Article

AEMB: An Automated Exploit Mitigation Bypassing Solution

Ruipeng Wang, Zulie Pan, Fan Shi and Min Zhang *

College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China; wangruipeng@nudt.edu.cn (R.W.); panzulie17@nudt.edu.cn (Z.P.); shifan17@nudt.edu.cn (F.S.)
* Correspondence: zhangmindy@nudt.edu.cn

Abstract: Modern operating systems set exploit mitigations to thwart the exploit, which has also become a barrier to automated exploit generation (AEG). Many current AEG solutions do not fully account for exploit mitigations, and as a result, they are unable to accurately assess the exploitability of vulnerabilities in such settings. This paper proposes AEMB, an automated solution for bypassing exploit mitigations and generating useable exploits (EXPs). Initially, AEMB identifies exploit mitigations in the system based on characteristics of the program execution environment. Then, AEMB implements exploit mitigations bypassing the payload generation by modeling expert experience and constructs the corresponding constraints. Next, during the program's execution, AEMB uses symbol execution to collect symbol information and create exploit constraints. Finally, AEMB utilizes a solver to solve the constraints, including payload constraints and exploit constraints, to generate the EXP. In this paper, we evaluated a prototype of AEMB on six test programs and seven real-world applications. Furthermore, we conducted 54 sets of experiments on six different combinations of exploit mitigations. Experiment results indicate that AEMB can automatically overcome exploit mitigations and produce successful exploits for 11 out of 13 applications.

Keywords: software security; automated exploit generation; exploit mitigation; symbolic execution

check for updates

Citation: Wang, R.; Pan, Z.; Shi, F.; Zhang, M. AEMB: An Automated Exploit Mitigation Bypassing Solution. *Appl. Sci.* **2021**, *11*, 9727. https://doi.org/10.3390/app11209727

Academic Editor: Amerigo Capria

Received: 14 September 2021
Accepted: 13 October 2021
Published: 18 October 2021

1. Introduction

With the advancement of cyber security, there is an increase in the study of software security, particularly software vulnerabilities [1–3]. Automated exploit generation (AEG) is one of the best ways to assess the exploitability of vulnerabilities and is drawing more and more attention. Software developers can utilize it to evaluate vulnerabilities and prioritize high-risk vulnerabilities to patch first, while defenders can generate security rules to block zero-day attacks through analyzing the yielded exploits.

Given a proof-of-concept (PoC) sample that triggers a vulnerability, existing AEG solutions [4–9] in general start exploring from the program state triggered by the PoC to generate exploits (EXPs). However, the above solutions have not separately discussed the exploit mitigations, which have become a standard security feature in modern systems due to the development of safe compilers and software security. Thus, it is demanded for AEG solutions to support automated bypassing of exploit mitigations. In general, there are several challenges in generating available EXPs that can bypass the exploit mitigatons.

Challenge 1: Modeling and automation of techniques for bypassing of exploit mitigation.In order to overcome the exploit mitigations, the EXP must deliver an attack payload in the program's operating memory. Generating a compliant payload necessitates modeling and automating the technique for bypassing the exploit mitigation. Therefore, a solution that can automate the implementation of techniques for bypassing existing exploit mitigation is required.

Challenge 2: Modeling and automation of exploit techniques. The goal of AEG is to generate working EXPs, which should be able to satisfy the exploit constraints and payload constraints at the same time. The construction of exploit constraints need to refine and model the expert experience. Therefore, the automated exploit-mitigation bypassing solution needs to address the issue of modeling and automation of exploit techniques.

Our solution: In this paper, we present an automated solution *AEMB* to address the aforementioned challenges. It first generates payload constraints through exploit-mitigation bypassing model. Then, it constructs the exploit constraint based on the exploit experience and utilizes the solver to generate EXPs.

The bypass of the exploit mitigation mechanism is divided into three stages: exploit mitigation identification, bypassing payload generation, and payload constraint construction. We analyze and research the features of the exploit mitigations and the bypassing method. Based on the above, *AEMB* first automatically identifies the exploit mitigations. Then, the *AEMB* conducts unified scheduling and generates the essential parameters of bypassing of exploit mitigations since there may be a combination of several exploit mitigations in the environment. Finally, *AEMB* constructs exploit-mitigation-bypassing payload constraints and implements payload generation according to the key parameters ofbypassing exploit mitigations.

After the payload constraint is constructed, generating proper EXPs is still difficult. It requires patience, sophisticated experience, and high precision. Even for an experienced human expert, it takes a long time to craft a working EXP sample. Regarding automated EXP generation, researchers manage to generate EXP samples that require high precision by sketching complex exploitation-related constraints via symbolic expressions and solving constraints to yield working EXPs. Previous work shows symbolic execution is one of the most effective approaches to do this work [4–7,9–11]. *AEMB* also employs symbolic execution to generate EXPs. Through the analysis of existing exploit methods, *AEMB* can automatically complete the generation of exploit constraints. Finally, through the solver, *AEMB* can solve the exploit constraints and payload constraints to get the working EXP.

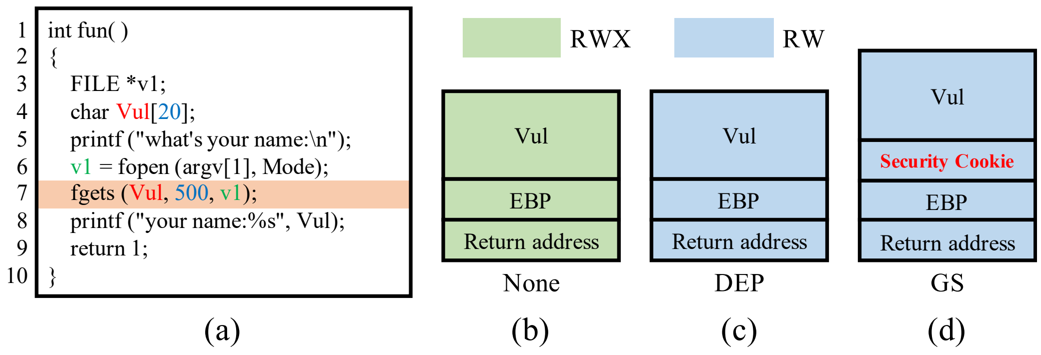
Evaluation result: We have built a prototype of *AEMB* based on the binary analysis engine S2E [12] and evaluated it on six test programs and seven real-world applications that run in the operating systems Windows and Linux. It could generate working EXPs for 11 out of 13 programs. More importantly, we evaluated six different combinations of exploit mitigations for each application (54 groups in total) and successfully generated working EXPs for 37 groups of them (as expected, the failure is all in the acceptable range), while existing open-source AEG solutions could not solve all of them. This demonstrated that *AEMB* is effective on automated exploit-mitigation bypassing and could generate working EXPs for most of them. Last but not least, we have evaluated the efficiency of *AEMB*, and the results show that *AEMB* can automatically bypass the exploit mitigations and generate EXPs in an average of 7.1 s.

In summary, we make the following contributions in this paper.

- We propose an automated solution *AEMB*, which can transform PoCs into EXPs that can bypass the exploit mitigations.
- We propose a new automated *exploit mitigations bypassing* solution to modeling and automating the method of the exploit mitigation bypassing.
- We have implemented a prototype of *AEMB* and demonstrated its effectiveness in test and real-world programs.

2. Motivation

In this section, we present the overview of our solution *AEMB* with a running example, as shown in Figure 1a.



```
 1  int fun( )
 2  {
 3      FILE *v1;
 4      char Vul[20];
 5      printf ("what's your name:\n");
 6      v1 = fopen (argv[1], Mode);
 7      fgets (Vul, 500, v1);
 8      printf ("your name:%s", Vul);
 9      return 1;
10  }
```

Figure 1. An example of stack overflow. (**a**) shows the source code of example. The vulnerability at line 7 could overwrite the return pointer of the *fun* function. (**b**) shows the stack memory state without exploit mitigation. (**c**) shows the stack memory state with DEP exploit mitigation. (**d**) shows the stack memory state with GS exploit mitigation. "RWX" means the memory has the permission of reading, writing, and execution. "RW" means the memory only has the permission of reading and writing.

2.1. Vulnerability and Exploit

There is an overflow vulnerability at line 7, as shown in Figure 1a. The data from file pointer `v1` at line 7 will be received by buffer `Vul`, which is placed on stack memory. The maximum size of the received data is 500 bytes, whereas the buffer `Vul` has a size of 20 bytes.

To exploit the above-mentioned vulnerabilities, we must first comprehend the stack's structure. As shown in Figure 1b, the function's return address is kept on the stack memory and is located beneath the buffer `Vul`. The function's return address is kept on the stack memory and is located beneath the buffer `Vul`, as shown in Figure 1b. If the overflow data corrupt the return address, the program's control flow will be hijacked, and the function will return to an unexpected location. The vulnerability can be exploited by crafting the overflow data and hijacking control flow to the shellcode in the stack.

2.2. Impediment

Even though the aforementioned vulnerability exploit approach might lead to getshell, it may fail, owing to a variety of impediments, particularly exploit mitigations. Some of them are listed below.

2.2.1. DEP

Data Execution Prevention (DEP) [13] is a security mechanism that monitors and protects certain memory pages or regions, preventing them from executing code. In Windows, it's known as DEP, while in Linux, it is known as NX [14]. We can see that DEP designates the stack memory as non-executable by comparing Figure 1b and Figure 1c.

The code in the data segment cannot be executed due to this mechanism's protection. A crash will occur after the program's control flow is hijacked to the shellcode, which is built from overflow data. As a result, the approaches for exploit listed above will be invalid.

2.2.2. GS

GS [15], a compilation option, is a security mechanism to protect the stack frame. In Windows, it is known as GS, while in Linux, it is known as Canary. The allotted space is loaded using a security cookie that is calculated once at module load on the function entrance. A helper function is run when the function returns to ensure that the cookie value remains the same. By comparing Figure 1b and Figure 1d, we can find that the security cookie protects the EBP pointer and return address once its value tampers with indicators that an overwrite of the stack may have occurred.

The overflow data must be restored to the value of the security cookie, which cannot be obtained directly, under the protection of this technique. Once the overflow data are overwritten to the security cookie and modified to different data from the previous, a crash will occur when the function returns. As a result, the approaches for the exploit listed above will be invalid.

2.3. Concept

For ease of understanding, we introduce several concepts:

- **Bypassing Payload**: The bypassing payload [16] is a custom code that the attacker wants the system to execute and bypass the exploit mitigations.
- **Symbolic constraint**: During the symbolic execution, the information related to the external input of the program in the system is represented by symbols. These symbols can represent the constraint relationship between external input and program operation, and they are called symbol constraints.
- **Payload constraint**: The payload constraint is a symbolic constraint for arranging the payload in the memory, and the arrangement of the payload can be realized by solving it.
- **Exploit constraint**: The exploit constraint is a symbolic constraint for vulnerability exploit, and the PoC can be converted to EXP by solving it.

2.4. Our Solution: AEMB

We proposed a novel solution *AEMB*, to break the above impediments. At the high level, *AEMB* bypasses the exploit mitigations, *AEMB* automatically builds exploit mitigations bypassing payload constraint based on expert experience, and then it utilizes symbolic execution to generate EXPs. As shown in Figure 2, it contains two major components.

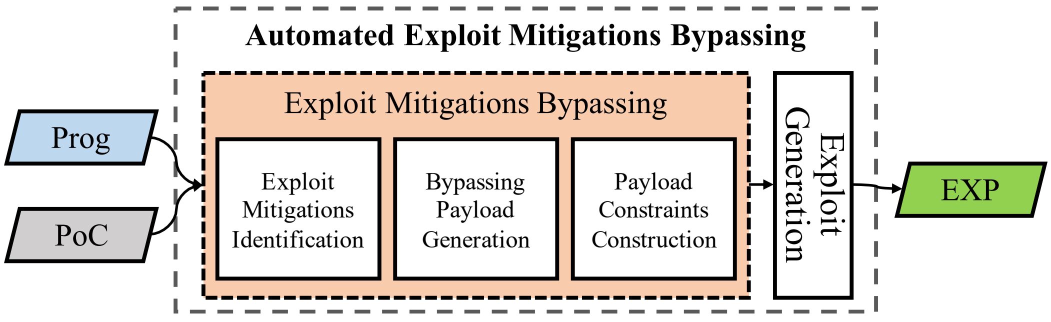

Automated Exploit Mitigations Bypassing
Exploit Mitigations Bypassing
Prog
PoC
Exploit Mitigations Identification
Bypassing Payload Generation
Payload Constraints Construction
Exploit Generation
EXP

Figure 2. Overview of *AEMB*. The input is the vulnerable program and the PoC that triggered the vulnerability, and the output is the EXP.

2.4.1. Exploit Mitigations Bypassing

Exploit mitigations bypassing is the step to build the payload constraints that can bypass the exploit mitigations.

This stage mainly includes three steps: exploit mitigations identification, bypassing payload generation, and payload constraint construction. The descriptions of them are provided below.

- **Exploit Mitigations Identification.** The *Exploit Mitigations Identification* judges which exploit mitigations exist in the current running environment of the program. This part mainly analyzes the characteristics of exploit mitigations and constructs the corresponding identification model.
- **Bypassing Payload Generation.** The *bypassing payload generation* is mainly to realize the generation of necessary parameters for the mitigations bypassing according to the expert experience. This part focuses on automating existing bypassing technology for exploit mitigations .
- **Payload Constraints Construction.** The *payload constraints construction* mainly builds the relationship between memory and load through constraints. This part mainly

constructs the constraint between symbolic memory and bypassing payload through symbolic execution, thus providing the constraint for subsequent EXP generation.

2.4.2. Exploit Generation

Exploit generation is the step that can generate the exploits meeting the exploit constraints and payload constraints. As with most AEG solutions, *exploit generation* builds exploit constraints from expert experience and solves all constraints by SMT [17] to generate EXPs.

3. Exploit Mitigations Bypassing

Given a target application and a PoC, *AEMB* first identifies the exploit mitigations in the running environment, then generates the corresponding key bypassing parameters of the security-mechanism, and finally constructs the payload constraints for EXP generation.

3.1. Exploit-Mitigation Identification

The first stage of *bypassing exploit mitigations* is to identify exploit mitigations, which is mainly to determine which exploit mitigations exist in the current environment. The following takes some specific exploit mitigations, which are the most basic mechanisms DEP, GS, and SAFESEH, on Windows and Linux, as examples to illustrate. They are introduced as follows.

- **DEP.** The DEP exploit mitigation requires the support of the scompiler and operating system under Linux and Windows, and its essence is to disable the execution permissions of the data memory page. The *AEMB* dynamically executes the target application and then detects the permissions of memory pages that store the data, thereby judging the the DEP mechanism to be enabled, as shown in Figure 3a.
- **GS and SAFESEH**. The GS exploit mitigation is to include stack frame protection codes, which involve saving the security cookie at the calling of the function and verifying the cookie at the return of function at compile time. The *AEMB* statically analyzes the assembly code to judge the enablement of the GS mechanism through the existence of stack frame protection codes. As shown in Figure 3b.
  The Safe Exception Handlers (SAFESEH) [18] is a Windows mechanism that can prevent the bypassing of the GS mitigation. It can produce a table of the process's safe exception handlers. This table specifies for the operating system which exception handlers are valid for the process. The *AEMB* determines if the SAFESEH security mechanism is enabled by determining whether the current operating system supports the SAFESEH security mechanism and whether the SAFESEH flag bit is enabled in the binary's file format, which is the PE file.

3.2. Bypassing Payload Generation

The second stage of *exploit mitigations bypassing* is to overcome the exploit mitigation based on professional expertise and produce the essential parameters, which are the bypassing payloads, for the subsequent process.

The following section describes and explains the particular exploit mitigation in Section 3.1 as an example.

3.2.1. Program Analysis

To begin, the *AEMB* must examine the target application and its memory layout in the operating environment. It can produce some basic information that is critical for either an automated or manual exploit. The following information is specifically required for *bypassing payload generation*.

- **The memory with known address**. This part is mainly aimed at the Address Space Layout Randomization (ASLR) exploit mitigation, which is inextricably linked to the whole exploit process, and therefore the system sees it as a feature of the operating system. The *AEMB* will determine whether the address of each memory page is known. The known address is obtained through two methods: 1. The address of the target memory page is fixed. 2. The address of target memory pages was leaked via the PoC provided.

- **The memory affected by the PoC**. This part mainly judges which memories may be affected by the PoC. It is required for the exploit since the PoC is converted into the EXP by creating the memory affected by the PoC.
  Taint analysis is used to acquire this kind of memory. The PoC is marked as the source of taint, and the location where the PoC triggers the vulnerability is marked as the sink of taint, as shown in Figure 4.

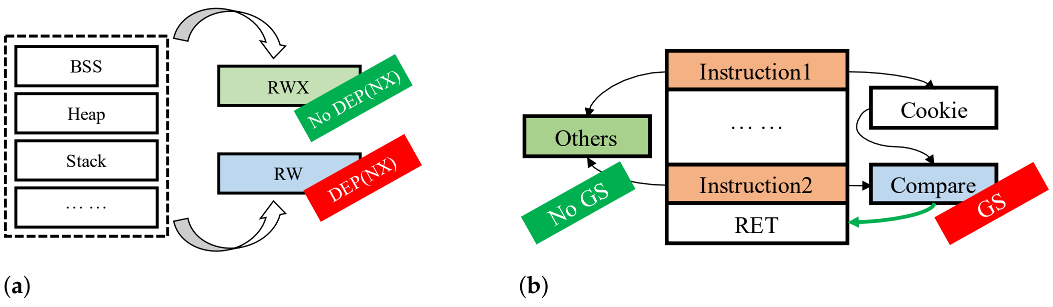


Figure 3. Identification of exploit mitigations. Mainly includes the identification of DEP and GS mitigations. (**a**) Identification of DEP exploit mitigation, mainly based on the permissions of the data segment. (**b**) Identification of GS exploit mitigation, mainly based on the storing and checking instructions of the security cookie.

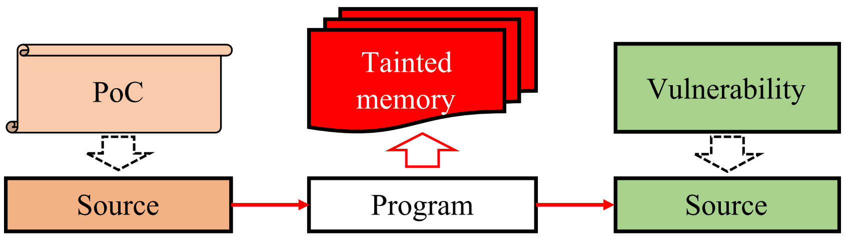


Figure 4. The process of obtaining memory affected by external input.

3.2.2. DEP Bypassing Payload Generation

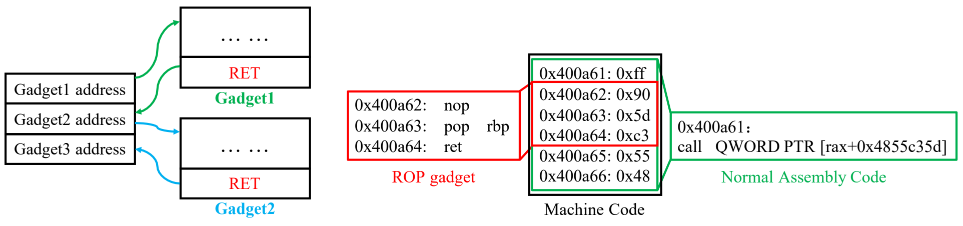
The DEP security mechanism primarily limits execution permission in the data segment, preventing the program's control flow from being hijacked to the data segment.

As a result, for exploit, Return-oriented Programming (ROP) technology is required, which does not directly execute code on the data segment but instead deploys function pointers on the data region and calls the lawful code in the code segment through the function call and return mechanism.

The DEP bypassing payload generation aims to create a sequence of ROP gadgets that can be combined to fulfill the exploit's goal.

The feature of ROP gadgets is a series of register operations, system calls, etc. with an end of `ret` instructions. The `ret` instructions allow gadgets to be linked and then combined to perform the required function, as shown in Figure 5a. These gadgets are not limited to the assembly code that exists in the program, but the machine code that exists in the process memory. In the binary's assembly code, for example, there are `call QWORD PTR [rax+0x4855c35d]` instructions that cannot be utilized as ROP gadgets. The intercept portion of the machine code, on the other hand, may turn it into a gadget, as shown in Figure 5b.

According to the aforementioned ROP gadget features, *AEMB* can rapidly scan the effective memory space of the application and get the anticipated ROP gadget address. Following that, *AEMB* decides which ROP must be built based on the target program running environment.



(**a**) (**b**)

Figure 5. The concept of ROP gadget. (**a**) The process of ROP gadget linking ROP. (**b**) Demonstration of machine code as a ROP gadget. The ROP gadget is not necessarily from assembly code, but it can be constructed by intercepting assembly machine code.

The main factors affecting the ROP structure are as follows:

- **Expected functionality**. This mainly refers to the expected function, which mainly refers to getshell, to be completed via ROP. Under Linux, the most straightforward way is to call the `system` function and pass in the `/bin/sh` parameter. Some functions and their corresponding ROPs are present in the *AEMB*. However, often due to other exploit mitigations, especially ASLR, the construction of ROP will not be as ssmooth.
- **Available ROP gadgets**. This mainly refers to gadgets, which are stored in the memory with a known address and can be used in ROP. If it does not exist, its needs to be replaced with one or some similar gadgets. The gadget with an unknown address should be replaced by one or more alternative gadgets that perform the same purpose. For example, `add rsp,8;ret` can be replaced by `push reg;ret`, and `leave;ret` can be replaced by `mov rsp,rbp;ret` and texttpop rsp;ret. *AEMB* sets many basic replacement rules such as as the above-mentioned rules.
  Furthermore, the *AEMB* employs the "shadow process", which is a simulated process that focuses on the memory of the source application in order to determine if the alternative gadget can replace the source gadget. The *AEMB* will simulate the execution of the source gadget and alternate gadget via the "shadow process", and record the content of the registers and stack memory. These data are used to ensure the immutability of the normal function of the gadgets (for example, if the RAX register is expected to be executed minus one by source instruction, the alternate instruction should also complete this operation), as well as the immutability of the memory layout, particularly the stack memory layout, as shown in Figure 6.
- **Size of the buffer**. Its primary purpose is to define the maximum length of the built ROP, which cannot be longer than the data buffer.

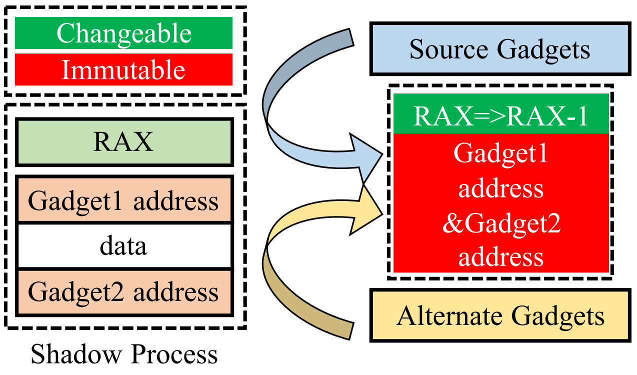


Figure 6. Demonstration of ROP replacement. The source gadgets decrement the RAX register by one, and alternate gadgets need to be consistent with the source gadgets, on the shadow process.

Taking into account the aforementioned contributing variables, the *AEMB* will first utilize the predefined gadgets to build the ROP in accordance with the anticipated purpose.

If the anticipated ROP gadget does not exist in the memory, it will be replaced with an alternate gadget.

In the process of ROP construction, it is necessary to determine the length of the ROP in real time. If the length of the ROP is larger than the size of the current buffer, it indicates that the preset ROP's construction failed and that other preset ROPs are utilized for construction. The process is shown in the Algorithm 1. After the ROP is constructed, the *AEMB* outputs the constructed ROP as the key parameters.

Algorithm 1: ROP construction algorithm

```
Data: ROP gadget $Rg_1 \dots Rg_n$, Buffer Size $S$, Default ROP $\mathbf{R_e}$
Result: Final ROP $\mathbf{R_f}$
1  $SizeR_f \leftarrow 0$;
2  for $i = 1$ to $LengthOf(\mathbf{R_e})$ do
3      if $SizeR_f > S$ then
4          Failed;
5      end
6      if $Exist(\mathbf{R_e}_i) == 0$ then
7          $< Rg_j, \dots, Rg_k > \leftarrow Replace(\mathbf{R_e}_i)$;
8          $< \mathbf{R_f}_i, \dots, \mathbf{R_f}_{i+k-j} > \leftarrow < Rg_j, \dots, Rg_k >$;
9          $SizeR_f \leftarrow SizeR_f + LengthOf(< Rg_i, \dots, Rg_j >)$;
10     else
11         $\mathbf{R_f}_i \leftarrow \mathbf{R_e}_i$;
12         $SizeR_f \leftarrow SizeR_f + 1$;
13     end
14 end
```

3.2.3. GS and SAFESEH Bypassing Key Parameter Generation

The GS safety mechanism mainly protects the stack frame data during the function call, and the bypassing for GS is mainly based on two types of techniques. One is to leak the Security Cookie so that the process can pass the GS, and it applies to any operating system. The other is hijacking the safe exception handler, which takes over the control flow of the program after the GS check fails, and it is mainly suitable for Windows hosts with a Structured Exception Handler (SEH) mechanism.

Leaking the Security Cookie requires stringent requirements to environmental and vulnerability, and it is hard to realize; therefore, we focus on the technique of hijacking SEH.

AEMB firstly checks whether the operating system supports the SAFESEH mechanism, which may cause the failure of bypassing due to protection for SEH. Then it examines the SAFESEH flags in the PE files, which contain all applications and dynamic link libraries loaded during runtime. Some programs and dynamic link libraries choose to compile in a low-version environment for program compatibility, thereby making them disable SAFESEH. Finally, it looks for appropriate instructions in the program or dynamic link libraries, which have not opened SAFESEH, to implement the GS security mechanism bypass.

AEMB mainly finds hijacked SEH instructions and corresponding memory addresses as key data for output. At first, *AEMB* searches in the target dynamic link library for instructions, which are `pop; pop; ret` style, since `sp +8` frequently points to the next SEH, which generally can be written in by external inputs. Therefore, the `pop; pop; ret` style instruction and the memory address of the next SEH will be output as key data first.

Furthermore, the aforementioned technique may not work in all circumstances; therefore, the operation of the SP register is also needed to complete the exploit. *AEMB* uses the above-mentioned taint analysis result to obtain the target memory, which can be written by external input. It also needs to find the corresponding instruction to adjust the SP register, such as `add esp, offset`, and adjust the offset size of ESP upwards to the target memory, as shown in Figure 7.

Finally, the instruction and the corresponding memory are output as key parameters when the next SEH and `pop; pop; ret` style instructions cannot be used as parameters.

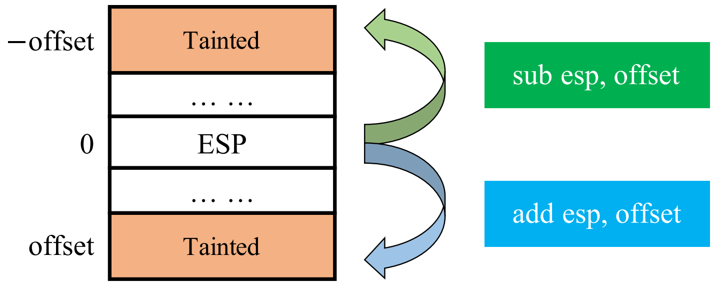


Figure 7. Demonstration of the adjustment of the ESP pointer by `add esp,offset` and `sub esp,offset`. The adjusted pointer needs to point to the tainted memory.

3.3. Payload Constraints Construction

The *payload constraints construction* mainly uses the bypassing payload parameters to construct symbolic payload constraints.

The *payload constraints construction* completes the above-mentioned tasks through the following steps: 1. Symbolize the program input—that is, symbolize the PoC—and execute the program through symbolic execution; 2. Construct the exploit-mitigation bypassing constraints according to the key parameters generated by the *AEMB*. Through symbolic execution technology and related memory constraints, the corresponding system safety-mechanism-bypassing payload is deployed in the memory through constraint solving.

The PoC is symbolized, and the symbolic execution status input by the user and the symbolic expression are collected during the symbolic execution of the program. The construction of the exploit-mitigation bypassing constraints is mainly for the expressions between symbolic memory and concrete values. For example, the one-byte memory symbol expression at *Targetmemory* is shown in Equation (1).

$$(Read\ w8\ 0\ Targetmemory) \tag{1}$$

The system expects that the expression for writing *Targetvalue* to the *Targetmemory* memory can be constructed as shown in Equation (2).

$$(Eq\ Targetvalue\ (Read\ w8\ 0\ Targetmemory)) \tag{2}$$

Through the above symbolic expressions, a one-to-one correspondence between symbolic memory and expected values can be constructed. In the bypassing of exploit mitigations, similar expressions often appear for example, in the DEP exploit-mitigation bypassing, to fill the specified memory with ROP, as shown in Equation (3).

$$\begin{gathered}(Eq\ ROP_0\ (Read\ w8\ 0\ Targetmemory))\\ \bigcap(Eq\ ROP_1\ (Read\ w8\ 1\ Targetmemory))\\ \bigcap(Eq\ ROP_2\ (Read\ w8\ 2\ Targetmemory))\\ \vdots\end{gathered} \tag{3}$$

Finally, the *payload constraints construction* outputs the constraints constructed by the above method to the EXP generation component.

4. Exploit Generation

The exploit generation technique is used to build a model of different exploit methods. Then, the model is used to construct the exploit constraints and payload constraints. Lastly, the solver is used to solve the constraints and produce input that can comply to the exploit conditions.

Specifically, *AEMB* first symbolizes the program's input. Then, using predefined vulnerability characteristics, it identifies the type of vulnerability based on the symbolic memory layout. Finally, it uses the preset exploit techniques to construct exploit constraints. The preset vulnerability features and exploit method are shown in Table 1.

Table 1. List of currently applicable vulnerability types, vulnerability features, and exploit methods.

Module	Vulnerability Features	Exploit Method
Control-flow Hijack Vulnerability	Symbolic PC	Hijack PC
Format String	Symbolic Arguments	Modify Return Pointer

4.1. Control-Flow Hijacking Vulnerability Exploit Constraint Construction

The characteristic of control-flow hijacking vulnerability is that data related to program control flow are controlled by user input, thereby changing the control flow of the program. After the program input is symbolized, due to control-flow hijacking vulnerability directly hijacks the program control flow, and the EIP register will be symbolic at a certain moment. Therefore, *AEMB* monitors the EIP register during program execution and judges whether the vulnerability is control-flow hijacking vulnerability by determining whether the EIP register is symbolized.

After detecting this type of vulnerability, *AEMB* will construct the exploit constraints. The main method of an exploit is to hijack the PC to the attack payload. If there are exploit mitigations, the attack payload will be generated by exploit mitigations bypassing (which is introduced in Section 3); otherwise, the shellcode, which can be constructed using Equation (4), will be used as the attack payload.

$$\begin{gathered}(Eq\ Shellcode_0\ (Read\ w8\ ShellcodeLocal\ TargetMemory))\\ \bigcap(Eq\ Shellcode_1\ (Read\ w8\ ShellcodeLocal+1\ TargetMemory))\\ \bigcap(Eq\ Shellcode_2\ (Read\ w8\ ShellcodeLocal+\ TargetMemory))\\ \vdots\end{gathered} \tag{4}$$

4.2. Format Vulnerability Exploit Constraint Construction

Format string vulnerability is caused by the tainted format string parameters by external input. After the program input is symbolized, due to format string vulnerability tainting the format string parameters, the format string parameters will be symbolic at a certain moment. Therefore, *AEMB* hooks the target format string function in the program and judges whether the vulnerability is the format string vulnerability by determining whether format string parameters are symbolized.

After detecting this type of vulnerability, *AEMB* will construct the exploit constraints. The main method of an exploit is to modify the return pointer to the attack payload. Similarly Section 4.1, it will indirectly hijack the PC to the attack payload (same as Section 4.1).

However, unlike the control-flow hijacking vulnerability, the format string vulnerability needs to use a format string payload, which is in the form of $address + padding + \%offset\$n$, to achieve arbitrary address writing, thereby modifying the return pointer. Among them, *address* represents the target write address, and *offset* represents the relative offset of *address* from the first parameter. The offset can be calculated using Equation (5) (where the symbol "*" represents the value in the memory with the operand as the address), and the overall format string payload can be constructed using Algorithm 2.

$$offset = (*(ESP+4) - (ESP-4))/4 \tag{5}$$

Algorithm 2: Format string payload construction algorithm

```
Data: Target address $T_a$, Target value $T_v$, relative offset of parameter Offset
Result: Target format string $\mathbf{F_s}$
1 $T_v\_high \leftarrow T_v \ll 16$;
2 $T_v\_low \leftarrow T_v\%65{,}536$;
3 if $T_v\_high < T_v\_low$ then
4     $F_s = (T_a + 2) + (T_a) + ''\%'' + (T_v\_high - 8) + ''s''$;
5     $F_s = F_s + ''\%'' + (Offset) + ''\$n\%'' + (T_v\_low - T_v\_high) + ''s\%'' + (Offset + 1) + ''\$n''$;
6 else
7     $F_s = (T_a) + (T_a + 2) + ''\%'' + (T_v\_low - 8)$;
8     $F_s = F_s + ''\%'' + (Offset) + ''\$n\%'' + (T_v\_high - T_v\_low) + ''s\%'' + (Offset + 1) + ''\$n''$;
9 end
```

4.3. Constraint Solving and Exploit Generation

As with most AEG solutions, *AEMB* combines the above-mentioned exploit constraints and payload constraints and solves all constraints by SMT [17] to generate exploits. (When there are exploit mitigations, it is generated by the *exploit-mitigation bypassing* component; otherwise, it is generated by *exploit generation* component.)

5. Evaluation

In order to evaluate the vulnerability automatic verification method in this paper, we implemented *AEMB* and designed an experiment.

5.1. Implementation

We implemented a prototype of *AEMB* based on S2E [12]. It consists of 612 lines of code to bypass exploit mitigations and 1022 lines of code to generate EXPs.

AEMB performs a full system simulation to monitor the running states of the target program by QEMU [19]. It implements the symbolic execution analysis of the system by KLEE [20] and realizes the constraint solution by Z3 [17]. The experiments are conducted in a 64-bit Ubuntu 18.04.4 LTS system on a server with 32 G RAM and Intel(R) i9-9880H CPU @ 2.30 GHz. In addition, the experiments use different versions of Windows operating system hosts and Linux hosts to test the effect of the EXP generated by the system.

5.2. Benchmarks

We applied *AEMB* to six test programs and seven real-world applications from CVE (Common Vulnerabilities and Exposures) [21] and CNVD (China National Vulnerability Database) [22]. The details of benchmarks are shown in Table 2.

The benchmark was constructed for the evaluation, mainly based on the following rules:

- The test programs should be able to cover different types of vulnerabilities, different operating systems, and different security mechanisms. Among them, the bof, Fmt-3, and BabyCookie vulnerabilities are mainly stack-related vulnerabilities, and Heap-func, Heap-uaf, and Heap-fmt are mainly heap-related vulnerabilities. Furthermore, they can cover Windows and Linux operating systems and different security mechanisms.
- The real-world applications are mainly from user-mode applications with public vulnerabilities and PoC. Current exploit mitigations automated bypassing method is for the user-mode, not the kernel-mode. The focus of this paper is the exploit mitigations automated bypassing, and vulnerabilities mining is not our main job. Therefore, this paper selects the software with open vulnerabilities.

Table 2. List of benchmarks evaluated with *AEMB*.

Type	Program	Num.	OS	Vuln.
Demo Program	bof	N/A	Windows/Linux	Stack-related Vulnerability
	Fmt-3		Linux	
	BabyCookie		Windows	
	Heap-func			Heap-related Vulnerability
	Heap-uaf		Linux	
	Heap-fmt			
Real-world Program	Free MP3 CD Ripper 2.6	CVE-2019-9766	Windows	Stack-related Vulnerability
	Free WMA MP3 Converter 1.8	CNVD-2014-077615		
	Adrenalin Player 2.2.5.3	CNVD-2014-01185		
	AudioCoder 0.8.18	CNVD-2017-26587		
	Nidesoft MP3 Converter 2.6.18	CNVD-2016-12839		
	Mini-stream Ripper 2.7.7.100	CNVD-2011-3422		
	DNSTracer 1.9	CVE-2017-9430	Linux	

We use both test programs and real-world applications in the evaluation. The real-world applications and the details of their vulnerabilities can be searched through the name of the application, corresponding version, and vulnerability number. The test programs are introduced below.

bof The *bof* is an ELF file, which is mainly run on a Linux operating system. The interaction of the program is through the command line, and it mainly receives data from STDIO. The program contains a stack overflow vulnerability. When the length of the string received is greater than a buffer length, the stack frame metadata will be overwritten, which can hijack the control flow of the program.

BabyCookie The *BabyCookie* is a PE file, which is mainly run on the Windows operating system. The interaction of the program is through the command line, and mainly receives data from external files. Unlike the bof program, the program uses a test dynamic link library, which has been compiled. To simulate the behavior of much existing software considering compatibility, the dynamic link library is compiled with a low-version compiler, without SAFESEH or other compiling-related system safety mechanisms.

Fmt-3 The *Fmt-3* is an ELF file, which is mainly run on the Linux operating system. The interaction of the program is through the command line, and it mainly receives data from STDIO. The program contains a format string vulnerability, and external input can directly affect the format string parameters on the stack memory, resulting in the occurrence of format string vulnerability. There is a loop structure, and the ending condition is inputting an "exit" string. In the loop, the program can trigger the format string vulnerability multiple times.

Heap-func The *Heap-func* is a PE file, which is mainly run on the Windows operating system. The interaction of the program is through the command line, and it mainly receives data from STDIO. The program contains a heap overflow vulnerability; the overflow data can overwrite the function pointer of the structure on the heap space. The program will call this function pointer during subsequent executions.

Heap-uaf The *Heap-uaf* is an ELF file, which is mainly run on the Linux operating system. The interaction of the program is through the command line, and it mainly receives data from STDIO. The program contains a use-after-free vulnerability. A vulnerable structure in the program can continue to be used after being freed, and the vulnerable structure contains function pointers, which will be called during subsequent executions. After the target vulnerability structure is freed, other heaps can reoccupy it through allocation functions, and overwriting the function pointer in the vulnerability structure.

Heap-fmt The *Heap-fmt* is an ELF file, which is mainly run on the Linux operating system. The interaction of the program is through the command line, and it mainly receives data from STDIO. The program contains a format string vulnerability. External input can directly affect the format string parameters on the heap memory. It can only trigger the format string once.

The evaluation tests the benchmark in the environment with a combination of various exploit mitigations. The ASLR mechanism bypassing needs complex methods (such as heap spraying, address disclosure, etc.), which often requires unique software types, strong vulnerability capabilities, and high PoC quality for real software. Therefore, the automation of ASLR bypassing is still an academic difficulty, and there is currently little work discussing the bypass of ASLR on real-world software, and we turn off the ASLR exploit mitigation on the evaluation of real-world applications.

5.3. Overview of Results

We evaluated based on the above benchmark and conducted 54 sets of experimental evaluations for different software and different exploit mitigations and compared them with the **Rex** [8], which is one of the state-of-the-art open-source AEG solutions.

In the evaluation, some security mechanisms in the specific application were not considered. For example, Bof, Fmt-3, Heap-uaf, and Heap-fmt were not tested to bypass the SAFESEH mechanism, which is not supported by Linux. Free MP3 CD Ripper 2.6, Free WMA MP3 Converter 1.8, etc. were not tested on the environment without GS safety mechanism, and Mini-stream Ripper 2.7.7.100 was not tested on the environment with the GS safety mechanism because the software is not open source and only the binary program can be obtained. Therefore, the security mechanism cannot be turned off or on after the source binary program is compiled.

The overview of the evaluation is shown in Table 3. *AEMB* was able to successfully generate the EXP in 11 applications and successfully bypassed 31 groups of exploit mitigation combinations, 24 of which were failed to bypass by Rex.

In addition, we also tried to bypass the ASLR exploit mitigation for the test programs, which was not considered by Rex. The results are shown in Table 4. A total of 12 experiments were conducted, and S2MAB was able to successfully generate EXP in five of them.

Table 3. List of results evaluated with *AEMB* and open-source approach Rex on benchmarks. The first row of each column represents the experimental results of *AEMB*, the second row represents the experimental results of Rex, and the gray entries represent the cases where *AEMB* can be exploited but Rex cannot.

Program	Solution	Null	DEP	GS	GS + SAFESEH	DEP + GS	DEP + GS + SAFESEH
bof	**AEMB**	**Y**	**Y**	**N**	**-**	**N**	**-**
	Rex	Y	Y	N	-	N	-
Fmt-3	**AEMB**	**Y**	**Y**	**Y**	**-**	**Y**	**-**
	Rex	N	N	N	-	N	-
BabyCookie	**AEMB**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**
	Rex	N	N	N	N	N	N
Heap-func	**AEMB**	**Y**	**N**	**Y**	**Y**	**N**	**N**
	Rex	N	N	N	N	N	N
Heap-uaf	**AEMB**	**Y**	**Y**	**Y**	**-**	**Y**	**-**
	Rex	Y	Y	Y	-	Y	-
Heap-fmt	**AEMB**	**Y**	**Y**	**Y**	**-**	**Y**	**-**
	Rex	N	N	N	-	N	-
Free MP3 CD Converter 1.8	**AEMB**	**-**	**-**	**Y**	**Y**	**Y**	**Y**
	Rex	-	-	N	N	N	N
Free WMA MP3 Converter 1.8	**AEMB**	**-**	**-**	**Y**	**Y**	**N**	**N**
	Rex	-	-	N	N	N	N
Adrenalin Player 2.2.5.3	**AEMB**	**-**	**-**	**Y**	**Y**	**Y**	**Y**
	Rex	-	-	N	N	N	N
AudioCoder 0.8.18	**AEMB**	**-**	**-**	**N**	**N**	**N**	**N**
	Rex	-	-	N	N	N	N

Table 3. *Cont.*

Program	Solution	Null	DEP	GS	GS + SAFESEH	DEP + GS	DEP + GS + SAFESEH
Nidesoft MP3 Converter 2.6.18	**AEMB**	**-**	**-**	**N**	**N**	**N**	**N**
	Rex	-	-	N	N	N	N
Mini-stream Ripper 2.7.7.100	**AEMB**	**Y**	**Y**	**-**	**-**	**-**	**-**
	Rex	N	N	-	-	-	-
DNSTracer 1.9	**AEMB**	**Y**	**Y**	**N**	**-**	**N**	**-**
	Rex	Y	Y	N	-	N	-

Table 4. List of results evaluated with *AEMB* with ASLR exploit mitigation.

Program	bof	Fmt-3	BabyCookie	Heap-func	Heap-uaf	Heap-fmt
ASLR	Y	N	Y	Y	N	N
ASLR+DEP	Y	N	N	N	N	N

5.4. Analysis of Results

AEMB can bypass the common DEP, GS, and SAFESEH exploit mitigations and it can successfully exploit 29 cases that REX did not. Rex can generate ROP based on program context, but it cannot bypass the GS and SAFESEH mechanisms. However, *AEMB* gives more consideration to other exploit mitigations. In addition, Rex does not fit well with Windows and therefore cannot generate code for Windows applications, but *AEMB* has been adapted forWindows and Linux platforms. In summary, compared to Rex, *AEMB* has stronger capability for exploit-mitigation bypassing and a wider application range.

There are three main reasons for the failure of some cases: 1. symbolic execution platform and symbolic execution performance limitations; 2. the program running environment being unable to meet the exploit mitigations bypassing conditions; 3. functional limitation of *AEMB*.

The failure cases due to the symbolic execution platform and symbolic execution performance limitations are *AudioCoder* and *Nidesoft MP3 Converter*. The reasons are as follows:

- *AudioCoder* is complicated. When the system automatically exploits its vulnerability, it still cannot produce results until all memory resources are consumed. When symbolic execution analyzes this application, problems such as constraint explosions are caused due to the large program scale and complex business logic, so the system cannot generate EXP.
- The input of *Nidesoft MP3 Converter* vulnerabilities needs to be copied and pasted into the software window. Therefore, limited by the symbolic execution platform, the system cannot symbolize this type of inputs and cannot perform symbolic execution analysis on it, so it cannot generate EXP. At present, the system can only symbolize file-type input and STDIO input. Therefore, the input that triggers vulnerabilities such as network sending, mouse clicks, etc. cannot be analyzed by the system, and the system cannot automatically generate the EXP of the above-mentioned input methods.

Since the program's operating environment cannot meet the exploit mitigation bypass conditions, some cases failed, such as *bof* with GS, *Heap-func* with DEP, and *Free WMA MP3 Converter* with DEP + GS.

Since the Linux system does not support the SEH mechanism, it cannot meet the GS bypassing conditions. Therefore, *AEMB* cannot generate the EXP for *bof* with GS.

For heap-related vulnerability, DEP bypassing often involves stack-migration operations. However, at the moment of hijacking EIP, no ROP gadget conforms to the stack migration in the running environment of *Heap-func*, so *AEMB* cannot generate the EXP for *Heap-func* with DEP.

Similar to *Heap-func*, the *Free WMA MP3 Converter* failed because there is no suitable ROP gadget to adjust the stack.

The case of exploit failure due to system function limitations is mainly reflected in the bypassing of the ASLR. However, the bypassing method for ASLR is mainly passive, using known addresses or using addresses in memory pages that are disabled for randomization. The system does not support the inclusion of active address leaking [23], which leads to the failure of the experiments in cases where the exploit conditions cannot be met in the known addresses.

5.5. Exploit Generation Time

We recorded and organized the time consumed in each case, as shown in Figure 8.

The prototype system can automatically bypass the exploit mitigations of the above test program at an average speed of 7.1 s (excluding the time of system startup and interface loading). The time efficiency is acceptable.

It often consumes more time when bypassing the exploit mitigations under the Windows system, whereas the time consumption under the Linux system is smaller. Real-world applications consume more time than test programs, mainly due to the scale and complexity of the cases. When multiple exploit mitigations are turned on, the bypassing time will be longer.

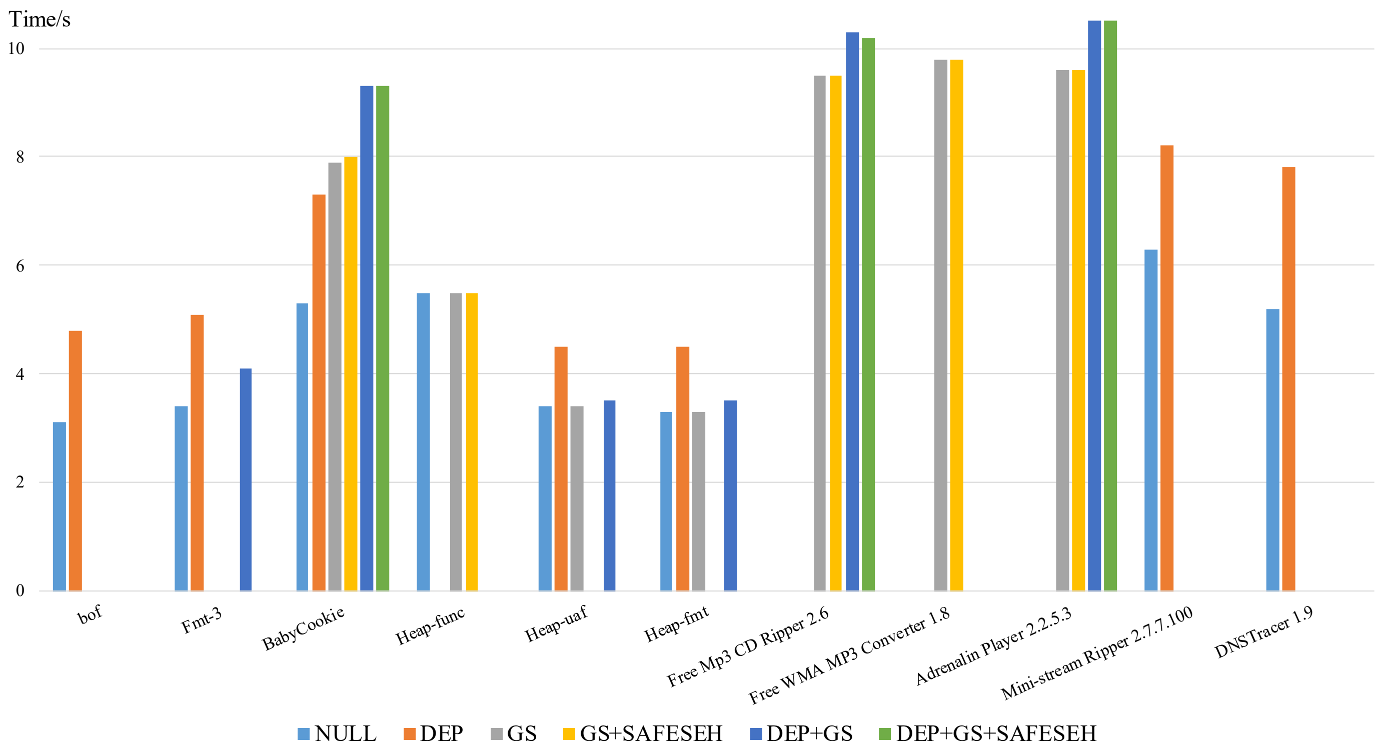


Figure 8. The time efficiency evaluated with *AEMB* on benchmarks.

6. Discussion

AEMB overcomes some of the dilemmas facing current research into bypassing of exploit mitigations. However, *AEMB* still has some limitations and issues that need to be overcome, so we discuss the *limitations* and *future work*.

6.1. Limitations

AEMB can bypass some of the exploit mitigations in a short time, but it still has some problems:

- **imitations of Exploit Techniques.** There are many different types of vulnerability and their corresponding exploit techniques. However, *AEMB* currently does not support so many exploit techniques, resulting in the failure of some types of vulnerabilities exploit automatically. For example, *Unlink Attack* technology [24] is one of the techniques for heap vulnerability exploit, but due to the complex attack process of this technology, *AEMB* has not completed its modeling and automation.

- **Limitations of Passive Bypassing Method.** The current method of bypassing the vulnerability exploitation mitigation mechanism in *AEMB* is to use PoC information to passively complete the mitigation mechanism bypass. However, in many cases, it is necessary to actively discover a new execution path to complete the mitigation mechanism bypass. For example, in bypassing ASLR, actively leaking addresses [23] or heap spraying [25] requires a new execution path, except for the execution path in PoC. Unfortunately, *AEMB* cannot yet discover new execution paths.

6.2. Future Works

In the future, we will conduct further research on the above limitations.

- **Increase in Exploit Techniques.** More exploit techniques mean wider system applicability. We should analyze and model more exploit techniques. In this way, the system can automatically exploit more vulnerabilities and can use different technologies to bypass the exploit mitigations.
- **Exploration of Exploit Primitives.** The exploit primitives are used to compose EXPs. Although many primitives cannot cause such serious harm, they still play a significant role in exploiting vulnerabilities. Therefore, it is necessary to scientifically define and search for these exploit primitives in the program. This can also help us automatically bypass more exploit mitigations.
- **Combination of Exploit Primitives.** After obtaining the exploit primitives, they need to be combined according to specific rules. These rules are often abstracted from expert experience and should reasonably guide the combination of exploit primitives. In addition, the splicing between exploit primitives is often not straightforward, and some techniques need to be used to operate them.

7. Related Work

7.1. Automatic Exploit Generation

The APEG [5] is a patch-based solution, which is an early attempt at AEG. It focuses on the location of the patch, but this method does not deeply study the exploitation. The AEG [4] solution uses source code assistance to extract the path constraints that trigger the vulnerability. The Mayhem [7] solution uses dynamic instruction instrumentation and taint analysis technology to detect whether jump instructions such as *call* and *jmp* are controlled by input and then determines whether the execution control flow of the program can be directly affected by external input; finally, it uses symbolic execution to generate vulnerability verification code. The CRAX [6] solution is implemented based on the S2E [12] symbolic execution platform. It dynamically analyzes the PoC that triggers program crashes, improves symbolic execution efficiency, and can automatically exploit vulnerabilities based on format strings, stack overflows, etc.

Rex [8] is an open-source AEG solution and it is one of the components of machaPhish that won third place in the 2018 CGC finals. Refs. [26,27] contributed to the discovery of exploit primitives. Ref. [28] can automatically generate multiple EXPs for a vulnerable program using one corresponding abnormal input. Ref. [29] analyze the characteristics of vulnerabilities and then propose to generate EXPs via the use of several proposed attack techniques that can produce a shell based on the detected vulnerabilities.

Revery [10] attempts to exploit the vulnerability provided with non-exploitable PoCs and proposes a novel layout-oriented fuzzing and a control-flow stitching solution. It also contributed to the exploitation of heap memory vulnerabilities. Maze [30] manipulates proof-of-concept (POC) samples' heap layouts. It models the heap layouts techniques (such as heap feng shui), and implements automated heap layout manipulation.

However, none of the above AEG solutions have separate discussions and studies on the bypassing of exploit mitigations.

7.2. Bypassing of Exploit Mitigations

Software security mechanism mitigation technology can mitigate exploit attacks to a certain extent. Typical security mechanisms include Stack Guard [15], DEP (Data Execution Prevention) [13], ASLR (Address Space Layout Randomization) [31], and CFI (Control -Flow Integrity) [32]. The research on automated countermeasures against security mechanisms is an important link in the automatic exploitation of software vulnerabilities.

Reference [33] can automatically generate ROP payloads for a given binary to exploit. The authors of [34] developed a new technique called data-flow stitching, which systematically finds ways to join data flows in the program to generate data-oriented EXPs.

R2dlAEG [35,36], through the combination of the basic method of controlling flow hijacking and the automatic realization of the Return-to-dl-resolve process, aiming at control-flow hijacking vulnerabilities, it realizes the automatic generation of EXP that can bypass the ASLR and NX protection mechanisms.

VScape [37] points out that a wide range of virtual call protections, which do not break the C++ ABI (application binary interface), are vulnerable to an advanced attack COOPLUS, even if the given vulnerabilities are weak. It assesses the effectiveness of virtual call protections against this attack.

The above-mentioned work has researched the bypassing of exploit mitigations, but they have not systematically studied the automatic bypassing of exploit mitigations. In addition, there is no solution that studies the GS and SAFESEH, etc. in Windows.

8. Conclusions

In this paper, we propose a novel AEG solution *AEMB* to automatically bypass the exploit mitigation mechanism. We first employ *exploit mitigations bypassing* to generate the exploit mitigations bypassing payload and then use symbolic execution to achieve the generation of working EXPs. We evaluated *AEMB* on real-world applications and tested programs with six different combinations of exploit mitigations. Results showed that *AEMB* is much more effective than previous works in bypassing exploit mitigations. Furthermore, *AEMB* is efficient because it can finish the task in 7.1 s on average.

Author Contributions: Conceptualization, R.W. and Z.P.; methodology, R.W. and Z.P.; software, R.W. and M.Z.; validation, R.W.; investigation, R.W.; resources, Z.P. and F.S.; writing—original draft preparation, R.W.; writing—review and editing, Z.P., F.S. and R.W.; supervision, Z.P. and M.Z.; project administration, Z.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by National Key Research and Development Project of China (No. 2017YFB0802900).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: We would like to sincerely thank all the reviewers for your time and expertise on this paper. Your insightful comments help us improve this work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lyu, Q.; Zhang, D.; Da, R.; Zhang, H. ReFuzz: A Remedy for Saturation in Coverage-Guided Fuzzing. *Electronics* **2021**, *10*, 1921. [CrossRef]
2. Kim, Y.; Yoon, J. MaxAFL: Maximizing Code Coverage with a Gradient-Based Optimization Technique. *Electronics* **2021**, *10*, 11. [CrossRef]
3. Manès, V.J.M.; Han, H.; Han, C.; Cha, S.K.; Egele, M.; Schwartz, E.J.; Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.* **2019**. [CrossRef]

4. Avgerinos, T.; Cha, S.K.; Rebert, A.; Schwartz, E.J.; Woo, M.; Brumley, D. Automatic exploit generation. *Commun. ACM* **2014**, *57*, 74–84. [CrossRef]
5. Brumley, D.; Poosankam, P.; Song, D.; Zheng, J. Automatic patch-based exploit generation is possible: Techniques and implications. In Proceedings of the 2008 IEEE Symposium on Security and Privacy (sp 2008), Oakland, CA, USA, 18–22 May 2008; pp. 143–157.
6. Huang, S.K.; Huang, M.H.; Huang, P.Y.; Lai, C.W.; Lu, H.L.; Leong, W.M. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability, Gaithersburg, MD, USA, 20–22 June 2012; pp. 78–87.
7. Cha, S.K.; Avgerinos, T.; Rebert, A.; Brumley, D. Unleashing mayhem on binary code. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 380–394.
8. Shellphish. Rex. Available online: https://github.com/angr/rex (accessed on 13 October 2021).
9. Huang, S.K.; Huang, M.H.; Huang, P.Y.; Lu, H.L.; Lai, C.W. Software crash analysis for automatic exploit generation on binary programs. *IEEE Trans. Reliab.* **2014**, *63*, 270–289. [CrossRef]
10. Wang, Y.; Zhang, C.; Xiang, X.; Zhao, Z.; Li, W.; Gong, X.; Liu, B.; Chen, K.; Zou, W. Revery: From proof-of-concept to exploitable. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 1914–1927.
11. Zhao, Z.; Wang, Y.; Gong, X. HAEPG: An Automatic Multi-hop Exploitation Generation Framework. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 89–109.
12. Chipounov, V.; Kuznetsov, V.; Candea, G. S2E: A platform for In-Vivo multi-path analysis of software systems. *Acm Sigplan Not.* **2011**, *46*, 265–278. [CrossRef]
13. Andersen, S.; Abella, V. Memory Protection Technologies. Available online: http://technet.microsoft.com/en-us/library/bb457155.aspx (accessed on 13 October 2021).
14. Aafer, Y. Notes on Non-Executable Stack. Available online: https://web.ecs.syr.edu/~wedu/seed/Labs_12.04/Files/NX.pdf (accessed on 13 October 2021).
15. Sotirov, A.; Dowd, M. Bypassing Browser Memory Protections in Windows Vista. Available online: https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf (accessed on 13 October 2021).
16. Firmino, L. What Is the Difference between Exploit, Payload and Shellcode? Available online: https://www.linkedin.com/pulse/what-difference-between-exploit-payload-shellcode-luiz (accessed on 13 October 2021).
17. De Moura, L.; Bjørner, N. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 337–340.
18. Robertson, C. /SAFESEH (Image has Safe Exception Handlers). Available online: https://docs.microsoft.com/en-us/cpp/build/reference/safeseh-image-has-safe-exception-handlers?view=msvc-160 (accessed on 13 October 2021).
19. Bellard, F. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*; USENIX Association: Berkeley, CA, USA, 2005; Volume 41, p. 46.
20. Cadar, C.; Dunbar, D.; Engler, D.R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI* **2008**, *8*, 209–224.
21. CVE. Available online: https://cve.mitre.org/ (accessed on 13 October 2021).
22. China National Vulnerability Database. Available online: https://www.cnvd.org.cn/ (accessed on 13 October 2021).
23. Serna, F.J. The Info Leak Era on Software Exploitation. Black Hat USA. 2012. Available online: https://paper.bobylive.com/Meeting_Papers/BlackHat/USA-2012/BH_US_12_Serna_Leak_Era_Slides.pdf (accessed on 13 October 2021).
24. Designer, S. JPEG COM Marker Processing Vulnerability. Available online: https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability (accessed on 13 October 2021).
25. Ratanaworabhan, P.; Livshits, V.B.; Zorn, B.G. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In Proceedings of the USENIX Security Symposium, Montreal, QC, Canada, 15–19 August 2009; pp. 169–186.
26. Eckert, M.; Bianchi, A.; Wang, R.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Heaphopper: Bringing bounded model checking to heap implementation security. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 99–116.
27. Yun, I.; Kapil, D.; Kim, T. Automatic techniques to systematically discover new heap exploitation primitives. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Virtual Event, 12–14 August 2020; pp. 1111–1128.
28. Wang, M.; Su, P.; Li, Q.; Ying, L.; Yang, Y.; Feng, D. Automatic polymorphic exploit generation for software vulnerabilities. In *International Conference on Security and Privacy in Communication Systems*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 216–233.
29. Wang, Z.; Zhang, Y.; Tian, Z.; Ruan, Q.; Liu, T.; Wang, H.; Liu, Z.; Lin, J.; Fang, B.; Shi, W. Automated vulnerability discovery and exploitation in the Internet of Things. *Sensors* **2019**, *19*, 3362. [CrossRef]
30. Wang, Y.; Zhang, C.; Zhao, Z.; Zhang, B.; Gong, X.; Zou, W. MAZE: Towards Automated Heap Feng Shui. In *30th USENIX Security Symposium (USENIX Security 21)*; USENIX Association: Berkeley, CA, USA, 2021; pp. 1647–1664.
31. Shacham, H.; Page, M.; Pfaff, B.; Goh, E.J.; Modadugu, N.; Boneh, D. On the effectiveness of address-space randomization. In Proceedings of the 11th ACM conference on Computer and Communications Security, Washington, DC, USA, 25–29 October 2004; pp. 298–307.
32. Abadi, M.; Budiu, M.; Erlingsson, U.; Ligatti, J. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **2009**, *13*, 1–40. [CrossRef]

33. Schwartz, E.J.; Avgerinos, T.; Brumley, D. Q: Exploit Hardening Made Easy. In Proceedings of the USENIX Security Symposium, San Francisco, CA, USA, 8–12 August 2011; Volume 10.
34. Hu, H.; Chua, Z.L.; Adrian, S.; Saxena, P.; Liang, Z. Automatic generation of data-oriented exploits. In Proceedings of the 24th USENIX Security Symposium (USENIX Security 15), Washington, DC, USA, 8–12 August 2015; pp. 177–192.
35. Di Federico, A.; Cama, A.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. How the ELF Ruined Christmas. In Proceedings of the 24th USENIX Security Symposium (USENIX Security 15), Washington, DC, USA, 8–12 August 2015; pp. 643–658.
36. Fang, H.; Wu, L.; Wu, Z. Automatic Return-to-dl-resolve Exploit Generation Method Based on Symbolic Execution. 2019. Available online: http://www.jsjkx.com/EN/10.11896/j.issn.1002-137X.2019.02.020 (accessed on 13 October 2021).
37. Chen, K.; Zhang, C.; Yin, T.; Chen, X.; Zhao, L. VScape: Assessing and Escaping Virtual Call Protections. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual Event, 11–13 August 2021.