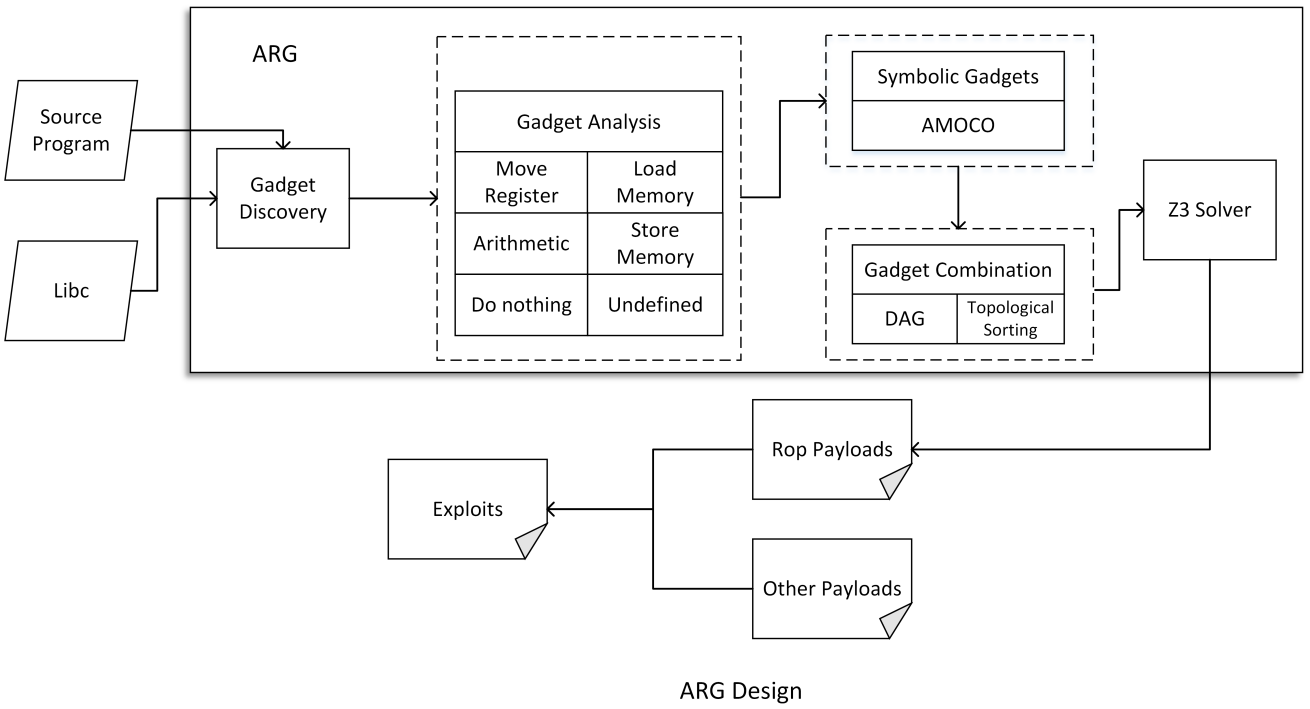


FIGURE 2. ARG design.

of gadgets are available in library functions and programs. These types of gadgets are combined with each other in accordance with the Turing-completeness that proposed by Shacham. We store these instructions by category for the next step.

TABLE 2. Gadgets category.

Type	Gadget
Move Register	mov eax, edx ; ret
Arithmetic	add ebx, esi ; ret
Load Memory	pop rbp ; ret
Store Memory	mov dword ptr [rdx], rax ; ret
Do nothing	nop
Undefined	other types

Symbolizing gadgets is a necessary precondition for automated analysis of the semantics of gadgets which is in order to describe the gadgets as a formal language. When these instructions are symbolized, exploits are turned into a formal verification problem, and the result is a set of constraints. Then the instructions use a constraint solver to obtain an answer that satisfies these conditions. This answer is usually called payload. We propose a method for symbolizing gadgets that directly converts semantics. It does not use the popular IL to analyze the semantics of instructions.

AMOCO provides a method to represent abstract and concrete symbol values in the virtual memory space of the process. After importing AMOCO, we respectively obtain different states of CPU during the execution of the gadget, and then build the mathematical model. Our method for symbolizing gadgets can accurately show the semantics of the instruction fragments and transform the instruction or function into a symbol algebraic model. However, using IL to convert gadget discards the side-effect instructions, which means that a large number of available instructions are excluded. Before using the Z3 solver, it is necessary to symbolize the instructions to satisfy the input constraints of the SMT. There are two ways to symbolize instructions: direct translation and IL. The direct translation is very complicated and difficult, but there is no side-effect due to converting into another language. Our work can achieve this effect. ARG does not use IL, so there are no side effects due to conversion. Our goal is to get the expressions of registers or memory locations along a chosen path so that we can better study a known Control Flow Graph (CFG) of the function path.

ARG uses static analysis technique to analyze the semantics of gadgets, and symbolizes gadgets through AMOCO's static analysis technique. The principle of ARG direct translation is shown in Algorithm 2.

In natural languages, we can easily understand the meaning of the assembly instruction *pop rdi*, which means to put the data on the top of the stack into *rdi*. But how can we make the machines understand the meaning of the assembly instruction? We use the code analysis strategies implemented in AMOCO to disassemble basic blocks directly. According to the extracted semantics, the mapper chain is constructed. From this mapper object we can reconstruct the symbolic CPU state after the execution of gadget. The offset of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2937585, IEEE Access

Algorithm 2 Symbolic Gadget

Input: Gadget (e.g. *pop rdi ; ret*)

```
cpu = LoadCpu(ARCH_CPU)
code = (asm(Gadget))
p = AmocoTypeBinaryGadgets(code,cpu)
blocks = list(AmocoInstructionBlock(p))
mp = NewAmocoMapper()
for block in blocks do
    if block[Instruction] ∩ 'call' then
        NeutralizeCall()
    end if
    mp ≫= block.map
end for
```

Output:

```
rip←{|[0:64]→M64(rsp+8)|}
rsp←{|[0:64]→(rsp+0x10)|}
rdi←{|[0:64]→M64(rsp)|}
```

CPU state change is obtained by mapper's shifts operators. Finally, we can evaluate a path of gadgets, we can get the expressions of registers or memory locations along a chosen path. For example, *pop rdi*, through mapper to record the change of each register assignment, maps *rdi* to *rsp*, and records the relative offset between this position (i.e. the position of *rip*, *rdi*, the next *rsp*) and the *rsp*. In this way, we can describe *pop rdi* as a formal language for the machines to understand, and map the instruction fragments into a symbol algebraic model, of which the purpose is to satisfy the solution by symbolization.

Emulator VS Symbol Execution

Before using symbol execution, we considered using emulators to analyze gadgets. We select three emulator tools: Unicom [33], Miasm [34], and Qemu, to automatically analyze gadgets. The emulator can simulate multiple CPUs (x86, PowerPC, ARM, Sparc) on multiple hosts (x86, PowerPC, ARM, Sparc, Alpha, MIPS). By using them, we can easily analyze and debug the target binaries. However, the underlying module design of Qemu is very complicated and the Miasm converted instructions are inaccurate and complicated to develop. In addition, we also try BARF, which has a huge performance overhead and takes is too long. Although Unicom makes it relatively easy for analysts to analyze programs, in practice, Unicom only implements the simulation of CPU instructions, and if not being careful, the analysts may fall into various problems, which lead to continuous tracking and debugging. It is very disadvantageous for us to automatically generate codes. Currently, almost all emulators do not support system calls, thus, memory should be mapped and data should be written into memory manually before emulation starts at the specified address.

In addition, the emulator forwards executing process. When writing a value into memory, the emulator can only get the path that this value passes, or get the memory address that this value reaches. If a data operation occurs during this process that causes the value to be changed or the sequence to be scrambled, it is difficult for the emulator to continue analyzing. Compared with the emulator, symbol execution is easier to analyze. Moreover, the emulator requires more manual operations and is not suitable for automated analysis.

In summary, we adopted the symbol execution technique to automatically generate ROP chains. Through extensive research, we found that many tools use IL, for example: Valgrind [14] and Angr [35]. LLVM is a very classic and popular tool, while LLVMIR as its IL is also commonly used. Almost all IL have such problem: discarding any instruction sequence that might cause the program to crash. Moreover, using IL can also causes a lot of performance overhead. ARG does not use IL. It is based on AMOCO and it can directly analyze binary programs. Compared with the way of using IL, ARG can directly obtain the mapping of symbols and improve the accuracy of analyzing gadgets, while reducing the overhead of performance.

D. COMBINING GADGETS AUTOMATICALLY

The automated combination of gadgets is the core part of this paper. There are two preconditions that must be satisfied before using the symbol execution to automatically construct ROP chains: (1) describe the transitive relation of the gadget. (2) determine the order in which the gadget is executed. The purpose of this is to create constraints for the Z3 solution.

1) DAG

As we all know, gadgets are scattered and discontinuous fragments of instructions in the programs and libc libraries. The relationship between these gadgets is like a complicated fishing net. This paper proposes a method to use DAG to represent the net of the relationship between gadgets, as shown in Fig. 3.

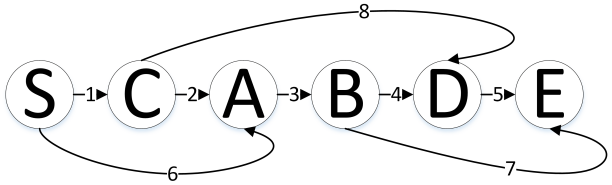


FIGURE 3. Convert gadgets to DAG. S represents the top of the stack pointer and ultimately passes the value to any register.

We define *esp* as the initial vertex with indegree 0, and gadget as the vertex of DAG, and directed edge as the representation of the transitive relation between gadgets. In this way, we can map the transitive relation of all gadgets to a DAG, as shown in Table 3, the value passes from *esp* to *ebx* via *eax*. DAG uses adjacency list to store gadgets.

For example, *pop eax ; ret* means putting the value of *esp* into *eax*, which is the line passed between registers. The second instruction, *mov ebx, eax ; ret* indicates that the value of *eax* is assigned to *ebx*. So gadget01 to gadget02 is a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2937585, IEEE Access

TABLE 3. Storage form of the DAG in the array.

Gadget	Addr Instruction	Gadget	Adjacent Gadget
Gadget01	0x1000: pop eax ; ret	Gadget01	[Gadget02]
Gadget02	0x2000: mov ebx ; eax ; ret	Gadget02	[Gadget04]
Gadget03	0x3000: pop ecx ; ret	Gadget03	
Gadget04	0x4000: mov edx ; ebx ; ret	Gadget04	

directed edge. In this way, all available gadgets are grouped into a DAG as in Fig. 3. DAG can avoid loops, and can well solve the problem of path explosion in symbol execution.

2) Topological Sorting

After building gadgets relationship net through DAG, there is another problem. If the order of the gadgets is wrong, it will overwrite the values previously set. This will directly lead to the failure of the ROP chains construction, which is fatal.

Before calling the function, we need to adjust the order of the gadgets to avoid re-overwriting the registers. As shown in Table 4, in the wrong combination, the register will be reset. After analysising, we find the key to the problem is that the general gadgets assign values to the target register all at once. But in the smaller programs, there is always a phenomenon that the gadget cannot be found. This directly leads to the failure of exploits due to lack of ROP chain components. At this time, we need a transfer station, for example, the instruction fragment *mov ebx, eax, ret;* is a transfer station, and *eax* is a transfer register. Although this move gadget can effectively solve the problem that key registers cannot be found, it brings new problems. As shown in Table 4, if the transfer *eax* is already set before running the instructions, it cannot be changed. When running *mov ebx, eax, ret;* instruction, we will disrupt the previous settings. This is undoubtedly a serious problem for the construction of ROP chains.

TABLE 4. An example of the comparison of correct and error sequence.

Assign [eax: 0x1111, ebx: 0x2222]			
0x00	0x1000: pop eax ; ret	0x00	0x1000: pop eax ; ret
0x04	0x2222	0x04	0x1111
0x08	0x2000: mov ebx, eax ; ret	0x08	0x2000: pop eax ; ret
0x0C	0x1000: pop eax ; ret	0x0C	0x2222
0x10	0x1111	0x10	0x2000: mov ebx, eax ; ret
0x14	next address	0x14	next address
Correct sequence		**Error sequence**	

Finally, we come to the conclusion that when combining gadgets. We cannot allow the post-executed gadgets to destroy the results that have already been executed. This problem mainly exists in gadgets composed of multiple instruction fragments, when registers are used a second time. Therefore, this paper introduces topological sorting to solve this problem. Topological sorting can determine the correct order, if there is a directed edge from U to V in the DAG, then U must be in front of V by using topological sorting. Moreover, this method can effectively solve the problem of re-overwriting the values that have already been set. When we assign a value to the register, we first get the path by traversing the DAG, then use topological sorting to determine the correct order of gadgets. We do the above two works to automatically combine gadgets, the effect is excellent.

In addition, the number of edges in the DAG is less than the number of vertices. Because using adjacency list representation saves space compared to using the adjacency matrix, the adjacency matrix is suitable for the dense graph, while the adjacency list is more suitable for the sparse graph. So we consider another storage structure (adjacency list), as shown in Fig. 4. We use adjacency list to store the results of topological sorting, as shown in Table 3. After completing the gadget's combination, the gadgets are stored in an array, making it easy to read/write gadgets while programming, and also more natural and performant to use.

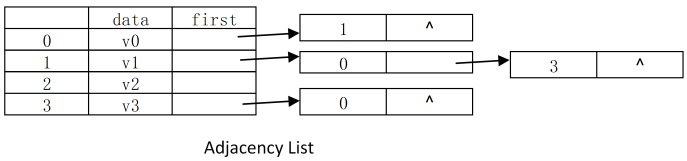


FIGURE 4. It is easy to get the mapping relationship of gadgets through the adjacency list.

E. CONSTRAINT SOLVING

Through the automated combination of gadgets, we get the transfer paths between all the registers, such as *pop rdi ; ret*, that is, we pass the data in the stack to *rdi*. Our approach is to use the constraint solver to get the results. We define a function $f(x)$ as the value of the register, where x means the position. ARG can transform the program into a logical mathematical model, that is an SMT model, from which we can obtain SMT satisfiability. Under the usual background theory, if there is an assignment that makes the formula true, the formula is satisfiable. Otherwise, the formula is unsatisfiable, and this assignment is called the model. We create an SMT model with the logical form of $A \wedge B$, which is shown in (1), and it is represented by the mathematical symbol: $\prod_{position}$, which is the position we want to calculate, that is the memory address where the data is to be written; B is represented by the mathematical symbol: $\prod_{value}$, which means the value we want to write at this position. In this formula, the true return value indicates the input is satisfiable, and the false return value indicates the input is unsatisfiable. The symbol $\wedge$ is interpreted as satisfying both $\prod_{position}$ and $\prod_{value}$. At the same time, we need to find a set of solutions that satisfy the correspondence between position and data. The result is represented by $\prod_{result}$ symbol, which is a set of the solution set, and the model of the solution set is shown in (2). In other words, the model of the solution set has a one-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2937585, IEEE Access

to-one mapping relation between position set and data set.

$$\prod_{position} \wedge \prod_{value} \Rightarrow \prod_{result} \tag{1}$$

In this section, we focus on the specific calculation process of the formula. The details of the above formula are shown as follows:

$$\begin{cases} f(n) = rsp + offset + n \\ SMT(Val ==^* Concat(f(n_1), f(n_2), f(n_3), \ldots, f(n_n))) \end{cases} \tag{2}$$

Here, the $\prod_{value}$ is interpreted as *Val*, which is a given value. The $\prod_{position}$ is interpreted as a mathematical formula: $Concat(f(n_1), f(n_2), f(n_3), \ldots, f(n_n))$, where $f(n_i)$ represents the specific position within the stack and its unit is a byte. $f(n)$ consists of three parts: *rsp*, *offset*, and *n*. The *rsp* represents the top-of-stack pointer, which is commonly referred to the *rsp* register. We use *rsp* as a base address, and all addresses that need to be calculated are related to *rsp* in the automated solution process. The *offset* represents the size of the address space occupied by the gadget, which is related to the addressing capability of the operating system. For example, a 64-bit instruction occupies 64 bits, but a 32-bit instruction occupies 32 bits. The *n* indicates the address number used to distinguish the position of the byte, which means it can automatically write the corresponding position by byte-by-byte. $Concat()$ is a connection function that concatenates all the bits together in order to get a full byte. The *rsp* and *offset* have been obtained in pre-work of constraint solving. We add two formulas to the Z3 solver, as shown in (2). If the formula is satisfiable, $check()$ returns ***sat***; while if not, $check()$ returns ***unsat***; when $check()$ cannot determine, it returns ***unknown***. Through $model()$, the solution set of the equation can be obtained. The solution set is shown as follows:

$$\begin{cases} n_1 \rightarrow Val_1 \\ n_2 \rightarrow Val_2 \\ n_3 \rightarrow Val_3 \\ \quad \vdots \\ n_n \rightarrow Val_n \end{cases} \tag{3}$$

The above solution set represents the relationship between the position and its value.

F. PROGRAM OPTIMIZATION

After completing the development of the program, we need to improve the performance of the program. We optimize the program in the following ways. Table 5 shows the time performance of the optimized ARG before solving Z3.

1) DAG build optimization

Because DAG generation consumes performance, it is necessary to optimize the generation of DAG paths, and we adopt the fastest and shortest generation method to optimize performance. Firstly, we find the smallest gadget. Secondly, if we cannot find the smallest gadget, we add a register in the intermediate pass register. For performance consideration, we set a threshold to limit the number of return paths to no more than 10. In this way, we can avoid a lot of waste of storage space and time caused by repeatedly generating paths, thus can improve the efficiency of ROP chain generation.

TABLE 5. Total time of the optimized ARG search, analysis and combination in libc and Xmms2. The First Run Time indicates the time of the first ARG run. The Second Run Time indicates the time of the second ARG run. The sizes of libc and Xmms2 are quite different, but their run time differs by less than a second because we set the threshold. When the search is beyond 200k, ARG stops discovering gadgets.

	Size	First Run Time	Second Run Time
Libc.so.6	873k	6.74s	2.95s
Xmms2	117k	5.81s	2.33s

2) Gadget search algorithm optimization

Searching for a large number of gadgets in binary files and libc wastes a lot of time, and we take two approaches for performance optimization: (1) Use multiprocess to find gadgets; (2) Delete duplicate, complex instructions, and save only the simplified instructions. The purpose is to complete the search process of gadgets in the shortest time.

3) Function templating

Since the calling function is more complicated, we simplify the use of the function and directly add the function through the form of *func_call*. For example, *rop.read(0, elf.bss(0x80))* is actually equivalent to *rop.call('read', (0, elf.bss(0x80)))*. The benefits of this approach are that reducing the lines of the script, providing a portable user interface, and simplifying the usage for analysts.

IV. IMPLEMENTATION

ARG consists of 4 major components: gadget discovery, gadget analysis, gadget combination, and constraint solving. ARG is written in Python and includes 3,231 lines of code, the amount of code is better than Q. We use cProfile[x] to measure the time of CPU and record the overhead of major functions. Taking welpwn as an example, ARG calls 3,159,428 functions, which takes only 4.841 seconds. Table 6 shows 10 functions with the most performance overhead. The ncalls indicates the number of function calls. While the tottime represents the total running time of the function, excluding the running time of the subfunction. The cumtime indicates the running time of the function and all its subfunctions, which is the time between the function call and return.

We introduce the search idea of the ROPgadget tool, redevelop it, and use the multi-process method to search for the gadgets. In addition, we choose the AMOCO to symbolize the gadgets and use the Z3 solver to solve the constraint. ARG supports i386, AMD64, ARM, and MIPS. Moreover, ARG has strong portability, where it can simplify functions and provide multiple easy-to-use interfaces.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2937585, IEEE Access

TABLE 6. ARG calls 3,159,428 functions in total, of which 3,015,093 is the original call.

ncalls	tottime	cumtime	function
38	0.657	0.657	Z3_solver_check_assumptions
79,737/1,656	0.399	1.233	_parseNoCache
15,689/10,794	0.335	0.968	core.py:634(_parse)
38	0.158	0.159	Z3_solver_assert
67,982/67,980	0.154	0.174	__init__
105,502	0.123	0.195	_read_stream
90,823	0.112	0.304	core.py:349(_parse)
206,662	0.098	0.098	container.py:40(__setitem__)
248,413/248,412	0.097	0.107	isinstance
11,255/1,656	0.091	1.217	parseImpl
3,159,428 function calls (3,015,093 primitive calls) in 4.841 seconds			

V. EVALUATION

In this section, we present experimental work on our prototype. Firstly, we evaluate the degree of automation. Secondly, we show the performance by 6 open-source international CTF projects and 3 Q's experimental data (CVE). Thirdly, we evaluate the hardening of automatically generating ROP chains whether it can effectively bypass both W⊕X and ASLR.

A. EXPERIMENTAL SETUP

We evaluated our system on 2 virtual machines running on a desktop with a 2.30GHz Intel(R) Core i5-6200 CPU and 12GB of RAM. Each VM had 4GB RAM and was running Ubuntu 16.04 Linux VM and Windows XP SP3 respectively.

B. THE DEGREE OF AUTOMATION

For generating the exploit, the degree of automation is an important evaluation in the aspects of efficiency, quality, and stability. The advantage of the prototype is that it is enough automated and intelligent. If the degree of automation is not as effective as the manual coding, then the work we have done is meaningless. Our evaluation of the degree of automation is mainly reflected in the following five aspects:

- **Preliminary test:** the prototype can support the most common types of gadget, which are the shortest and simplest types such as *pop rdi ; ret*, etc.
- **Intermediate test:** the prototype can support multiple register types, implement assignment, and passing between registers such as *pop eax ; ret ; mov ebx, eax ; ret*. This method is also the focus of this paper, and it is also the most basic purpose of this paper.
- **Advanced test:** it is not enough to only support gadget ending with *ret* instructions, we can also support other types of gadgets such as *jmp*, *call*.
- **The same origin test:** the values of both registers are derived from the same location on the stack such as *mov eax, [esp] ; pop ebx ; ret*.
- **Stack migration test:** when we construct the exploit, we frequently find that the size of the ROP chains or shellcode is too large, resulting in insufficient space on the stack. So, we need to migrate the stack to place with sufficient memory space such as *pop ebp ; ret ; leave ; ret*.

In order to demonstrate our experiment better, we select all the types of gadgets mentioned above as input and automatically generate ROP chains. Each type of gadget corresponds to a register, so that we can accurately determine whether each gadget is supported. Table 7 shows that all of the above types of gadgets can be applied, and each type of gadget can be a part of the ROP chains. Also, each register can be assigned successfully. The automated ROP chains are able to apply a variety type of gadgets and have a high ability to exploit multiple gadgets, as well as breaking the limitation of ending with *ret* instructions. This technique can bypass the flow detection for the hijacking control attacks ending with *ret* instructions well, and greatly improve the utilization of the automated ROP chains. Compared with Q, our prototype uses more types and quantities of gadgets in the automatic construction of ROP chains, with a strong degree of automation and the ability to find vulnerabilities.

C. PERFORMANCE EVALUATION

For better analysis and research, our experimental data is from 6 open-source international CTF projects and 3 CVEs in Q. CTF data is a vulnerability that extracted from real-world programs. Since Q is used for commercialization and no open-source code, we can only compare experimental results with Q, instead of using Q's code for performance comparison. Moreover, some experimental data in Q cannot be found or compiled, so we select 3 typical data in Q for performance comparison.

ARG is able to reduce all manual coding when generating ROP chains, and takes far less time than manual analysis in time performance. For example, 2015 international RCTF welpwn, which is a vulnerability program, the official code for manually writing Proof of Concept (POC) has 182 lines. However, the ARG automatically generated ROP chains have only 3 lines of code, and the overall amount of code for exploit has only 40 lines of code. In general, professionals take at least 60 minutes to manually analyze and generate ROP chains, while ARG only needs 3-5 seconds. The semantic analysis in Q takes at least tens of seconds and at most more than 300 seconds. However, ARG is much lower than Q, where mainly because our method for symbolizing gadgets does not use IL, which directly reduces a large amount of time lost in the process of converting to IL. In addition, Q uses Pin tracing [36], which symbolically executes the trace [37], obtaining the constraints. Although this can get more accurate constraints, the time will increase exponentially. In a complex real environment, if the program with a large amount of code or the exploit with complex multipath, the method represented by Q may cause the failure of automatic generation, or cause the program crash due to the infinite running time. However, the total running time of ARG is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2937585, IEEE Access

TABLE 7. We tested the types of gadgets separately and recorded the types supported by ARG. These types successfully assigned values to registers.

	Ret	Jmp,Call	Mov	Add	Xchg type	Migrate type	Same origin type
Q	Y	N	N	-	-	N	N
ARG	Y	Y	Y	Y	Y	Y	Y
Successful assignment	Y	Y	Y	Y	Y	Y	Y

only a few seconds, as shown in Table 8, because ARG has no complicated Pin tracing process. We use a mathematical model to represent the whole process of the exploit. In other words, this is also an algebraic operation, so we just need to solve the mathematical formula to get our results. Moreover, we use DAG, that can solve the path explosion problem very well [38]. When we generate a ROP subchain, the nodes associated with it are removed, thus directly avoiding the loop. Therefore, the performance of ARG is obviously superior to Q.

D. ROBUSTNESS OF EXPLOIT

We evaluate the robustness of automated generated exploit in two ways: (1) whether it can effectively bypass both W⊕X and ASLR; (2) how to handle exceptions that cannot find critical registers. From Table 8, we find that the exploit generated by ARG can bypass both W⊕X and ASLR and has the expected robustness.

The problem is that we may not find critical registers to satisfy the transitive relation. In the automated assignment of gadget, if there is no corresponding register transitive relation, the robust exploit cannot be generated by ARG. To solve this problem, we design and implement an exception handling mechanism to prompt the users that a certain type of register is not found.

In addition, we frequently encounter the problem that the official POC has a lot of fixed addresses, so POC may not be executed in a different environment. However, in this paper, the problem mentioned above does not occur, because the generated ROP chain is automatically solved by byte-by-byte, and it is written into memory once. Therefore, it is more robust against bypass defenses.

VI. DISCUSSION AND FUTURE WORK

This paper proposes and implements a new technology of automatically generating ROP chains — ARG to solve the problem of lost time and cost of manual construction. This approach can reduce up to 80% of ROP exploit payloads and take only 3-5 seconds to successfully exploit. Moreover, it not only improves the efficiency of analysis, but also can efficiently generate a large number of ROP chains code, without any manual intervention. The experimental results show that the automated construction of ROP chains is successful. We have successfully generated 9 ROP chains from 9 real experimental data (CVE). Also, all automated generated exploits can successfully bypass DEP and ASLR protection. Q is a classic paper that implements a highly reliable method of automatically generating ROP code in exploits. In the following, we will focus on the differences between ARG, Q and improvements of ARG.

• ***Ret*-less ROP**

ARG breaks the limitation of ending with only *ret* instructions in automatical generation of ROP chains. While Q requires the gadgets to end with only *ret* instructions. Compared with Q, the technique in this paper greatly broadens the types of available gadgets and improves the efficiency of gadgets. In our experiment, we show a variety of gadgets that do not end with *ret* instructions to construct ROP chains, and the results meet our expectations excellently. Currently, most security issues appear on control-flow hijacking attacks, researchers may solve this problem by monitoring the jumped gadgets to stop continuous gadgets. Therefore, researchers can improve this aspect of future work.

• **Turing-completeness**

Compared with Q, ARG takes Turing-completeness into account, which can realize basic data transmission, arithmetic operation, logical operation, branch statement, system call, and function calls. The classification of the gadgets in this paper can satisfy Turing-completeness that proposed by Shacham. Also, the iterative combination of these gadgets can achieve more purposes and functions. This paper simply shows the Turing-completeness of ARG, and we intend to improve this aspect in the future work.

• **Cache Technology**

In our approach, we use cache technology to store the gadgets that we need. We use the SHA256 [39] function to get the hash of the source program, and then extract the gadgets from it, and store these gadgets in the ZODB. ZODB is compatible with all Python data types and automatically stores gadgets based on the relationship between objects. Moreover, the data stored by ZODB has a fixed data format, and a small data storage space. When running the exploit for the second time, we can quickly find the gadget sequence, which reduces the time to re-find the gadget and improves the efficiency of the exploit.

• **DAG and Topological sorting**

This paper firstly proposes the application of DAG on the automated construction ROP chains, where the characteristics of the DAG can well describe the mapping relationship between gadgets. Moreover, DAG solves the problem of loops in the path, and avoids the problem of path explosion. At present, we only use the DAG to generate ROP chains, but we have not used it in the field of AEG to solve the problem of the ring, which may be studied in future work. After the DAG construction, there remains a problem that the data of the register is overwritten in the ROP chains. We use

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2937585, IEEE Access

TABLE 8. A list of ROP chains was generated by ARG. The time of each exploit is recorded. Reduction represents reduced ROP exploit payloads. Size(KB) represents the size of the binary file. Type presents the type of vulnerability as Remote Code Execution (RCE), Arbitrary Write (AW), Stack Buffer Overflow (SBO) and Information Leak (IL). No-eXecute (NX) can prevent malicious attacks by restricting memory pages from having both execute and write permissions, that is, isolating data and code. ARG supports multiple instruction architectures. In addition, CVE's data comes from Q. Although Q uses Intel(R) Core(TM) i7 cpu 920 @ 2.67GHz CPU, which is faster than our Intel(R) Core(TM) i5-6200U cpu @ 2.30GHz CPU, the performance of ARG is still much better than Q.

Program	Reference	Q			Time	Optimized Time	Reduction	Size (KB)	Access Complexity	Type	CPU	NX/ ASRL
		Tracing	Analysis	Total								
websrv	Github_armpwn	×	×	×	7.130s	5.996s	81%	10	High	RCE	Arm	Y/Y
trafman	Rwthctf2013	×	×	×	6.692s	2.176s	56%	10	Low	AW	Arm	Y/Y
welpwn	RCTF2015	×	×	×	4.671s	3.521s	83%	9	Medium	SBO	X86	Y/Y
pizza	CGFinals2015	×	×	×	4.282s	2.939s	88%	19	High	IL	X64	Y/Y
pwn200	SCTF2014	×	×	×	2.633s	2.301s	82%	4	Medium	SBO	X86	Y/Y
pwnme	Isg2015	×	×	×	2.718s	2.377s	66%	6	Low	SBO	X86	Y/Y
proftpd	CVE-2006-6563	30s	10s	40s	**5.265s**	**3.030s**	73%	549	Low	SBO	X86	Y/Y
rsync	CVE-2004-2093	60s	5s	65s	**4.165s**	**3.042s**	57%	975	Low	SBO	X86	Y/Y
opendchub	CVE-2010-1147	195s	30s	225s	**3.651s**	**2.746s**	78%	655	Medium	RCE	X86	Y/Y

topological sorting to determine the correct order of gadgets. It can effectively avoid the failure of generating ROP chains due to re-overwriting.

• **No IL**

At present, in the field of automated analysis and exploitation of vulnerabilities, mainstream methods generally use IL, where the use of IL produces good results. Q also uses IL, which makes it easier for analysts to interact with the system environment being exploited. However, ARG does not use IL to symbolize gadgets, it does not discard instructions, and there is no failure of ROP chains generation due to discarding critical instructions. In addition, it does not take a long time to convert IL, thus ARG can improve the efficiency of analyzing gadgets.

• **Other**

Finally, we do some work on details such as optimizing algorithms, avoiding bad characters, templating functions, and searching for gadgets in a multi-process manner. ARG supports multi-processor architectures such as i386, AMD64, ARM, and MIPS, and it improves the practicability and confrontation in the real environment. Since MIPS is less common, there is still some work left for further research.

VII. CONCLUSIONS

In this paper, we proposed and implemented a new technology — ARG, which supports automatically generating ROP chains. The proposed technique can satisfy the preconditions of constraint solving by automatically finding the available gadgets by using AMOCO analysis, DAG, topological sorting, and the Z3 solver. Instead of IL, we translated directly to symbolize gadgets, and then Z3 solver is adopted to solve random addresses and write it to the stack. The experimental results showed that ARG is able to reduce exploit payloads, and it takes only 3-5 seconds to finish control-flow hijacking, compared to manual analysis, which takes at least 60 minutes. Also, it can effectively bypass both W⊕X and ASLR. In addition, ARG has good compatibility and practicability, can support multi-processor architectures, provides multiple natural, easy-to-use interfaces, and can directly call these interfaces through pwntools. We believe that ARG will become a very popular and practical tool in the near future.

ACKNOWLEDGMENTS

This work is supported by the Beijing National Research Center for Information Science and Technology (BNRist) Network and Software Security Research Program under Grant No. BNR2019TD01004. The authors thank Purui Su at the Chinese Academy of Sciences, Yue Liu, and Chao Sang for their support of this work.

REFERENCES

[1] S. E. Friedman, D. J. Musliner, and P. K. Keller, "Methods and systems for defending against cyber-attacks," Oct. 23 2018, uS Patent App. 10/108,798.
[2] H. Marco-Gisbert and I. Ripoll, "On the effectiveness of full-aslr on 64-bit linux," 2014.
[3] H. Shacham et al., "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." in ACM conference on Computer and communications security. New York,, 2007, pp. 552–561.
[4] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Aeg: Automatic exploit generation," 2011.
[5] T. Avgerinos, S. K. Cha, B. Lim, T. Hao, and D. Brumley, "Automatic exploit generation," in Communications of the ACM. Citeseer, 2014.
[6] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in 2008 IEEE Symposium on Security and Privacy (sp 2008). IEEE, 2008, pp. 143–157.
[7] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in 24th {USENIX} Security Symposium ({USENIX} Security 15), 2015, pp. 177–192.
[8] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016, pp. 969–986.
[9] M. Prandini and M. Ramilli, "Return-oriented programming," IEEE Security & Privacy, vol. 10, no. 6, pp. 84–87, 2012.
[10] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," ACM Transactions on Information and System Security (TISSEC), vol. 15, no. 1, p. 2, 2012.
[11] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy." in USENIX Security Symposium, 2011, pp. 25–41.

[12] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in Proceedings of the 15th ACM conference on Computer and communications security. ACM, 2008, pp. 27–38.
[13] F. Cassez, A. M. Sloane, M. Roberts, M. Pigram, P. Suvanpong, and P. G. de Aledo, "Skink: Static analysis of programs in llvm intermediate representation," in International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2017, pp. 380–384.
[14] H. Obuchi, K. Ootsu, T. Ohkawa, and T. Yokota, "Efficient translation and execution method for automated parallel processing system by using valgrind," in 2015 Third International Symposium on Computing and Networking (CANDAR). IEEE, 2015, pp. 607–609.
[15] "ARG demo," https://youtu.be/S0AtN1bMb3o.
[16] "ARG source code," https://github.com/wy666444/auto_rop.
[17] A. V. Vishnyakov, "Classification of rop gadgets," Proceedings of the Institute for System Programming of the RAS, vol. 28, no. 6, pp. 27–36, 2016.
[18] S. Designer, ""return-to-libc" attack," Bugtraq, Aug, 1997.
[19] P. TEAM et al., "Address space layout randomization (aslr)," Documentation for the PaX Project. Retrieved from http://pax. grsecurity. net/docs/aslr. txt, 2001.
[20] D. J. Day and Z.-X. Zhao, "Protecting against address space layout randomisation (aslr) compromises and return-to-libc attacks using network intrusion detection systems," International Journal of Automation and Computing, vol. 8, no. 4, pp. 472–483, 2011.
[21] A. Tillequin, "Amoco," https://github.com/bdcht/amoco.
[22] P. Biondi, R. Rigo, S. Zennou, and X. Mehrenberger, "Bincat: purrfecting binary static analysis," in Symposium sur la sécurité des technologies de l'information et des communications, 2017.
[23] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in Handbook of Model Checking. Springer, 2018, pp. 305–343.
[24] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008, pp. 337–340.
[25] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software." in NDSS, vol. 5. Citeseer, 2005, pp. 3–4.
[26] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in 27th {USENIX} Security Symposium ({USENIX} Security 18), 2018, pp. 745–761.
[27] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in 2012 IEEE Symposium on Security and Privacy. IEEE, 2012, pp. 380–394.
[28] M. Wang, P. Su, Q. Li, L. Ying, Y. Yang, and D. Feng, "Automatic polymorphic exploit generation for software vulnerabilities," in International Conference on Security and Privacy in Communication Systems. Springer, 2013, pp. 216–233.
[29] F. Bellard, "Qemu, a fast and portable dynamic translator." in USENIX Annual Technical Conference, FREENIX Track, vol. 41, 2005, p. 46.
[30] J. Salwan, "Ropgadget–gadgets finder and auto-roper," 2011.
[31] J. Fulton, "Introduction to the zope object database," in Proceedings of the 8th International Python Conference, 2000.
[32] C. Heitman and I. Arce, "Barf: A multiplatform open source binary analysis and reverse engineering framework," in XX Congreso Argentino de Ciencias de la Computación (Buenos Aires, 2014), 2014.
[33] H. Dang and A. Nguyen, "Unicorn: Next generation cpu emulator framework," in The BlacNHat Conference, 2015.
[34] F. Desclaux, "Miasm: Framework de reverse engineering," Actes du SSTIC. SSTIC, 2012.
[35] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in 2017 IEEE Cybersecurity Development (SecDev). IEEE, 2017, pp. 8–9.
[36] O. Levi, "Pin-a dynamic binary instrumentation tool," 2018.
[37] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later." Commun. ACM, vol. 56, no. 2, pp. 82–90, 2013.
[38] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in NDSS, vol. 16, no. 2016, 2016, pp. 1–16.
[39] D. Rachmawati, J. Tarigan, and A. Ginting, "A comparative study of message digest 5 (md5) and sha256 algorithm," in Journal of Physics: Conference Series, vol. 978, no. 1. IOP Publishing, 2018, p. 012116.

YUAN WEI received B.E. and M.E. degrees from Beijing University of Posts and Telecommunications, Beijing China. He is currently pursuing the Ph.D. degree at the Information System and Security & Countermeasures Experimental Center, Beijing Institute of Technology. His current research interests include vulnerability detection, program analysis, and information security.

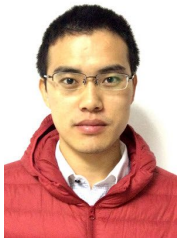
SENLIN LUO received B.E. and M.E. degrees from the College of Electrical and Electronic Engineering, Harbin University of Science and Technology, Harbin China, in 1992 and 1995, respectively, and a Ph.D. degree from the School of Information and Electronics, Beijing Institute of Technology, Beijing China, in 1998. He is currently a Deputy Director, Laboratory Director, and Professor of Information System and Security & Countermeasures Experimental Center, Beijing Institute of Technology. His current research interests include machine learning, medical data mining, and information security.

JIANWEI ZHUGE received a Ph.D. degree in Computer Science from Peking University, Beijing China. He is currently an Associate Research Professor with the Network and Information Security Laboratory, Tsinghua University, Beijing China. His current research interests include network and system security.

JING GAO received B.E. degree from Beijing Wuzi University, Beijing China and M.E. degree from Beijing University of Posts and Telecommunications, Beijing China. Her current research interests include program analysis, information security, and cloud computing.

ENNAN ZHENG received the B.M. degree in Management of Information System from University of International Relations, Beijing, China in 2015. He is currently pursuing M.E. degree in cybersecurity at UIR. He is an intern researcher at Qianxin Technology Research Institute.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2019.2937585, IEEE Access



BO LI received the B.E. degree from Zhengzhou University, Zhengzhou China, in 2013. He is currently pursuing the Ph.D. degree at the Radar & Countermeasures Technology Institute, Beijing Institute of Technology. His current research interests include signal and information processing, IoT and IoT system security.

LIMIN PAN received B.E. and M.E. degrees from the College of Electrical and Electronic Engineering, Harbin University of Science and Technology, Harbin China. She is currently working at Information System and Security Countermeasures Experimental Center, Beijing Institute of Technology. Her current research interests include machine learning, medical data mining, and information security.