Automatic Polymorphic Exploit Generation for Software Vulnerabilities

Minghua Wang[1], Purui Su[1], Qi Li[2], Lingyun Ying[1], Yi Yang[1], and Dengguo Feng[1]

[1] Laboratory of Trusted Computing and Information Assurance,
Institute of Software, Chinese Academy of Sciences, Beijing 100190, China,
{wangminghua,supurui,yly,yangyi,feng}@is.iscas.ac.cn
[2] Institute of Information Security, ETH Zurich, Switzerland,
qi.li@inf.ethz.ch

Abstract. Generating exploits from the perspective of attackers is an effective approach towards severity analysis of known vulnerabilities. However, it remains an open problem to generate even one exploit using a program binary and a known abnormal input that crashes the program, not to mention multiple exploits. To address this issue, in this paper, we propose PolyAEG, a system that automatically generates multiple exploits for a vulnerable program using one corresponding abnormal input. To generate polymorphic exploits, we fully leverage different trampoline instructions to hijack control flow and redirect it to malicious code in the execution context. We demonstrate that, given a vulnerable program and one of its abnormal inputs, our system can generate polymorphic exploits for the program. We have successfully generated control flow hijacking exploits for 8 programs in our experiment. Particularly, we have generated 4,724 exploits using only one abnormal input for IrfanView, a widely used picture viewer.

Key words: software vulnerability, dynamic taint analysis, exploit generation

1 Introduction

Software vulnerability is one of the major threats to the computer system. Exploit generation from the perspective of attackers is one of the most effective approaches for vulnerability assessment. Traditionally, exploit generation is performed manually and requires prior knowledge of the vulnerabilities. However, manually generating exploits is time-consuming and highly dependent on the experience of the analysts, and cannot satisfy the demand for vulnerability assessment and defense.

To address this issue, many exploit generation schemes have been proposed. Brumley et al [9] proposed an approach to automatically generate exploits for the potential vulnerabilities by comparing victim applications with their patched versions. Lin *et al.* [15] presented a dynamic exploit generation method by mutating a set of input values relevant to the execution of a vulnerable code location. The exploits they generated can only crash the programs so that their approaches are not able to verify whether the vulnerability is used to execute malicious code. Avgerinos *et al.* [3] proposed the first system to generate exploits containing malicious code by source code analysis and preconditioned symbolic execution. However, such approach cannot be used for the closed source software.

In this paper, we propose an automatic polymorphic exploit generation (PolyAEG) system that aims to generate polymorphic exploits containing malicious code by given program binary and an abnormal input causing it to crash. To achieve PolyAEG, the following questions need to be answered.

In order to hijack the control flow and make sure malicious code execution, (i)which input bytes should be modified? and (ii)what values should be assigned to them?(iii)Based on the abnormal input, how can we diversify the exploit generation for a vulnerable program?

To answer these questions, PolyAEG traces program execution and performs dynamic taint analysis. During the taint analysis, PolyAEG detects all possible hijacking points, generalizes the constraints for the current execution path and identifies all user-controlled memory regions. When a hijacking point is detected, PolyAEG leverages trampoline instructions and one shellcode under the current runtime context, and accommodates them into the appropriate user-controlled memory regions to ensure that the hijacked execution flow reaches the shellcode. The data dependencies between the program input and the accommodated elements can be clearly identified by PolyAEG, so PolyAEG can find all relevant input bytes. They should be modified for exploit generation.(answer i)

In addition, as for an effective exploit, the values of the bytes to be modified should satisfy both data dependencies mentioned above and the path constraints. PolyAEG solves all the values for these bytes respectively, and use them to construct the new input, i.e., the exploit. When the program runs with this exploit, the control flow can be hijacked from the hijacking point and the trampoline instructions together with the shellcode can appear at expected memory locations.(answer ii)

PolyAEG can diversify combinations of different trampoline instructions and shellcode to generate polymorphic exploits. Moreover, PolyAEG is able to identify all possible hijacking points. For each hijacking point, PolyAEG performs the same exploit generation procedure as above, which contributes to more exploits.(answer iii). The generated exploits can be used to systematically evaluate the severity of the program vulnerability.

This paper makes the following contributions:

- We propose an PolyAEG architecture that can automatically generate polymorphic exploits by given program binaries and abnormal inputs. PolyAEG performs dynamic taint analysis to extract the execution information, analyzes the layout of the memory to accommodate the shellcode and trampoline instructions, and eventually constructs exploits by modifying relevant input bytes.
- We propose a novel approach to produce exploits by diversifying combinations of trampoline instructions and shellcode. It is not only increase the chance

for generating one effective exploit, but also contribute to polymorphic exploit generation, which is important for a systematic evaluation of a found vulnerability.
- We implement PolyAEG and verify it by generating exploits for several real-world program vulnerabilities. PolyAEG successfully generated control flow hijacking exploits for each program. Especially, it generates 4,724 exploits for IrfanView, a widely used picture viewer, using one abnormal input.

The remainder of paper is organized as follows: Section 2 introduces PolyAEG's overview. We present different phases for exploit generation in Section 3 to 5. PolyAEG was evaluated with different vulnerable programs in Section 6. We discuss limitations and future work in Section 7. Related work is presented in Section 8 and conclusions are in Section 9.

2 Overview of PolyAEG

PolyAEG takes in one vulnerable program and one abnormal input, and generates polymorphic exploits. Figure 1 shows the architecture of PolyAEG. Basically, PolyAEG is performed in the following three phases: *Dynamic Information Extraction*, *Constraint Generation*, and *Exploit Generation*.

- *Phase 1: Dynamic Information Extraction.* In this phase, we dynamically run the vulnerable program with the given abnormal input that can crash the program, trace each instruction and perform dynamic taint analysis to collect execution information. We analyze the taint propagation procedure to detect hijacking points of the control flow and extract tainted memory regions for storing utilized trampoline instructions and shellcode.
- *Phase 2: Constraint Generation.* The goal of this phase is to generate the path constraints which ensure that the hijacking point is reachable when the program runs with the exploit as input. The path constraints are generated based on the tainted execution information from Phase 1. They are represented by a set of constraint formulas with the input data as variables to check.
- *Phase 3: Exploit Generation.* In Phase 3, we leverage trampoline instructions to construct a trampoline instruction chain which redirects the program's execution to the shellcode. We accommodate the chain and the shellcode into tainted memory regions. We eventually generate one exploit by modifying the relevant input bytes according to specified data dependencies and path constraints identified in previous phases. Diverse patterns of trampoline instruction chains and multiple alternatives for shellcode location contribute to polymorphic exploit generation.

We will further discuss the details in the following sections.

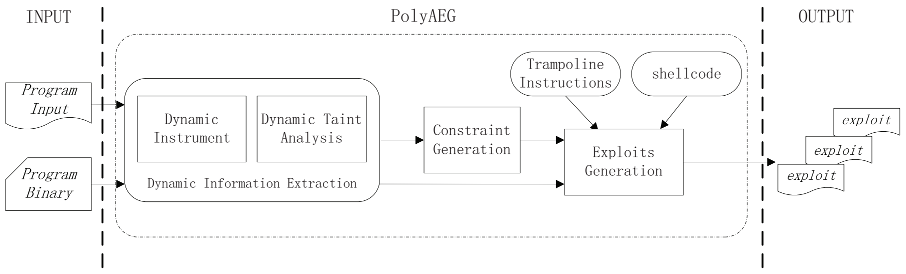

Fig. 1. The overview of PolyAEG.

3 Dynamic Information Extraction

An effective exploit must ensure the program's execution could be hijacked and the trampoline instructions and shellcode should be located at the appropriate places in the memory when the program runs with it as input. Therefore, we need to detect hijacking points, identify the path constraints restricting the execution to the hijacking point, and extract the layout of user-controlled memory regions that could be applied to accommodate trampoline instructions and the shellcode.

To achieve this, we trace the vulnerable program running with the abnormal input and perform fine-grained dynamic taint analysis at byte level. We enhance existing taint analysis approaches [11, 16] especially by constructing *iTPG*(*instruction-level Taint Propagation Graph*) and *GTSR*(*Global Taint State Record*), which not only records taint propagation but supports backtracking analysis.

iTPG records the taint propagation information during the vulnerable program running at instruction level. As is shown in Figure 2, a grey node represents a memory taint source corresponding to the data coming from the program input(e.g., files or network); a white node represents a tainted instruction. The edges linking nodes represents the data flow dependency among tainted data.

GTSR records taint states of memory bytes, general registers and bit flags in EFLAGS. Each item is represented by a 3-tuple <*TaintId, TaintStat, iTNode*>, where *TaintId* denotes the identifier of each byte or bit, *TaintStat* denotes whether it is tainted or not, and *iTNode* denotes a pointer which points to the tainted instruction last modifying the byte indicated by *TaintId*. Note that, a 32 bit register is specified by four bytes in *GTSR*.

iTPG and *GTSR* reflect the runtime context about taint propagation. The relationship between them is illustrated by Figure 2. When one tainted instruction *ti* modifies tainted bytes recorded in *GTSR*, a new *iTPG* node representing *ti* will be added into *iTPG*. We find the last tainted instruction *ti'* that modified those tainted bytes, and then link *ti* to *ti'*. Meanwhile, we update the corresponding *iTNode* pointing *ti'* before to point to *ti*. If executing *ti* also influences some bit flags in EFLAGS, we handle it similarly.

From *iTPG* and *GTSR*, we can idenfity the relevant input bytes of a tainted byte and the data dependencies between them. As is shown in Figure 2, a tainted

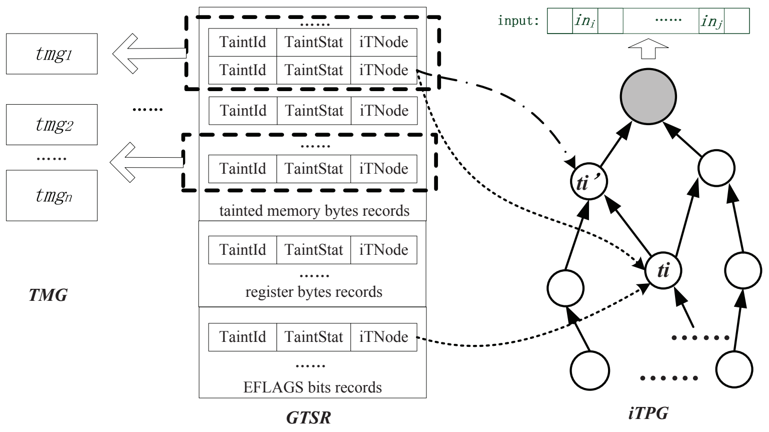

Fig. 2. *iTPG*, *GTSR* and *TMG*.

byte *tb* corresponds to one item in *GTSR*. The *iTNode* points to a node in *iTPG* which represents the tainted instruction last modifying *tb*. Backtracking along *iTPG* from this node to taint source nodes, we obtain a trace consisting of recorded tainted instructions. From that, we can identify the *tb*'s relevant input bytes $in_i, ..., in_j$ and their data dependencies $value(tb) = f(value(in_i), ..., value(in_j))$, where f can be educed by the semantics of the tainted instructions within the trace.

During dynamic taint analysis of the vulnerable program, we detect hijacking points by checking if tainted data are used in indirect control transfer(i.e., loaded on EIP) with *ret*, *jmp* and *call* instructions. When a hijacking point is detected, we identify the layout of tainted memory areas and path constraints. A tainted memory area consists of successive tainted bytes. We denote it as *t-mg*(*tainted memory garget*). It can be expressed as <*start*, *end*>, where *start* indicates the starting address of this area and *end* indicates the ending address. We extract all the *tmg*, denoted as *TMG*, from *GTSR*, and will utilize them to accommodate shellcode and trampoline instructions. Path constraints are identified by analyzing the executed path indicated by *iTPG*. We discuss it in the next section.

4 Constraint Generation

The exploits are generated by modifying relevant input bytes. The modifications should satisfy specified predicates that guarantee the program can execute to the hijacking point, especially when the input contains checksum fields.To ensure the hijacking point is reachable, we identify all "input-derived" branches within the path to the hijacking point, and generalize the constraints which reflect all the corresponding branch-taken results.

We denote "input-derived" branches as *tainted branches*. At each *tainted branch*, we identify the relevant input bytes that influence its branch-taken result, and generalize the corresponding constraints. A *tainted branch* instruction corresponds to an *iTPG* node in *iTPG*. Backtracking along *iTPG* from this node to taint source nodes, we can obtain the relevant input bytes and an *iTPG* nodes sequence which represents a trace of recorded tainted instructions. The trace can be used to symbolically generalize constraints for this *tainted branch*. In this paper, we utilize Z3 [12] which is a high-performance SMT solver to achieve this. First, we assign the relevant input bytes to different symbolic variables, and then perform concolic symbolic execution for each tainted instruction within the trace. At the branch instruction, since the corresponding bits in EFLAGS indicate the branch-taken result, we generate constraint formulas according to their values in SMT format [4].

Path constraints generated at *tainted branches* definitely guarantee that the hijacking point can be reached. However, it may have side effects, such as the following example.

```
    if(strcmp(taintstr,"http"))
        goto loc_1;
    else
        goto loc_2;
loc_1:
    //do sth causing the return address overwritten.
    // ...
    return; // hijacking point!
    // ...
loc_2:
    return;
```

If taintstr is "xttp", the hijacking point can be reached and the corresponding constraint will be generated as "$taintstr[0]! = h$" at one branch instruction in *strcmp*. However, according to the constraint, when taintstr is "hxtp", the hijacking point cannot be reached which is obviously incorrect. If we generalize constraints when *strcmp* returns instead of generalizing constraints at *tainted branches* within *strcmp*, we can obtain "$taintstr[0]! = h||taintstr[1]! = t||taintstr[2]! = t||taintstr[3]! = p$" which makes more sense.

To solve this problem, we perform constraint generation primarily at *tainted branches*, and secondarily at *tainted library calls*. When a tainted library function is called, we pause *tainted branches* constraint generation procedure, and identify the return address and the arguments of the function. When the function returns, we generate constraint formulas with the tainted arguments as symbolic variables according to the function's semantics and return result. After that, we resume *tainted branches* constraint generation. In our implementation, we handle comparison library functions for strings or memory such as *strcmp*, *strncmp*, *memcpy* etc. They are commonly used in vulnerable programs and influence whether hijacking points can be reached.

5 Exploit Generation

The exploit generation procedure is conducted when one hijacking point is detected. According to current execution context, we construct a trampoline instruction chain consisting of different trampoline instructions to redirect the control flow to the shellcode. Diverse patterns of trampoline instruction chains enable the variety of generated exploits. Together with the shellcode, the trampoline instructions within an adopted trampoline instruction chain should be accommodated into the tainted memory regions, since they must be contained within the exploit.

To construct the exploit, we find all relevant input bytes for the shellcode and the trampoline instructions within the chain. Then, we modify them to the appropriate values to ensure the expected exploiting procedure, i.e., when the vulnerable program runs with the generated exploit, 1)the utilized trampoline instructions and the shellcode can appear at the expected locations, 2)the control flow can be taken over at the hijacking point, and 3)the trampoline instructions are executed one by one until the execution of the shellcode eventually.

5.1 Trampoline Instruction Chain Construction

We mainly leverage three types of trampoline instructions to construct a trampoline instruction chain.

- {call/jmp register} For this type of trampoline instructions, the only operand is register. Normally, eight general registers can be used as the operands of the call/jmp instructions. Therefore, we can obtain 16 trampoline instructions.
- {call/jmp [register + offset]} The only indirect memory operand is decided by eight general registers and an offset. In this paper, the offset range is set between -256 and 256, and then we can construct 8192 trampoline instructions.
- {successive instructions sequence} Each trampoline of this type is a sequence of successive instructions in the code sections loaded into the process address space. They act as one instruction during executing, so we regard them as one trampoline instruction. We only consider one trampoline instruction of this type, i.e., *pop, pop, ret*, in this paper. It is commonly utilized for SEH exploits.

Given a trampoline instruction *I*, we can accurately compute its execution target address which is denoted as *I.target* in the current runtime context. Only when *I.target* is in one *tmg*, i.e., *tmg.start*≤*I.target*<*tmg.end*, it is considered as a candidate for constructing a trampoline instruction chain. We denote this *tmg* as *I.tmg*. Therefore, we can obtain a set of candidate trampoline instructions $Cand = \{I|tmg.start \leq I.target < tmg.end, tmg \in TMG\}$.

We denote a trampoline instruction chain as $TrampChain{=}I_0, I_1 \cdots, I_n$, where $1{\leq}\mathrm{n}{\leq}|Cand|$, $I_0, I_1 \cdots, I_n \in Cand$ and different from one another. A successful execution redirection by them is illustrated with the dotted line in Figure 3, where $I.addr$ represents the I's memory address in the process address space and $I.code$ represents the opcode of I. We can obtain the following characteristics:

(i)$I_0.addr$ replaced the tainted data used in indirect control transfer(i.e., the tainted return address or function pointer).

(ii)$I_{j+1}.code$ is accommodated at $I_j.target$, where 0≤j<n;

(iii)shellcode s is accommodated at $I_n.target$.

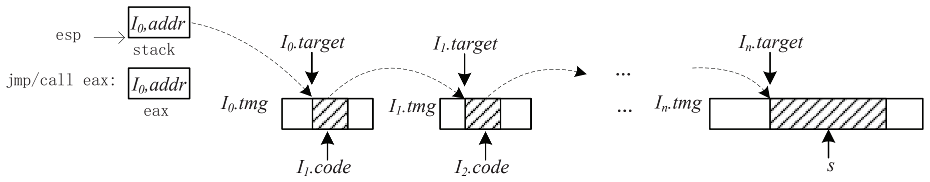

Fig. 3. The demonstration of a *TrampChain*.

Inspired by the characteristics above, we construct *TrampChain*s by analyzing all the possibilities of combining different candidate trampoline instructions in *Cand*. Then we select the *TrampChain*s that meet the following three criteria.

(i) $I_0.addr$ can be found in the unrandomized modules loaded into the process address space;

(ii) $len(I_{j+1}.code) \leq I_j.tmg.end - I_j.target$, for 1<j<n;

(iii) $I_n.tmg.end - I_n.tmg.start >= len(s)$;

There may be address conflicts in accommodating the trampoline instructions in a *TrampChain*. We mainly consider two cases of address conflicts:(i) I_j and I_k are accommodated in the same *tmg* and I_j is overlapped by I_k, where 0<j<k≤n, (ii)I_j is overlapped by the return address pushed when one *call* trampoline instruction I_m executes. We solve these conflicts as follows.

The address conflicts for case (i) contain two situations respectively shown in Figure 4(a),(b). For the former situation, i.e., $I_{k-1}.target < I_{j-1}.target < (I_{k-1}.target + len(I_k.code))$, we try to put $I_k.code$ into another memory area in this *tmg* which is long enough and not occupied by trampoline instructions, for instance, at $I_{k-1}.target'$. Meanwhile, we utilize an extra *jmp* instruction which is accommodated at $I_{k-1}.target$ to reach that position. For the latter situation, i.e., $I_{j-1}.target \leq I_{k-1}.target < (I_{j-1}.target+len(I_j.code))$, it cannot be solved and the corresponding trampoline instruction chain under construction cannot be used to generate a valid exploit.

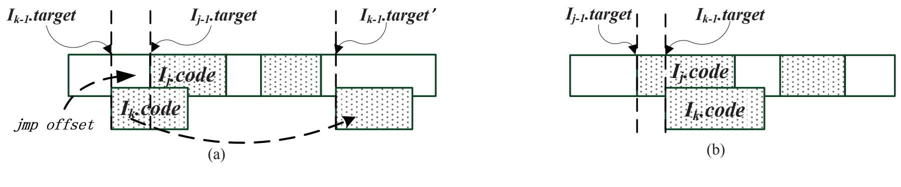

Fig. 4. Address conflicts for case(i) and the solutions.

For the case (ii), for any 0< j $\leq n$, if j>m≥0, the return address of I_m will corrupt I_j that has not executed, so the trampoline instruction chain will be destroyed. This address conflict cannot be solved and we discard the chain. However, if j<m≤n, since I_j finishes executing before I_m, the execution flow to the shellcode will not be influenced. In this situation, we enlarge the length of I_j from *len(I_j)* to *len(I_j)'*, as is shown in Figure 5. We then use *len(I_j)'* to solve the following address conflicts related to I_j.

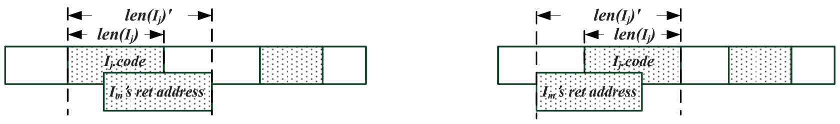

Fig. 5. Address conflicts for case(ii) and the solutions.

Diverse trampoline instruction chains can contribute to polymorphic exploit generation for the vulnerable program. Meanwhile, we know that the size of I_n*.tmg* may be larger than *len(s)*. Besides I_n*.target*, *s* has multiple alternatives to locate itself, i.e., the addresses before or after I_n*.target*, as is shown in Figure 6. We can enumerate all alternative positions for locating *s* in I_n*.tmg* to enable more exploits by leveraging an extra *jmp* instruction at I_n*.target* if needed. Address conflicts may happen in this situation as well. We use a similar approach to resolve them.

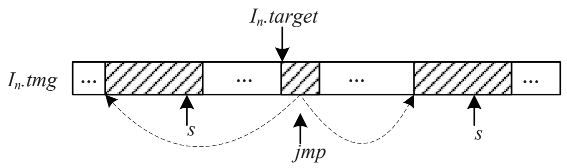

Fig. 6. Multiple alternatives for shellcode location in I_n*.tmg*.

Although *tmg*s may be loaded at different places because of stack or heap randomization when the program runs again, the offset between *I.target* and *I.tmg.start* will keep the same. After I_0 seizes the program's execution, the other trampolines within the *TrampChain* can be surely executed one after another as expected and the shellcode will get executed ultimately.

Our approach also works well when DEP is enabled. Since we are able to acquire the executable properties of tainted areas at runtime, we can construct a *TrampChain* on executable *tmg*s to bypass it.

5.2 Exploit Construction

If the hijacking point is detected and we finish accommodating the trampoline instruction chain and the shellcode without address conflicts in the tainted memory regions, we can finally construct the exploit by modifying all relevant input bytes.

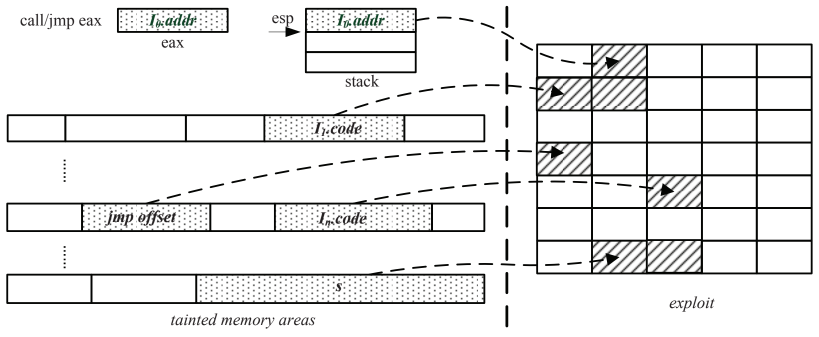

Fig. 7. The demonstration of exploit construction.

Figure 7 demonstrates the exploit construction.The hacked return address or function pointer is replaced by $I_0.addr$. The other instructions within the *TrampChain* and the shellcode are correspondingly accommodated at the previous instruction's target address. Any tainted byte *tb* is relevant to specific input bytes $in_i, ..., in_j$ and satisfy the formula $value(tb){=}f(value(in_i), ..., value(in_j))$, as is discussed in Section 3. We collect all such formulas for all accommodated bytes and submit them to a SMT solver [12] together with the path constraints that guarantee the hijacking point can be reached. If we successfully obtain the satisfying answers to all relevant input bytes, the exploit can be constructed by only modifying these input bytes to such new values.

Note our method can also be applied when only partial bytes(i.e., not 4-bytes) are controlled in the hacked return address or function pointer, which traditionally means insufficient control over the program's execution so that exploiting cannot be successful. We enumerate all possible values of the hacked return address or function pointer by altering the values of its controlled bytes. If there exists one alternative equivalent to one available trampoline's address, we are able to hijack the program's execution and conduct the following execution flow redirection. Our current approach for this case works well in the systems without module randomization. We leave that as our future work.

6 Evaluation

We developed PolyAEG based on QEMU [5]. We modified QEMU to support process identification, dynamic instrumentation, system call interception and dynamic taint analysis. We use Z3 [12] to generalize path constraints and query satisfying answers. We searched trampolines' addresses from the modules loaded into the vulnerable process. We generate trampolines' opcodes according to Intel assembly syntax. Shellcodes are selected from Metasploit [2]. We produce diverse exploit data containing trampolines and shellcode based on the program input, and eventually write them into files respectively as final multiple exploits. In all, PolyAEG consists of approximately 30,000 lines of C/C++ code. We evaluated PolyAEG on a Linux machine with a 3.2 GHz Intel(R) Core(TM) i5-3470 CPU, 500 GB hard disk and 4 GB RAM. We used 8 real world vulnerable programs. Their information are shown in Table 1. They all were performed in Windows XP SP2, QEMU's guest OS.

Table 1. List of programs that PolyAEG generated exploits for.

Program	Advisory ID.	Input Size	Type of hijacking points
IrfanView v3.99	CVE-2007-2363	2648	return address
Mp3 CD Ripper v2.6	CVE-2011-5165	4432	return address
WAV Converter v1.5	CVE-2010-2348	8208	function pointer
CoolPlayer v2.19.2	CVE-2009-1437	601	return address
Aviosoft DVD Player	CVE-2011-4496	1472	return address
FreefloatFtp v1.00	CNNVD-201302-349	981	return address
AutoPlay v1.33	CVE-2009-0243	701	function pointer & return address
Internet Download Manager v6.12	N/A	2340	return address

6.1 Method Validation

We use a test case to illustrate the process of exploit generation for vulnerable programs. For convenience, we present it using the statistics collected from the runtime context. They might be different as the program runs again. However, the offset between a trampoline's target and the corresponding *tmg*'s starting address would be unchanged, as mentioned in Section 5. It guarantees the correct execution flow redirection.

Freefloat Ftp will crash when it processes malformed remote user commands. In this experiment, we sent a user command(a string consisting of 1024 'A')that could crash it. PolyAEG performed dynamic taint analysis and showed that when "ret 0x8" at 0x00402ebb was executed, the taint data '0x41414141' would be loaded to *EIP* as a return address. Therefore, PolyAEG detected one hijacking point.

We used 338-bytes shellcode *BIND* shown in Table 5 to produce exploits. Table 2 shows candidate trampoline instructions for the vulnerable program. We denote $Cand_L$ as the set of trampoline instructions whose corresponding *tmg*'s size is larger than the shellcode, while $Cand_S$ is the ones whose corresponding *tmg*'s size is smaller than that. We know that *call esp* and *call* [*ebp*+0x14] could be employed as I_n within *TrampChain*, since their corresponding *tmg*s were large enough to accommodate the shellcode. In addition, only the addresses of *call edi* and *call esp* were available in the code sections of the process address space, they could be employed as I_0 within *TrampChain*. After analyzing different possibilities to construct *TrampChain*s with these trampolines, we obtained four effective *TrampChain*s shown in Table 3, with *#exploit* representing the corresponding count of generated exploits.

Table 2. The *Cand* of Freefloat Ftp when using shellcode *BIND*.

Cand	trampoline	*target*	*tmg*
$Cand_S$	*call edi*	0x911b24	<0x911b24,0x911c4d>
$Cand_L$	*call esp*	0xc0fc2c	<0xc0fb25,0xc0fef9>
	call [*ebp*+0x14]	0x911850	<0x911735,0x911b09>

We choose the fourth *TrampChain* shown in Table 3 to elaborate the execution flow redirection. The trampolines applied within the *TrampChain* were shown in Table 4. For *call esp*, as I_0, we used its address 0x7c934393 which was found in ntdll module and placed it at the stack space where stored the tainted return address, i.e., 0xc0fc20. As for *call edi* and *call* [*ebp*+0x14], they were put at the previous instruction's target respectively. We used their opcodes. The execution flow was redirected as follows: *call esp* initially hijacked the program's execution when "ret 0x8" was called. The execution flow reached 0xc0fc2c where *call edi* was placed. Then *call* [*ebp*+0x14] at 0x911b24 gained the execution flow after executing *call edi*. Finally, the shellcode located at 0x911850, the target address of *call* [*ebp*+0x14], got executed successfully.

Table 3. *TrampChain*s and the corresponding number of generated exploits.

TrampChain	*#exploit*
call esp	128
call esp − > *call* [*ebp*+0x14]	37
call edi − > *call* [*ebp*+0x14]	102
call esp − > *call edi* − > *call* [*ebp*+0x14]	100

Multiple alternatives in *tmg*<0x911735,0x911b09> were available for accommodating the shellcode besides 0x911850. We put the shellcode at other places in this *tmg* and leveraged an extra *jmp* at 0x911850 to reach the shellcode. Thus, we obtained multiple exploits (i.e., 100) under this *TrampChain*.

The exploits were eventually constructed by modifying all relevant input bytes according to the path constraints and the data dependencies between accommodated bytes and the program input. We validate them by running the vulnerable program. They turned out to be effective and could be applied to exploit the program.

Table 4. The statistics of trampolines in the 4th *TrampChain*.

I	trampoline	target	address	contents of I
I_0	*call esp*	0xc0fc2c	0xc0fc20	\x93\x43\x93\x7c
I_1	*call edi*	0x911b24	0xc0fc2c	\xff\xd7
I_2	*call* [*ebp*+0x14]	0x911850	0x911b24	\x33\xc0\xc6\xc0\x14\xff\x14\x28

Diverse patterns of *TrampChain*s and multiple choices for shellcode accommodations contribute to polymorphic exploit generation. Note that the length of the shellcode decides the last trampoline leveraged(i.e., I_n) to construct a *TrampChain*. Thus, if another shellcode with a different length is leveraged, we can obtain another set of different *TrampChain*s. Further, more exploits are available.

6.2 Polymorphic Exploit Generation

PolyAEG generated polymorphic exploits for 8 real world vulnerable programs. It identified 9 hijacking points. Of these, AutoPlay had two hijacking points. Both of them could be leveraged to generate exploits. In the experiments, we used AutoPlay$_1$ and AutoPlay$_2$ to denote the program with different hijacking points. Exploits were generated by hijacking function pointers in WAV Converter and AutoPlay$_2$. In other programs, PolyAEG generated exploits by hijacking return addresses.

PolyAEG could generate exploits containing various shellcode. We selected one shellcode for each vulnerable program randomly. Their properties are shown in Table 5. Table 6 illustrates the statistics about polymorphic exploit generation.$|Patrn|$ is the number of effective *TrampChain*s and *#exploit* is the number of generated exploits. Among these vulnerable programs, PolyAEG generated the maximum number of exploits for IrfanView. The main reason is that IrfanView had a broad memory area to accommodate the shellcode. CoolPlayer had the second quantity of exploits with the most patterns of trampoline instruction chains. The produced exploits with various attacking patterns are beneficial for systematically evaluating the vulnerability in CoolPlayer.

As GS security cookie protection [1] has been imported, the successful rate to exploit by hijacking an overflowed return address has reduced. Exploiting SEH(Structured Exception Handler) is a more effective and practical exploiting method. However, safeseh mechanism was introduced to prevent such exploits in Windows operating systems. Despite all this, SEH exploits can successfully be produced by our approach. Since not all modules loaded by a process arm with safeseh, we choose trampoline instructions in non-safeseh modules to bypass this protection mechanism. AutoPlay$_2$ and WAV Converter are such samples that we handled in this way.

Table 5. The properties of shellcodes.

ID	Functionality	Length	ID	Functionality	Length
CMD	spawn a shell	21	CALC	pop up a calculator	226
MSG	pop up a message box	45	RVSE	bind reverse tcp	366
ADD	add a new local account	233	NTPD	popup a notepad	86
DWN	download and execute	297	BIND	listen at port 4444	338

Table 6. The statistics of polymorphic exploit generation for all programs.

Program	shellcode	$\|Patrn\|$	$\#exploit$
IrfanView	DWN	3	4724
FreefloatFtp	BIND	4	367
CoolPlayer	CMD	29	3750
$AutoPlay_1$	MSG	3	282
$AutoPlay_2$	MSG	1	64

Program	shellcode	$\|Patrn\|$	$\#exploit$
Mp3CDRipper	ADD	1	3399
WAVConverter	CALC	4	180
Internet DMgr	NTPD	3	112
Aviosoft DTV Player	RVSE	1	126

Address conflicts could be well addressed. For instance, in an exploit of WAV Converter, *pop,pop,ret* was used as the trampoline instruction to hijack the program's execution and it would be overwritten if shellcode were located at its target address. To solve this problem, the shellcode was stored at a address before *pop,pop,ret* and *jmp* was used to redirect the execution flow to the shellcode.

In conclusion, PolyAEG is capable of generating exploits automatically and polymorphically for vulnerable programs. The polymorphic exploits generated with various attacking patterns will be conductive to systematically assess the severity of the vulnerabilities.

6.3 Performance Overhead

The overhead is dominated by the cost on dynamic taint analysis and exploit generation. The former is basically decided by recording taint propagation, and the latter is mainly decided by solving constraints. Figure 8 shows the quantities of tainted instructions(*#tainted inst*), and the average counts of constraint formulas(*#constraints*) and symbolic variables(*#variables*) for one exploit generation. *#tainted insts* could reflect the overhead of taint propagation for the program. Both *#constraints* and *#variables* indicated the overhead of solving constraints to construct exploits.

Figure 9 presents the time overhead of generating one exploit for each program. Mp3 CD Ripper and WAV Converter cost far more time than the others because they cost the most both on dynamic taint analysis and constraint solving. Freefloat Ftp ranked third since it had to solve the most constraints amongst all the programs except for Mp3 CD Ripper and WAV Converter.

We evaluated memory overhead of polymorphic exploit generation for each program. We used %MEM, i.e., the program's share of the physical memory, to present memory cost. It was mainly dominated by the quantity of all produced

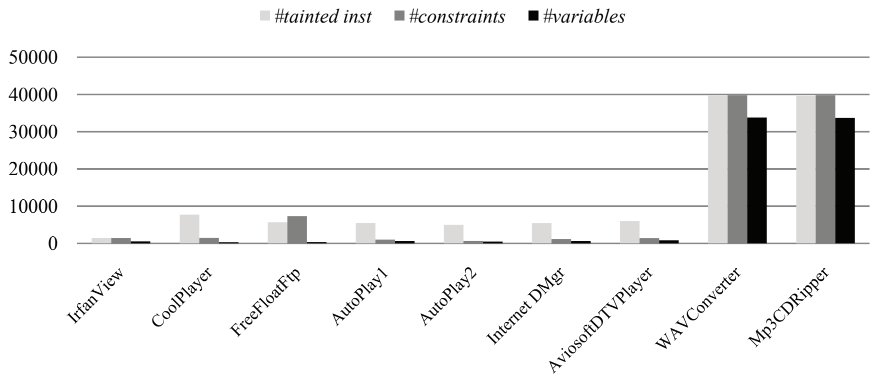

Fig. 8. The statistics about tainted instructions, constraint formulas and symbolic variables for all programs.

exploits and the expense for each exploit generation. From Figure 10, we know that both of these factors contributed to the highest cost of Mp3 CD Ripper amongst these programs. In addition, AutoPlay2 not only cost the least memory on generating one exploit, which was indicated as Figure 8, but also had the least exploits. It thus consumed the lowest memory resource.

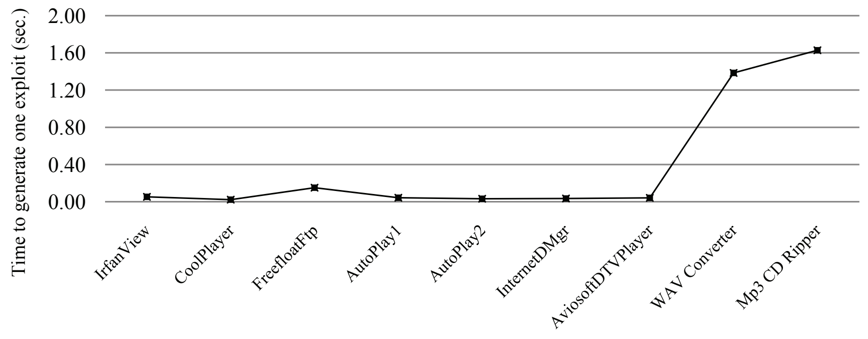

Fig. 9. The time overhead on one exploit generation.

In summary, as an off-line exploits generation system, PolyAEG has reasonable overhead on both time and memory consuming. We currently have little consideration on optimization of constraint solving. We will further handle it by eliminating duplicated constraints and simplifying the symbolic variables.

7 Limitations and Future Work

Our approach has some limitations. First, PolyAEG generates control flow hijacking exploits. It makes limited effort to bypass ASLR and DEP. Second, par-

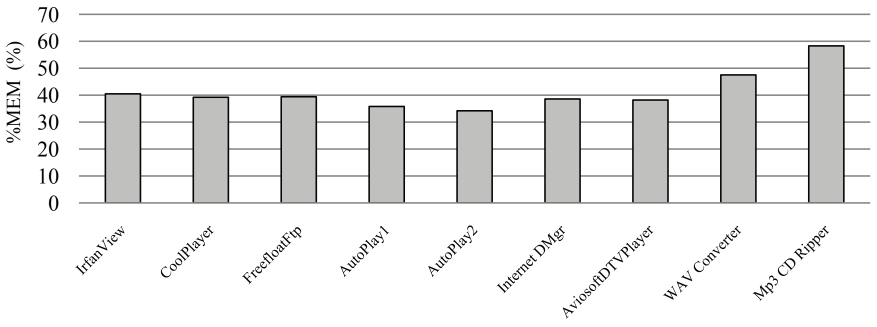

Fig. 10. The memory overhead on polymorphic exploit generation.

tially controlling to a hacked return address or function pointer is an obstacle to exploiting. PolyAEG can produce exploits only under specified conditions. Third, shellcode is wholly stored. We do not consider the situation when shellcode is split into several units and placed into different tainted areas. In addition, PolyAEG does not identify all possible paths to a hijacking point due to high expense on whole-system symbolic execution [3, 10]. But it generalizes path constraints that guarantee a reliable path to hijack program's execution.

We plan to extend PolyAEG to overcome those limitations in future work. We will also enhance PolyAEG to deal with more advanced exploitable situations about heap corruptions, use-after-free, and so on. Moveover, it is still an open problem to exploit non-control flow hijacked vulnerabilities, and we will do further research on it.

8 Related Work

Dynamic Taint Analysis. Dynamic taint analysis [17] can be used to tackle problems such as protocol reverse engineering, vulnerability detection, exploit generation, signature generation and so on. A few of general frameworks are available, such as [16, 11, 6, 18, 19]. TaintCheck [16] is one of the first dynamic taint analysis tools for protecting binary program from memory corruption attacks. TaintEraser [19] applies taint analysis for binaries to identify information leaks. Dytan [11] is flexible for taint analysis, allowing users to customize taint sources, sinks and propagation policy. Minemu [6] provides fastest taint analysis despite the limited functionally.

libdft [14] is a fast and reusable data flow tracking framework. It provides API for building dynamic taint analysis tools, e.g., libdft-DTA, and can be tailored to implement problem-specific instances.

Our enhanced taint analysis techniques with supporting backtracking analysis can be applied to support various analysis situations. Specifically for exploit generation, it enable us to accurately identify control flow hijacking, path constraints and data flow dependency.

Automatic Exploit Generation. Thanassis Avgerios *et al.* [3] proposed AEG for potential buggy programs. They used preconditioned symbolic execution to find exploitable paths and generated exploits by solving path predicate and exploit predicate. AEG worked solely on close source programs, and used hardened memory address of shellcode instead of trampolines, which would failed to exploit under address randomization. By contrast, PolyAEG aimed to generate exploits for vulnerable binary programs, and leveraged trampolines to redirect the execution flow to shellcode. Thus the generated exploits turned out to be more effective and practical to exploit a vulnerable program.

S. Heelan *et al.* [13] described a technique to automatically generate an exploit by given a crashing input for a vulnerable program by employing jump-to-register trampolines. However, few candidates for trampolines limited the ability of polymorphic exploit generation. PolyAEG provided multiple alternatives for trampolines, and leveraged them to construct diverse trampoline instruction chains. It not only increased the successful rate to generate one effective exploit, but enabled to generate multiple exploits which contributed to systematical evaluation of the severity of the vulnerability.

S.Cha *et al.* implemented MAYHEM [10] for finding exploitable bugs in binary programs and proving with working shell-spawning exploits. They proposed hybrid symbolic execution and index-based memory modeling that made exploitable bugs discovered efficiently. However, the exploit generation policy was similar to related works above. Thus, polymorphic exploit generation was not addressed in their system.

Brumley *et al.* [9, 8, 7]and Lin, Z.Q *et al.* [15] also gave solutions to automatic exploit generation problems. However, the exploits they generated in their works were not the same with ours. Their exploits were simply aimed to make the program run in an unsafe state, such as crashing or consuming 100% CPU, instead of executing a injecting shellcode.

9 Conclusions

We propose PolyAEG, a system that automatically generates multiple exploits for a vulnerable program using one corresponding abnormal input. To generate different polymorphic exploits, we fully leverage trampolines to construct diverse trampoline instruction chains in order to hijack execution flow and redirect it to shellcode within the runtime context. We used PolyAEG to successfully generate exploits for 8 vulnerable binary programs. In particular, we have generated 4,724 exploits using only one abnormal input for IrfanView, a widely used picture viewer.

References

1. GS. `http://msdn.microsoft.com/en-us/library/8dbf701c(v=vs.80).aspx`.
2. Metasploit. `http://www.metasploit.com/`.

3. T. Avgerinos, S. Cha, B. Hao, and D. Brumley. Aeg: Automatic exploit generation. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2011.
4. C. Barrett, A. Stump, and C. Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*, volume 13, 2010.
5. F. Bellard. Qemu, a fast and portable dynamic translator. USENIX, 2005.
6. E. Bosman, A. Slowinska, and H. Bos. Minemu: The world's fastest taint tracker. In *Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2011.
7. D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–16, may 2006.
8. D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Theory and techniques for automatic generation of vulnerability-based signatures. *Dependable and Secure Computing, IEEE Transactions on*, 5(4):224 –241, oct.-dec. 2008.
9. D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143 –157, may 2008.
10. S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.
11. J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007.
12. L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
13. S. Heelan. *Automatic generation of control flow hijacking exploits for software vulnerabilities*. PhD thesis, University of Oxford, 2009.
14. V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 121–132. ACM, 2012.
15. Z. Lin, X. Zhang, and D. Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 247 –256, June 2008.
16. J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS 2005)*. Internet Society, 2005.
17. E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317 –331, may 2010.
18. A. Zavou, G. Portokalidis, and A. D. Keromytis. Taint-exchange: a generic system for cross-process and cross-host taint tracking. In *Advances in Information and Computer Security*, pages 113–128. Springer, 2011.
19. D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Tainteraser: protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, 2011.