2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering

Pangr: A Behavior-based Automatic Vulnerability Detection and Exploitation Framework

Danjun Liu/s per 1st Author
College of Computer
National University of Defense and Technology
Changsha, China
liudanjun12@nudt.edu.cn

Jingyuan Wang, Zelin Rong, Xianya Mi, Fangyu Gai, Tang Yong, Baosheng Wang
College of Computer
National University of Defense and Technology
Changsha, China
{wangjingyuan12, rongzelin12, mixianya09, gaifangyu15, ytang, bswang}@nudt.edu.cn

***Abstract*—Nowadays, with the size and complexity of software increasing rapidly, vulnerabilities are becoming diversified and hard to identify. It is unpractical to detect and exploit vulnerabilities by manual construction. Therefore, an efficient automatic method of detecting and exploiting software vulnerability is in critical demand.**

This paper implements Pangr, an entire system for automatic vulnerability detection, exploitation, and patching. Pangr builds a complete vulnerability model based on its triggering behavior to identify vulnerabilities and generate *exp* or exploit schemes. According to the type and feature of the vulnerability, Pangr can generate the specific patch for the software. In the experiment, we tested 20 vulnerable programs on 32-bit Linux machine. Pangr detected 16 vulnerabilities, generated 10 exp, and patched 14 programs.

Keywords—automatic detection; automatic exploit generation; software security; automatic patching

I. INTRODUCTION

With the development and popularization of machine learning and big data processing technologies, the research and application of artificial intelligence have entered a new upsurge. Cyberspace and software security is highly sensitive to new technologies. In 2010, Stuxnet [2] infected more than 45,000 networks worldwide. In 2017, WannaCry ransomware [3] exploited the risk of Vulnerability "EternalBlue" from NSA leakage to spread across at least 150 countries worldwide, and more than 100,000 machines were infected. The outbreak of these events also reveals the gradual emergence of network threats and the trend of intelligent development. The strength of artificial intelligence is that they can find and fix bugs much faster than human beings, and automating this task can contribute to greater system and application security at a lower cost, dramatically increasing the agility of network defense.

DARPA organized the world's first Cyber Grand Chanllenge [4] at Defcon 2016 in the United States [4]. The goal of CGC is to establish an automated offenses and defenses system that can detect, exploit and repair software vulnerabilities, in contrast to the current vulnerability-based software security offensive and defensive which relies heavily on people. Seven teams participated in the final, in which a team named Shellphish posted the source of their tool *angr* [1]. Although computer programs during the CGC Finals and DEFCON CTF have illustrated the ability to make outstanding exploits and fixes, they still cannot compete with human security experts for detecting and exploiting vulnerabilities. CGC can be regarded as a milestone in network security's automatic offensive and defensive. Later, under the guidance of the Chinese Central Network Information Office, a company named YongXinZhiCheng held a CGC-like contest, which is officially called Robo Hacking Game.

angr is a binary code analysis tool that can automate the analysis of binaries. The main challenge to Find and exploit vulnerabilities is the difficulty of visualizing the data structures and the information of control flow in binary code. *angr* is a python-based binary vulnerability analysis framework that integrates a variety of existed analysis techniques (eg, KLEE [5] and Mayhem [6]), which performs binary and system state simulation by loading and analyzing a binary. These techniques include static analysis (Control Flow Graph [7], Value Flow Graph, Backward Slicing, etc.), dynamic analysis (symbolic execution [8], debugging) and constraint solving [9].

We create Pangr, a framework of automatic exploit system. *angr* is just a platform to perform symbolic execution, so it cannot find any vulnerability. Pangr adopts *angr*'s symbolic execution to model behavior of vulnerability conveniently. Pangr can analyze the behavior of binaries and perform automatic detection and exploitation of format string, stack overflow and heap overflow vulnerability, and generate the corresponding protection schemes. Tools such as AEG [10], APEG [11], Mayhem [6], PolyAEG [13] failed to exploit heap overflow vulnerability and did not consider repairing it. Pangr also automatically identifies functions and exploits the vulnerability in a variety of ways.

II. CHALLENGE

The development of Pangr faces many challenges. One is about operating environment. Target binaries are running on a commonly used operating system (binaries are compiled and tested in 32-bit Linux Centos 6.5 version), and it is different from DECREE system used in DARPA CGC, a simplified version of common operating system. As a result, the

Sponsored by National Natural Science Foundation of China (61472437)

vulnerability exploiting approach of tools in CGC is relatively simple. In a real Linux system, the exploitation becomes more complicated.

Second problem is about state explosion. As we know, the number of path branch in a binary grows exponentially. Pangr adopts the technique of symbolic execution in *angr*, and symbolic execution will execute every path branch of a binary. That causes a state explosion. Strategies adopted by *angr* are veritesting [14] and function summary. However, they can only mitigate the state explosion. To solve the problem of state explosion, we must solve the problem of function identification first, and adopt a new path optimization strategy.

The third problem is related to vulnerability detection. In the past, AEG tools stopped when it found one vulnerability. Pangr's goal is to detect all vulnerabilities in the target binary as much as possible, which makes sense for software security defense. In addition, how to classify vulnerabilities correctly is also a problem. The VUzzer [15] adopted !Exploitable [16] to analyze the generated crash, but the result is not satisfactory, and most of the discovered crashes are unknown. Pangr needs to determine the type of vulnerability for the target binary during detection so that different exploits can be implemented according to the specific vulnerability type.

The fourth is the problem of exploiting vulnerability. In the past, some AEG tools only got Proof of Vulnerability (PoV), which was only proved in theory, and they did not really exploit it. Some tools like APEG and PolyAEG could exploit stack overflow, format string, but did not take the ShellCode layout into account. For example, what if there is no sufficient space for ShellCode or there is a significant pointer on the stack. None of the tools above can exploit heap overflow vulnerability.

The last problem is about defense technology. We have to ensure software security without source code. Most of the time, if you find a vulnerability, you have to wait for the vendor to fix the vulnerability. That would take a long time. During the waiting time, the vulnerability might have already been exploited. Previous AEGs rarely consider the issue of defense, some are violent patching, which may affect target binary's functionality itself. How to generate correct defense rules or intelligent patch, which does not affect the functionality of the binary, is a serious problem.

In order to diversify exploits, Pangr turned off stack protection (DEP) and address space layout randomization (ASLR). DEP can be bypassed by Return-oriented programming [34]. ASLR can be bypassed by leaking vulnerability. Neither of them is the key point of this paper. Successful situations of exploiting are divided into five types. One is crash. The second is to set the EIP as a specified address. EIP is a pointer to an address where the program is executing. The third is to read at any address. The fourth is arbitrary address write and the final goal is to obtain a shell.

III. OVERVIEW

Pangr is composed of five elements. As Figure 1 shows, after a binary gets an input, the crash engine tests the binary and finds a new input that can make the program crash. The crash data will then be sent to the test engine and vulnerability analyzer to check its validity and perform further analysis. The initializer preprocesses the binary file before symbolic execution. The vulnerability analyzer then uses symbolic execution to detect vulnerability exploitable based on the preprocessed information. If an exploitable vulnerability is found, it will be sent to vulnerability exploiter to generate exp. After that process is completed, defense engine generates a defense rule or a binary patch.

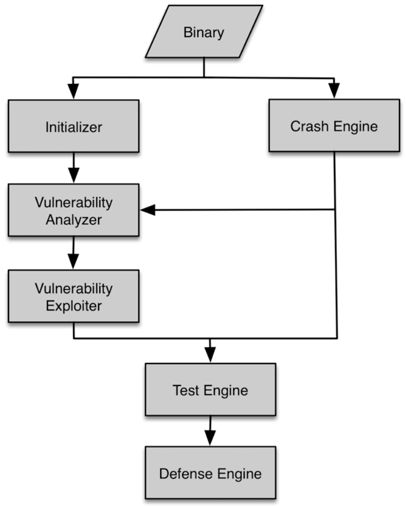

Figure 1. Architecture

This article has following 4 contributions:

- It proposes behavior-based vulnerability detection modeling, which can detect and triage vulnerability effectively. This behavior-based modeling can be used to detect heap overflow vulnerability and generate corresponding exploiting scheme. Up to now, no academic institution can exploit heap overflow vulnerability automatically.
- The system can automatically identify library functions, effectively reducing the complexity of symbol execution.
- This article has achieved different level of vulnerability exploitation. It defines five criteria for successful exploiting, crash, EIP hijacking, arbitrary address read, arbitrary address write and a shell returning.
- Pangr can generate defense rules (length rules, data rules) and a patch for corresponding binary according to the program's vulnerability type.

IV. AUTOMATIC VULNERABILITY DETECTION

The stage of vulnerability automatic detection involves three engines, crash engine, initializer and vulnerability analyzer. Crash engine is responsible for binary fuzzing [32]. If a crash is found, it will be sent to the vulnerability analyzer for further analysis and to the test engine for testing. Initializer is responsible for some initializing work, such as the function identification and function hooking work. Vulnerability analyzer will further detect and analyze the vulnerability. In the rest of this section, this paper will introduce three engines and the technology used in detail. Note that, this article takes 32-bit Linux programs for example.

A. Crash Engine

Crash engine uses Symbolic-execution assisted fuzzing, which refers to Driller [17]. Symbolic-execution assisted fuzzing is based on AFL fuzzer [18] and *angr* that serves as a symbolic tracker. By monitoring AFL executes, Driller can decide when to begin symbolic tracking of specific inputs generated by AFL. When AFL fails to find a new state transition after a round of input mutation, it calls *angr* to perform symbolic execution, tracing the specific input provided by the AFL. Whenever a conditional instruction such as "cmp" is encountered, there might be a new branch. If the new branch is not known to AFL, it will invoke the SMT solver [9] to generate an input to reach the new path and feed the input back to the AFL. Then it continues the fuzzing test. The disadvantage of fuzzing is that only shallow paths are generally explored, for many generated inputs from AFL cannot pass the check of conditional instructions at all. Symbolic- assisted fuzzing takes advantage of symbolic execution on semantic understanding, and fuzzing's short executing time.

B. Initializer

Initializer is mainly responsible for preparing the symbols before symbolic execution. As we all know, symbolic execution's biggest problem is state explosion. In a binary file, with the execution of the program, the program will encounter a lot of conditional instructions, resulting in an explosive increase of branch. *angr*'s symbolic execution will execute each branch. If there are too many branches, the time consumed will increase, and the memory will be quickly depleted to preserve the status of every branch. Pangr needs to avoid exploring unrelated code, such as the code in link library, as much as possible. Thus Pangr needs to identify library functions in statically compiled binaries accurately.

Byteweight [20] proposed a more accurate method of function identification than Hex-Ray's Interactive Disassembler Pro (IDA), but only detected the existence of a function and Byteweight was not sure what it was. John McMaster [19] mentioned in the paper that using the FLIRT signature algorithm to detect known malware works well.Pangr's method is to calculate a CRC16 checksum for each function, and then compare the CRC values of the two functions to figure out whether they are the same function, which can solve the problem of large search volume. If a function does not contain variables, then you can take the first N bytes of the function to do CRC check. Pangr sets the corresponding variable to 0 or does not deal with it.

Initializer works by generating CRC16 signature for corresponding library functions of the target binary and then identifying library functions based on the generated signatures. That has been reported to the *angr* team and they have begun to improve *angr*.

C. Vulnerability Analyzer

Vulnerability analyzer is the most important part in Pangr, which applies the novel behavior-based modeling. Pangr models the behavior of format string, stack overflow and heap overflow vulnerability, respectively, and exploits the intrinsic semantics triggered by vulnerabilities during symbolic execution to find valuable loopholes. After finding a vulnerable point, which is conducive to the vulnerability exploitation later, the vulnerability analyzer records input values and the context information, such as registers, stack, heap and environment variables, etc. In the rest of this section, this paper will introduce the novel behavior-based modeling specifically.

1) Format String Vulnerability

The format string vulnerability [21] [22] typically occurs in the following types of functions:

TABLE I. FUNCTIONS THAT CAN CAUSE FORMAT STRING VULNERABILITY

Function name	Calling convention	Format argument's location
printf	int printf(const char *format,...);	1
sprintf	int sprintf(char *buffer, const char *format, [argument] …);	2
snprintf	int snprintf(char *str, size_t size, const char *format, …);	3

Once the format parameter is affected by input value, the attacker may control the format parameter. It means that the target binary contains format string vulnerability. According to the characteristics of compiling rules, general format parameter is a value that has already been determined when a program is written. It is stored in a binary section such as *.data* and *.bss*, so it is in the program's own address space. Generally, user's input value is located in the stack or heap, and stack and heap space address is beyond the scope of the program itself. Thus, in a 32-bit x86 system, the data dependency graph [7] [31] is generated first and then the corresponding format parameter on the stack is checked only when the program executes to a suspicious library function point. If the format parameter is affected by input, then the binary can be judged as format string vulnerability, and the context information is recorded at this time.

2) Stack Overflow Vulnerability

Stack overflow [23][24] is a kind of vulnerability caused by stack buffer overflow. This may cause data overrides or execution flow hijacking. As is shown in Listing 1, *main()* function inputs a string of which length is greater than 20. Then it calls the *vul()* function, and *str* string overflows *buf* buffer. The stack structure is shown in Figure 2.

Anyway, the final result of a stack overflow is to make the *ip* pointer point to a user-controllable address. Then during symbolic execution, it is most likely to be stack overflow vulnerability once Pangr finds out that *ip* is affected by input. Because format string and heap overflow vulnerability cannot directly hijack the program control flow. Pangr also records the context information.

```
void vul(char *str)
```

```
{
    char buf[20]="";
    char *ptr;
    ptr=malloc(0x10);
    strcpy(buf,str);
    read(0,ptr,0xf);
}
int main()
{
    char str[100];
    read(0,str,100);
    vul(str);
    return 0;
}
```

Listing 1. Stack overflow vulnerability

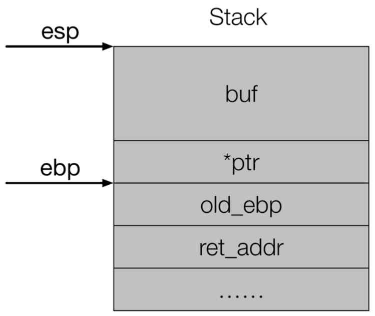

Figure 2. The layout of the stack

*3) **Heap Overflow Vulnerability***

Heap overflow [25][26] is a kind of vulnerability caused by a heap buffer overflow. As it doesn't cause crash, so it is difficult to detect. This may cause next heap's content or heap structure to be overwritten. This type of vulnerability is difficult to detect, for it cannot directly lead to a crash, and conditions to trigger it are far more complicated. As is shown in Listing 2, the *main()* function first allocates a heap block and then invokes the *get_str()* function to read data to the heap block. Obviously, if the requested heap block has only 8 bytes, the input of which length is 0x1f is copied to the heap block, causing an overflow. Pangr's strategy is that, in the symbolic execution process, once it encounters a program point where *malloc()* function is called, it records the address of heap block and size into the *GlobalChunkArr* array. Just like *malloc(num)*, if *num* is a certain value, then it can be directly recorded. If *num* is a tainted value (affected by input value), Pangr uses SMT solver to calculate the smallest value and record it. And then it continues symbolic execution. If there is an instruction writing to a chunk recorded in *globalChunkArr* array, it might be a string copying operation. The string copying operation is generally implemented by library functions or loops, as is shown in Listing 2. These library functions include *read*, *strcpy*, *memcpy*, *sprint*, *snprintf* etc. During the copying process, once Pangr finds that an access crossing the chunk boundary, Pangr regards it as heap overflow vulnerability.

```
int get_str(char *str)
{
    int i=0;
    char buf[0x20]="";
    read(0,buf,0x1f);
    strcpy(str,buf);
    return 0;
}
int main()
{
    int size;
    char *ptr;
    printf("Please input the chunk size:");
    scanf("%d",&size);
    if (size<8) size=8;
    ptr=malloc(size);
    get_str(ptr);
    return 0;
}
```

Listing 2. Heap overflow vulnerability

V. AUTOMATIC VULNERABILITY EXPLOITATION

Methods used in vulnerability exploiter are divided into three kinds. In fact, there are many ways to exploit, and some ways may be relatively rare. This paper only discusses more general ways. Pangr's scalability is good, so you can easily add new models later.

A. Format String Vulnerability

Exploiting format string vulnerability is mainly based on three features of format string. One is to use "%s" to read data from the target memory address. The second is to use width modifier "$" to control the output number of characters. The third is to use "%n" to write the output number of characters to the target memory address. Note that "%n" writes 4 bytes at a time, "%hn" writes 2 bytes at a time, and "%hhn" writes 1 byte at a time. For example, "%8$s" can read the value at the address that the eighth parameter on the stack points to.

B. Stack Overflow Vulnerability

There are two ways to exploit stack overflow vulnerability. One is to use ShellCode, and the other is to construct ROP chain, which refers to Q [27].

There are two problems with ShellCode building. One is that the stack may contain important data and a reference error occurs if it is overwritten. As is shown in Listing 1, if *strcpy()* results in *ptr* being overwritten, a pointer reference error will occur before the *vul()* function returns. Thus you cannot successfully hijack control flow. The second problem is that stack space is not enough to put down a whole ShellCode. As is shown in Figure 2, *buf* and the area behind *ret_addr* can hold ShellCode. Just *buf* area is obviously not enough. Pangr implements storage of segmenting ShellCode to solve the problem of space fragmentation. As is shown in Figure 3, ShellCode = ShellCode0 + ShellCode1. There are two kinds of ShellCode arrangement solutions for the vulnerability in Listing 1. When automatically arranging ShellCode, Pangr can segment the original ShellCode intelligently, in order to make full use of the controllable space. At the same time, Pangr should pay attention to stack information recovery. **ptr* pointer must be a valid address, but cannot be replaced by non-meaningful padding characters. Directly covering the return address will be able to achieve *eip* controlling, and ShellCode can lead to crash, arbitrary address read, arbitrary address write and shell access.Furthermore, another tool by ShellPhish team named *angrop* can be used to construct ROP chain to bypass DEP protection [12].

C. *Heap Overflow Vulnerability*

Heap overflow does not directly cause the return address being covered, so it is very complicated to exploit it. There is no research institutes or academic articles having achieved automating exploiting heap overflow vulnerability.

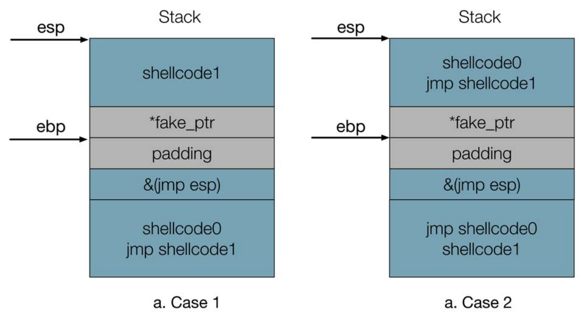

Figure 3. Layout of ShellCode

The difficulty to exploit heap overflow reflects on two main aspects. First, heap management program is very complicated. Heap management program is a management procedure between system and user, and it is to achieve efficient management of heap space. It reduces user to frequently allocate and release heap space, improving executing speed and resource utilization. Second, there are a lot of security checks in the heap management program. Once Pangr fails to place the correct data in the corresponding location when forging a heap chunk, it cannot pass the heap security check.

TABLE II. FEATURES OF HEAP OVERFLOW VULNERBILITY

Feature	**Double Free**	**Forging chunk**	**unlink**	**Shrinking free chunks**	**House of spirit**	**House of lore**	**House of force**	**House of einherjar**	**Overlapped chunk**
Repeatable malloc	✔	✔	✔	✔	✔	✔	✔	✔	✔
Repeatable edit	✘	✔	✔	✔	✘	✘	✘	✔	✔
Chunk malloc size	Fast bin	Fast bin	any	Large chunk (>=256)	Fast bin	Small chunk	Top chunk (big)	Not limited	Not limited
Overflow length	any	any	>=5	>=1	any	>=32 or UAF	>=8	>=5	>=5
free()	✔	✔	✔	✔	✔	✔	✘	✔	✘
Repeatable free()	✔	✘	✘	✘	✘	✘	✘	✘	✘
Use-after-free	✘	✔	✘	✘	✘	✔	✘	✘	✘
Controlling chunk pointer	✘	✘	✘	✘	✔	✘	✘	✘	✘
Controlling other space	✘	✘	✘	✘	✔(stack)	✔(stack)	✘	✔(stack)	✘
Existing a pointer to malloc chunk (not on stack)	✘	✘	✔	✘	✘	✘	✘	✘	✘

Taking the flexibility and complexity of the heap exploiting into account, Pangr does not directly produce an exp that can be used, but according to characteristics of the vulnerability, it automatically generates an exploiting scheme to guide people to write a valid exp. Pangr divides methods of heap exploiting into 9 types, and defines 10 kinds of feature. Then Pangr generates exploiting schemes based on features the vulnerability has. If the vulnerability satisfies too many features defined, then there might be several exploiting schemes. Specific method of exploiting heap vulnerability is already described in [28]. The features defined are shown in Table 2.

9 schemes above are all by forging structures of heap chunks to cheat the heap management program to write a given value at the target address. Then you can change normal function address that will be called on .got table to system address. You can choose those normal functions like atoi or free. Finally, you can get a shell access.

VI. AUTOMATIC VULNERABILITY PATCHING

Defense engine uses two strategies. One is to generate filtering rules, using regular expressions to filter the input. The advantage of filtering rules is that they do not affect the program's own function. However, if the binary is an interactive program, usually one interaction triggers vulnerability at a certain step. Filtering rules do not only filter out the input at this step, but inputs at each interacting step are filtered out. That may lead to false filtering. The second strategy is automatic patching. You can filter out inputs at certain program points, but the disadvantage is that it may affect binary's function. In the following, Pangr introduces strategies to patch binaries automatically and correctly.

Pangr uses *patchkit* to patch binary. *patchkit* provides a python interface for patching binary files, so that you can directly modify binary code, inject code, and hook functions. According to the vulnerability type detected by vulnerability analyzer, defense engine applies different patch strategies without affecting the functionality of the binary itself. For format string vulnerability, Pangr filters characters such as "%n" and "%s" at the address of the vulnerability such as *printf* function. Pangr can also add a "%s" argument to the vulnerable function. For stack overflow vulnerability, Pangr modifies its length parameter if the overflow is caused by *strncpy* or *memcpy* functions. Pangr can replace *gets* with *read*, preventing copying overflow or overwriting the return address and other important data. For heap overflow vulnerability, Pangr can limit the copy length based on the chunk size recorded before. Pangr can also apply for more heap space when finding a *malloc* function, and it can replace *free* function with empty instructions to prevent exploiting.

VII. Evaluation

Pangr tests 20 binaries from RHG competition, including various types of vulnerability mentioned above. Each binary's size is about 500K or much more and 20 binaries are 32-bit Linux programs, statically compiled. All receive data from standard input and send the data to standard output. The experiment is performed on an Intel(R) Core(TM) i7-4790 3.60GHz machine with 8GB memory and 64-bit Ubuntu OS which kernel is 4.4.0.

A. *Test crash engine*

The purpose of this experiment is to compare crash engine's test result with other techniques. Crash engine applies fuzzing and symbolic execution like *Driller*. After the experiment, there are three major findings. The test results can be found in Table 3. A total of 15 crashes were found in 20 binaries. Fuzzing found 14 crashes, and SE+fuzzing found 15 crashes. As you can see, in addition to the 15th and 20th binary, SE+fuzzing (Pangr) found vulnerabilities faster than simply using fuzzing, which shows that symbolic execution helps to improve fuzzing's efficiency. SE+fuzzing (Pangr) found a vulnerability in the 19th program, which fuzzing did not find. After a manual analysis of this binary, we find that it encrypts the input and checks the prefix of the encrypted input. Only if an input passes the check can it trigger a stack overflow vulnerability, which means that simply using fuzzing is difficult to make the binary execute a deeper function. Symbolic execution can help fuzzing explore unknown codes, so as to discover hidden vulnerabilities more effectively. SE+fuzzing (Pangr) costs much less time than just fuzzing does when testing complicated binaries such as 1th, 8th, 12th and 16th. It shows that Symbolic-execution assisted fuzzing is more effective to detect vulnerabilities in complicated binaries.

TABLE III. Vulnerabilities Found and Time Spent

Number	Binary name	Vulnerability	Fuzzing	SE+Fuzzing	SE	Exploitable
1	b64_encode_1	Crash	5.0380589962	5.04984593391	X	✘
2	Equation_Parser_bad_index	Crash	109.633465052	25.1422688961	X	✘
3	Equation_Parser_overflow_ROP	Stack	591.431312006	230.345306759	97.9112250805	✔
4	Equation_Parser_overflow	Stack	431.472472906	159.838685989	6.58338212967	✔
5	HTML_filter_INTOverflow_eip_1	None	None	None	None	None
6	HTML_filter_INTOverflow_eip_2	Stack	X	X	X	✘
7	notes_DoubleFree	Heap	X	X	791.403729102	✘
8	Read_Httpd_Log_1	Crash	10.113517046	6.04957199097	X	✘
9	YY_IO_BS_003_ROP	Stack	83.7479410172	55.3512039185	6.92026495934	✔
10	YY_IO_BS_005_eip	Stack	59.330696106	19.1218309402	7.37649106979	✔
11	_81_pwn01	Stack	X	X	X	✘
12	_83_pwn03	Crash	9.07820081711	5.04027295113	X	✘
13	_86_pwn06	Stack	X	X	X	✘
14	_88_pwn08	Stack	691.4743011	685.06931901	13.7531511784	✔
15	_89_pwn09	Format string	57.3135669231	63.3263599873	5.85865688324	✔
16	_90_pwn10	Heap	6.05639815331	5.05384206772	27390.9301628	✔

Number	Binary name	Vulnerability	Fuzzing	SE+Fuzzing	SE	Exploitable
17	_94_pwn14	Stack	378.135900021	19.1248691082	X	✘
18	_95_pwn15	Heap	84.483536005	61.3487529755	29831.2839019	✔
19	_103_pwn23	Stack	X	617.341187954	906.839201837	✔
20	_105_cb	Stack	8807.79094815	10318.6179779	23017.6372128	✔

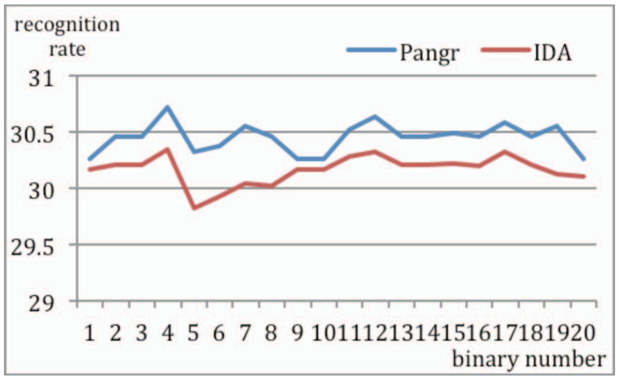

Figure 4. Recognition of Pangr and IDA

B. *Test function recognizing*

The main goal of function recognizing is to improve the speed of symbolic execution, but not to exceed IDA. To test the effectiveness of Pangr on function recognition, we tested these 20 programs with IDA and Pangr respectively. Interactive Disassembler Professional, often referred to as IDA Pro, or IDA for short, is now one of the best static dissembling software for many members of the 0day world and ShellCode security analysts as an indispensable weapon. IDA Pro is interactive, programmable, scalable, multiprocessor and it can run on Windows, Linux, WinCE and MacOS platform for program analysis. Its ability to recognize functions is the best currently available. We compared initializer's ability to identify library functions with IDA and found that IDA cannot identify version libc-2.12's library. When we generate the corresponding 2.12 version of the library function signature, IDA works well. Pangr's average accuracy to recognize library functions reached 30.45%, slightly higher than IDA 30.16%. It is found that IDA does not even recognize some common functions such as *printf*, *scanf* and *memset*, while Pangr can do it.

C. *Test vulnerability analyzer*

Vulnerability analyzer is implemented using pure symbolic execution, of which the purpose is to provide necessary information for vulnerability exploitation, so only exploitable vulnerabilities can be detected. It cannot detect vulnerabilities that are not exploitable in the 20 binaries. For example, 1th, 2th, 6th and 8th are memory access errors caused by an invalid address access. 5th will fall into an endless loop, resulting in collapse, and it cannot be exploited. Program logic in 11th, 12th, 13th and 17th is too complicated that symbolic execution takes up too much memory, and machines with better performance may solve this problem.

One format string vulnerability, seven stack overflow vulnerabilities and three heap overflow vulnerabilities were found out. The heap overflow vulnerability in 7th binary was a vulnerability that traditional fuzzing and symbolic execution techniques could not find out, for heap vulnerability usually cannot cause a crash. In this test case, you can see the advantages of behavior modeling. Vulnerability analyzer is fully implemented using symbolic execution, so its speed depends on characteristics of the vulnerability, the program's own complexity and machine's performance. The test results can be found in Table 3.

To sum up, our behavior modeling successfully found exploitable vulnerability, and even found heap vulnerability that other AEGs couldn't find.

D. *Test vulnerability exploiter*

Pangr successfully generated *exp* for one format string vulnerability and seven stack overflow vulnerabilities, and the relevant *exp* could result in a crash, arbitrary address read, arbitrary address write, *eip* hijacking and shell returning. Pangr also generated relevant exploiting scheme for three heap vulnerabilities. A binary named *notes_DoubleFree* does not meet conditions of any exploiting scheme, so Pangr judged *DoubleFree* as not exploitable. 16th binary has four kinds of exploiting schemes, for it meets four characteristics, repeatable malloc, repeatable free, random malloc size, any byte overflow, and it has a value pointing to the malloc chunk. Four exploiting schemes are Unlink, Shrinking free chunks, House of force (assuming heap address known) and Overlapped chunk. The exploiting scheme of 18th binary is House of force. We manually construct exploiting scripts according to the schemes generated. The scripts achieve crash, arbitrary address write, *eip* hijacking and shell returning in 16th and 18th binary. We cannot achieve arbitrary address read, for the binary itself does not have a read interface.

Defense engine successfully generated patches for 14 binaries and effectively prevent the crash error and remote shell access. Pangr did not find vulnerability in 5th, 6th, 7th, 11th and 13th binary, so it cannot generate relevant patches. 17th binary's vulnerability is caused by *brainfuck* language [33] and this language is unfamiliar to Pangr, so it cannot be automatically patched.

The final test results are shown in Table 4.

TABLE IV. PERCENTAGE OF THE OVERALL TEST RESULT

Item	Percentage
Vulnerability by fuzzing	70%
Vulnerability by SE+fuzzing	75%
Vulnerability by SE	55%

Item	Percentage
Vulnerability by Pangr	80%
Success of exploiting	50%
Success of defense	70%

VIII. Discussion

Pangr's crash engine found 15 vulnerabilities, one more than merely using fuzzing technology, and Pangr is much faster. Initializer engine has a slightly higher recognition rate for non-symbol binaries than IDA, which greatly speeds up symbolic execution. Vulnerability analyzer engine successfully discovered 11 program vulnerabilities, including 3 heap overflow vulnerabilities, of which one heap vulnerability was not found in crash engine. Pangr's brightest spot is using symbolic execution to detect suspicious behavior of binaries and determine the type of vulnerability. The accuracy of this method is higher, for it can effectively identify exploitable vulnerabilities. At the meantime, it has high scalability, so researchers can easily add new models to it based on vulnerability's behavior. Vulnerability exploiter generates corresponding *exp* or exploiting schemes for format string, stack overflow and heap overflow vulnerability. Defense engine effectively repairs vulnerabilities found.

Pangr also has some deficiencies. The first, target binaries are 32-bit x86 programs, so platforms supported need to be expanded. Secondly, the speed is very slow, for Pangr adopts symbolic execution. Thirdly, the process of exploiting heap vulnerability is not automatic enough, for current heap overflow vulnerability is very complicated and diversified. We need to do further testing in the future. Fourthly, vulnerability modeling is not perfect. Models of other vulnerability types remain to be expanded. Fifthly, in terms of vulnerability exploiting, we need to consider bypassing system protection, like address space layout randomization. But DEP and ASLR can be bypassed by return-oriented programming and leaking vulnerability, which is not the key point of our research. Maybe we can do some research on automatic leaking address in the future.

References

[1] Shoshitaishvili Y, Wang R, Salls C, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis[C]//Security and Privacy (SP), 2016 IEEE Symposium on. IEEE, 2016: 138-157.

[2] Karnouskos S. Stuxnet worm impact on industrial cyber-physical system security[C]//IECON 2011-37th Annual Conference on IEEE Industrial Electronics Society. IEEE, 2011: 4490-4494.

[3] Mohurle S, Patil M. A brief study of wannacry threat: Ransomware attack 2017[J]. International Journal, 2017, 8(5).

[4] https://en.wikipedia.org/wiki/2016_Cyber_Grand_Challenge "The Cyber Grand Challenge (CGC) seeks to automate cyber defense process". Cybergrandchallenge.com. Retrieved 17 July 2016.

[5] Ramos D A, Engler D R. Under-Constrained Symbolic Execution: Correctness Checking for Real Code[C]//USENIX Security Symposium. 2015: 49-64.

[6] Cha S K, Avgerinos T, Rebert A, et al. Unleashing mayhem on binary code[C]//Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012: 380-394.

[7] Xu L, Sun F, Su Z. Constructing precise control flow graphs from binaries[J]. University of California, Davis, Tech. Rep, 2009.

[8] Schwartz E J, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)[C]//Security and privacy (SP), 2010 IEEE symposium on. IEEE, 2010: 317-331.

[9] De Moura L, Bjørner N. Z3: An efficient SMT solver[J]. Tools and Algorithms for the Construction and Analysis of Systems, 2008: 337-340.

[10] Avgerinos T, Cha S K, Rebert A, et al. Automatic exploit generation[J]. Communications of the ACM, 2014, 57(2): 74-84.

[11] Brumley D, Poosankam P, Song D, et al. Automatic patch-based exploit generation is possible: Techniques and implications[C]//Security and Privacy, 2008. SP 2008. IEEE Symposium on. IEEE, 2008: 143-157.

[12] Andersen S, Abella V. Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies[J]. 2004.

[13] Wang M, Su P, Li Q, et al. Automatic polymorphic exploit generation for software vulnerabilities[C]//International Conference on Security and Privacy in Communication Systems. Springer, Cham, 2013: 216-233.

[14] Avgerinos T, Rebert A, Cha S K, et al. Enhancing symbolic execution with veritesting[C]//Proceedings of the 36th International Conference on Software Engineering. ACM, 2014: 1083-1094.

[15] Rawat S, Jain V, Kumar A, et al. Vuzzer: Application-aware evolutionary fuzzing[C]//Proceedings of the Network and Distributed System Security Symposium (NDSS). 2017.

[16] Foote J. Cert triage tools[J]. 2013.

[17] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting Fuzzing Through Selective Symbolic Execution[C]//NDSS. 2016, 16: 1-16.

[18] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[19] McMaster J. Issues with FLIRT aware malware[J]. 2011.

[20] Bao T, Burket J, Woo M, et al. Byteweight: Learning to recognize functions in binary code[C]. USENIX, 2014.

[21] Tsai T, Singh N. Libsafe 2.0: Detection of format string vulnerability exploits[J]. white paper, Avaya Labs, 2001.

[22] Newsham T. Format string attacks[J]. 2000.

[23] Kuperman B A, Brodley C E, Ozdoganoglu H, et al. Detection and prevention of stack buffer overflow attacks[J]. Communications of the ACM, 2005, 48(11): 50-56.

[24] Cowan C, Pu C, Maier D, et al. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks[C]//USENIX Security Symposium. 1998, 98: 63-78.

[25] Zeng Q, Wu D, Liu P. Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures[C]//ACM SIGPLAN Notices. ACM, 2011, 46(6): 367-377.

[26] Chien E, Ször P. Blended attacks exploits, vulnerabilities and buffer-overflow techniques in computer viruses[J]. Virus, 2002, 1.

[27] Schwartz E J, Avgerinos T, Brumley D. Q: Exploit Hardening Made Easy[C]//USENIX Security Symposium. 2011: 25-41.

[28] https://heap-exploitation.dhavalkapil.com

[29] Cifuentes C, Van Emmerik M. Recovery of jump table case statements from binary code[C]//Program Comprehension, 1999. Proceedings. Seventh International Workshop on. IEEE, 1999: 192-199.

[30] Sutton M, Greene A, Amini P. Fuzzing: brute force vulnerability discovery[M]. Pearson Education, 2007.

[31] Müller U. Brainfuck–an eight-instruction turing-complete programming language[J]. Available at the Internet address http://en. wikipedia. org/wiki/Brainfuck, 1993.

[32] Prandini M, Ramilli M. Return-oriented programming[J]. IEEE Security & Privacy, 2012, 10(6): 84-87.