IntFinder: Automatically Detecting Integer Bugs in x86 Binary Program

Ping Chen[1], Hao Han[1], Yi Wang[1], Xiaobin Shen[2], Xinchun Yin[2], Bing Mao[1], and Li Xie[1]

[1] State Key Laboratory for Novel Software Technology, Nanjing University, Department of Computer Science and Technology, Nanjing University, Nanjing 210093
{chenping,wangyi}@sns.nju.edu.cn, {maobing,xieli}@nju.edu.cn
[2] College of Information Engineering, Yangzhou University, Yangzhou Jiangsu 225009, China
xcyin@yzu.edu.cn

Abstract. Recently, Integer bugs have been increasing sharply and become the notorious source of bugs for various serious attacks. In this paper, we propose a tool, IntFinder, which can automatically detect Integer bugs in a x86 binary program. We implement IntFinder based on a combination of static and dynamic analysis. First, IntFinder decompiles a x86 binary code, and creates the suspect instruction set. Second, IntFinder dynamically inspects the instructions in the suspect set and confirms which instructions are actual Integer bugs with the error-prone input. Compared with other approaches, IntFinder provides more accurate and sufficient type information and reduces the instructions which will be inspected by static analysis. Experimental results are quite encouraging: IntFinder has detected the integer bugs in several practical programs as well as one new bug in slocate-2.7, and it achieves a low false positives and negatives.

1 Introduction

Integer bug is a notorious bug in programs written in languages such as C/C++, and it can not be obviously witnessed by traditional bug detector. Integer bugs can be classified into four categories: integer overflow, integer underflow, signedness error and assignment truncation. Recently, Integer bug has been increasing sharply, Common Vulnerability and Exploit (CVE) shows that integer overflow and signedness error have been increasing [7]. Besides, in OS vendor advisories, integer overflow has become the second most common bug type among all the bugs.

In recent years, several researches are focused on Integer bugs. Given program source code, there are three approaches to detect Integer bugs. (1)Safe Language, this method either translates the C program into type safe language (such as CCured [21], Cyclone [16]) or uses safe class (such as SafeInt [17], IntSafe [15]). (2) Extension to C compilers, this method inserts checking code for certain operations when compile the source code (such as BLIP [14], RICH [9]). (3) Static source code analysis, this method inspects the whole program to find the suspect instruction (such as LCLint [13], integer bug detection algorithm proposed by Sarkar et al. [11]). Although the works mentioned above may be effective for detecting Integer bugs, they all need source code, which is not available for COTS programs. To prevent integer bugs at binary level, several techniques are proposed, such as UQBTng [25],BRICK [10], IntScope [24] and SmartFuzz [20]. UQBTng [25] is a tool capable of dynamically finding integer overflow in Win32 binaries. BRICK [10] is a binary tool for run-time detecting integer bugs. IntScope is a static binary analysis tool for finding integer overflow. SmartFuzz generates test cases that cause Integer bugs at the point in the program where such behavior could occur. The state-of-the-art techniques of detecting Integer bugs at binary level have several limitations. First, some tools, such as IntScope and UQBTng, can only detect integer overflow. Second, although these works render type reconstruction by using hints from the x86 instruction set, they do not explicitly discuss how to extract this information, and no approach proposes type extraction based on both dataflow and control-flow analysis. Third, the binary static method like IntScope has false positives, because it is difficult to statically reason about integer values with sufficient precision. IntScope leverages symbolic execution only for data that from outside input to the sinks defined by the tool, however, it lacks the information of constraints between inputs and global information [24]. By contrast, the dynamic binary approach such as BRICK [10] and UQBTng [25] may be time-consuming or have high false negatives, because it either checks all the suspect operations including the benign ones or ignores several critical operations. SmartFuzz can generate test cases to trigger Integer bugs, however, without dynamic detecting tools specific to Integer bugs, just using the memory error detecting tool such as Memcheck [22], SmartFuzz would not report certain test cases, which trigger Integer bugs but can not be detected by Memcheck. To sum up, existing binary level detection tools have at least one of the following limitations: (1) ineffective for integer bugs except Integer overflow (2) imprecise type reconstruction (3) high false positives of static analysis (4) time-consuming and high false negatives of dynamic analysis.

To overcome the limitations mentioned above, we implement our tool at binary level, and automatically detect Integer bugs by using static and dynamic analysis. Our paper makes three major contributions:

– We reconstruct the sufficient type information from binary code.
– We propose a systematic method of combining static and dynamic analysis to specifically detect integer bugs in executables.
– We implement a prototype called IntFinder and use it to analyze real-world binaries. Experimental results show that our approach has low false positives and negatives and is able to detect 0-day integer bugs.

The rest of this paper is organized as follows: The characteristic of Integer bugs are described in section 2. In section 3, we present the overview of IntFinder. The implementation of IntFinder is illustrated at section 4. Section 5 provides the evaluation of our tool. Section 6 examines its limitations. Finally, section 7 concludes our work.

2 Integer Bugs

We studied 350 Integer bugs on CVE [5], and noticed that the common root cause of Integer bugs is the mismatch of operand value and its type. We also noticed that exploiting Integer bugs leverages several common mechanisms, which can also be used for us to detect the integer bugs. We summarize the most common tricks as following.

- *Memory Allocation Function*: Memory allocation functions directly deal with the memory space and certainly become the first target for attacker. As the size argument of memory allocation function is unsigned integer. There are two mechanisms to achieve attack. First, through integer overflow/underflow, the size parameter will get smaller or larger value than expected, and the allocated memory will be either insufficient or exhausted. Second, signedness error may lead to bypass the comparison operation, and transfer a negative value to the size argument of memory allocation function.
- *Array Index*: Array index can be regarded as an unsigned integer, and it is often leveraged to compute the offset from the base address to access the memory. When the array index is crafted by Integer bugs, it will facilitate the attacker to access any memory space.
- *Memory Copy Function*: Memory copy function often has the unsigned parameter, and the parameter determines the size of memory which is copied from source operand to destination operand. If the parameter is crafted by Integer bugs, it will lead to buffer overflow.
- *Signed Upper Bound Check*: Signed upper bound check (often comparison operation) can be bypassed by negative values. And later this value will be converted to be a larger values via a signedness error or integer underflow. If the large value is used as an argument of memory allocation or memory copy functions, an attacker may be able to exploit this bug to corrupt application memory.

3 Overview

In this section, we will describe the architecture and working process of IntFinder. There are three main components in IntFinder: (1) Extended type analysis on decompiler (2) Taint analysis tool (3) Dynamic detection tool. Figure 1 shows the architecture.

The work flow of our approach is as below: first, IntFinder leverages the decompiler to translate x86 binary program into its SSA-like intermediate language. Second, we

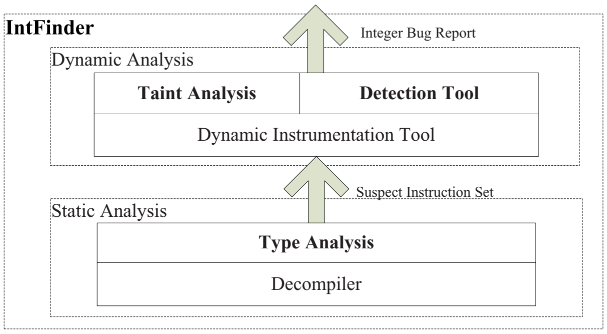

Fig. 1. Architecture of IntFinder

Table 1. Example of control flow based type information

```
[1]    char * buf;
[2]    int size;
[3]    if(...){
[4]       if(size>100)
[5]       {
[6]           ...
[7]       }
[8]       else
[9]         ...
[10]   }
[11]   else{
[12]     ...
[13]   buf=(char *)malloc(size);
[14]   }
```

extend the type analysis of decompiler in order to get more type information from binary code and create the suspect integer bug set. Third, with the suspect integer bug set, we implement the dynamic detection tool combined with taint analysis to further diagnose the instructions in the suspect sets, and determine which are Integer bugs.

– *Type Analysis*: Our type analysis is based on decompiler. It provides the relatively precise type information and gives the suspect instruction set for dynamic testing or debugging. Compensating to existing type information extracted by decompiler, IntFinder further rebuilds the type information based on exploited function and statement just discussed in Section 2. In addition, IntFinder also reconstructs the type information by using both data flow analysis and control flow analysis. Control flow analysis will be benefit because type information of operand may be collected from different basic blocks. Table 1 shows an example of control flow related type information, which contains apparently type conflicting about signed integer variable named *size*, and has potential signedness error. Specifically, at line 4 *size* is illustrated as signed integer, while at line 13 *size* will be illustrated as unsigned integer. Note that statements at line 4 and line 13 are at different basic blocks. Thus, conflicting type information may be missed without control flow analysis.
– *Taint Analysis*: To further reduce the number of suspect instruction and false positives of our tool, we use dynamic taint analysis to select those instructions with tainted data.
– *Dynamic Detection Tool*: Our detection tool is implemented on dynamic instrumentation tool. During program is executing, the tool selects the suspect instruction which is tainted and uses our checking mechanisms to verify whether it is a real Integer bug.

4 Implementation Details

In this section, we give the detailed design of IntFinder, which is implemented on decompiler Boomerang-0.3 [12] and dynamic instrumentation tool PIN-2.2 [19].

4.1 Type Analysis

Type Information Extraction. As discussed in Section 3, our type analysis is implemented on decompiler. To get sufficient type information, we extend type analysis on Boomerang from the following three aspects:

– Existing integer type reconstruction of Boomerang only considers some specific arithmetic operation, such as IMUL and SAL, to determine the signedness type of operand, we additionally extract the type information at several sensitive points just as discussed in Section 2.
– Boomerang provides the sparse type information, that is to say, it does not store the type information for operand. To be convenient for type analysis, we additionally store type information for memory operand in each statement. In addition, Boomerang does not consider the backward type information propagation(in the reverse order of normal decompilation). However, for Integer bugs, we need to trace back to determine the previous undefined type. For example, in signedness error, we need to find whether current type conflicts with the previous type information.
– Existing type analysis of Boomerang only extracts the type information based on data flow analysis, and it can only extract type information within single basic block. We traverse the control flow graph, and propagate the type of operand to other control branch and eventually provide sufficient type information.

We rebuild the type information in the following steps: First, at loading time, we modify the signature of some library functions. For example, we modify the type of the memcpy's third argument from *"size_t"* to *"unsigned int"*. Second, decompiler executes its own type analysis. Third, at certain functions and statements discussed in Section 2, we extract additional type information and propagate the type information backwards within a basic block. Fourth, we traverse the control flow graph, and propagate the type information to other basic blocks. Note that we only propagate the type with sufficient information (contains both signedness and width type). When we find the type conflicting, we set the type of the operand as *"BOT"*, which is not propagated. We stop traversing the control flow when there is no updated type information.

Suspect Instruction Set. After type analysis, we traverse the intermediate statements generated by decompiler, and create the suspect instruction set. The format of suspect instruction is shown in Table 2. *Address* field is the address of suspect instruction. *bugtype* field is the type name of the bug. *opcode* field is specific to integer overflow/underflow, including arithmetic operations. *signedness type* field is the signedness type of the destination operand in the instruction. *type of left operand* field is the type of left operand, including memory, register and constant value. *type of right operand* field is the type of right operand. *size* field is the width type of the destination operand.

Table 2. Format of suspect instruction set

Address	bugtype	opcode	signedness type	type of left operand	type of right operand	size

4.2 Taint Analysis

We find that Integer bugs often exist in the applications which get input resources just like network package, configuration file, database file, user command and so on. Some typical source functions manipulate the input data, like read, fread ,recv and so on. We select these functions as the source of taint analysis, and tag the memory which holds the data from these functions, and then, we propagate the tag according to different kinds of instructions in the granularity of byte-wise. We only check those suspect instructions whose operands are tainted. Taint analysis can help us to distinguish the malicious instruction from the benign instruction which is often introduced by programmer or compile optimization, and then reduce the false positives.

4.3 Dynamic Detection Tool

Our detection tool is implemented on dynamic instrumentation tool PIN [19], it leverages the suspect instruction set produced by type analysis and apply our checking scheme to these instructions. Our checking scheme can be divided into three categories:

– *Integer Overflow/Underflow*: We check integer overflow/underflow at arithmetic operations. Followed the integer overflow detecting rule introduced by William Stallings [23], we determine integer overflow/underflow by using EFLAGS register. However, there are several exceptions that we need to re-calculate the result of the arithmetic operation. Take 8/16 bits addition for example, GCC compiler promotes the 8/16 bits operand to 32 bits register, and then do the addition operation. It will fill dirty value to the high part of the register, and finally set the *CF* or *SF* flag incorrectly. Instead, we re-calculate the addition for these two cases.
– *Signedness Error*: We check the value of the operand which has the conflicting type information, and determine whether it has a negative value.
– *Assignment Truncation*: When truncation occurs, the value of the source operand will be larger than the maximum value which destination operand can hold, then we check whether the high-order bits of source operand are not zero.

5 Evaluation

We evaluated IntFinder with several utility applications. The evaluation is performed on an Intel Pentium Dual E2180 2.00GHz machine with 2GB memory and Linux 2.6.15 kernel. Tested programs are compiled by gcc 3.4.0 and linked with glibc 2.3.2.

5.1 Suspect Instruction Set

As discussed in Section 4, IntFinder statically selected the suspect instructions, it is the first step to reduce the instructions to be checked. Table 3 shows the number of suspect instructions produced by type analysis. We can see that, in average, IntFinder statically reports about six suspect instructions per 100k. In order to evaluate the precise of type reconstruction, we compare the type information listed in suspect instruction set to original source code manually. As shown in Table 3, we find that the precise of type

Table 3. Suspect Instruction Set of IntFinder

Program	Size	Overflow/Underflow	Signedness Error	Truncation	Total	Precise
PHP-5.2.5	10.3M	330/339	84/84	234/234	648/657	98.6%
slocate-2.7	46.8K	6/6	1/1	1/1	8/8	100%
zgv-2.8	284.7K	23/24	19/19	16/16	58/59	98.3%
python-2.5.2	3.4M	95/96	21/21	61/67	177/184	96.2%
ngircd-0.8.1	329.1K	15/17	13/13	18/19	46/49	93.9%
openssh-2.2.1	150.8K	14/15	1/1	9/9	24/25	96%

Precise rate of type reconstruction is in the form of *x/y*, "x" represents the instructions with precise type, and "y" represents the total suspect instructions.

reconstruction is nearly to 100%. Note that some false positives for type reconstruction exists in IntFinder, because it is hard to distinguish the following two cases: (1) pointer and integer variable; (2) pointer and array. The former case may cause some pointer arithmetic operation being regarded as potential integer overflow/underflow. The later case may cause the imprecise type analysis for the offset of pointer, because the offset of pointer should be regarded as signed integer, which is different from the unsigned type of the array index. However, we consider that they are non-trivial problems.

5.2 Analysis of False Positives and False Negatives

We choose several real applications to verify whether integer bug can be efficiently detected. The experiment results in Table 4 and Table 5 show that IntFinder has low false negatives and false positives.

We also test the false positives of IntFinder. Note that we uses the same error-prone input as used in false negatives testing. Table 5 shows that the false positives of IntFinder is relatively low. There is no false positives in our experiments.

Table 4. Applications with Integer Bugs Tested on IntFinder

CVE#	Program	Vulnerabiltiy	Types	IntFinder
2008-1384	PHP 5.2.5	php_sprintf_appendstring() bug [6]	Integer Overflow	✓
2003-0326	slocate-2.7	parse_decode_path() bug [1]	Integer Overflow	✓
2004-1095	zgv-5.8	multiple integer overflow [2]	Integer Overflow	✓
2008-1721	python-2.5.2	zlib extension module bug [8]	Signedness Error	✓
2005-0199	ngIRCd-0.8.1	Lists_MakeMask() bug [3]	Integer Underflow	✓
2001-0144	openssh-2.2.1	detect_attack() bug [4]	Assignment Truncation	✓

Table 5. False Positives of IntFinder

CVE#	Program	Overflow/Underflow	Signedness Error	Truncation	Real Bugs
2008-1384	PHP-5.2.5	1	0	0	1
2003-0326	slocate-2.7	1	0	0	1
2004-1095	zgv-5.8	11	0	0	11
2008-1721	python-2.5.2	0	1	0	1
2005-0199	ngIRCd-0.8.1	1	0	0	1
2001-0144	openssh-2.2.1	0	0	1	1

5.3 Performance Overhead

We evaluate the performance overhead of IntFinder. Table 6 shows the slow down factors of IntFinder. Each program runs natively, under PIN without instrumentation and under IntFinder without taint analysis once respectively. We find that the average performance overhead of PIN without instrumentation is 3.7 X, while the average performance overhead of IntFinder without taint analysis is 4.4 X. Note that we also test the performance overhead of taint analysis, which is not list in Table 6. The average slow down factor of IntFinder with taint analysis is nearly 50X. We provide the interface of taint analysis for the user to choose open/close it.

Table 6. Performance Slow Down

Program	Benchmark	Native Run	Under PIN	Under IntFinder	Slow Down of PIN	Slow Down of IntFinder
PHP-5.2.5	CVE-2008-1384	0.022s	1.101s	1.432s	50.0 X	65.1 X
slocate-2.7	CVE-2003-0326	0.008s	0.296s	0.390s	37.0 X	48.7 X
zgv-5.8	CVE-2004-1095	0.101s	0.348s	0.447s	3.4 X	4.4 X
python-2.5.2	CVE-2008-1721	0.585s	2.033s	2.592s	3.5 X	4.4 X
ngIRCd-0.8.1	CVE-2005-0199	1.156s	3.945s	4.377s	3.4 X	3.8 X
Average		0.420s	1.545s	1.848s	3.7 X	4.4 X

5.4 New Bugs

IntFinder uncovered one new signedness error in slocate-2.7. This bug exists in decode_db function in main.c. In slocate, a signed integer “tot_size” is passed as the size argument to realloc at main.c :1224, but this value may be negative after shift operation at main.c: 1222, it depends on the size of database file which reads from at main.c:1232. When the size of database reaches “G” level, it will trigger the integer bug. However, we could find no evidence in mailing lists or CVS logs that the developers were specifically trying to fix this bug.

6 Discussion

6.1 Decompile Limitation

– *Missing certain functions*: In experiment, decompiler Boomerang may fail to decompile some functions, we find that these functions either have indirect jump which may be hard for decompiler to construct accurate path, or have some relationship with other functions which can not be decompiled. Fortunately, from the experiment results, we find that the accurate decompile rate is above 95%.
– *Imprecise of decompiler*: Boomerang suffers from the common limitations of decompiler: (1) Imprecise of decoding the indirect jump. (2) Imprecise differentiate the pointer from integer variable. (3) Imprecise differentiate the pointer from array.

6.2 Dynamic Detection Limitation

– *Semantic Error* IntFinder may lose Integer bugs caused by semantic error. For example, NetBSD has an Integer bug and suffers from forcing a reference counter to wrap around to 0, which may cause the referenced object to be freed even though it was still in use.

– *Logic Operation* In our current implementation, we ignore the Integer bugs associated with certain logic operation, except the SHL and SAL. We have the following consideration: (1) It is hard to distinguish the benign logic operation from the malicious one. (2) Logic operation occurs frequently in the program. If we check the logic operation, it will bring more false positives and raise performance overhead.

6.3 Dynamic Testing Input Limitation

In our current implementation, we use existing error-prone inputs published by vulnerability database just like CVE. However, this method can only detect the known integer bugs. To further leverage our tool to find more unknown vulnerability, we need to construct the input associated with the suspect instructions. To achieve the goal, there are certain feasible methods:(1) Using symbolic execution to generate inputs to trigger the suspect instructions. (2) Using some verification tools, such as dynamic testing tool [18], to construct the relationship between input and suspect points.

7 Conclusion

In this paper, we present the design, implementation, and evaluation of IntFinder, a tool for automatically detecting integer bugs. Given a binary code, IntFinder decompiles a binary, and creates the suspect instruction set. Then IntFinder dynamically inspects the instructions in the suspect set and confirms which instruction is real Integer bug with the error-prone input. Compared with other approaches, IntFinder provides more accurate and sufficient type information and reduces the instructions which will be dynamically inspected. The evaluation of IntFinder shows that it has low false positives and negatives.

Acknowledgements

This work was supported in part by grants from the Chinese National Natural Science Foundation (60773171, 90818022, and 60721002), the Chinese National 863 High-Tech Program (2007AA01Z448), the Chinese 973 Major State Basic Program (2009CB320705), and the Natural Science Foundation of Jiangsu Province(BK2007136).

References

1. Integer overflow in parse_decode_path of slocate. CVE (2003), `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0326`
2. Integer overflow in zgv-5.8. CVE (2004), `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-1095`
3. Integer underflow in ngircd before 0.8.2. CVE (2005), `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0199`
4. Ssh crc-32 compensation attack detector vulnerability. CVE (2005), `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144`
5. Cve version: 20061101. CVE (2006), `http://www.cve.mitre.org/cgi-bin/cvekey.cgi?keyword=integer`

6. Integer overflow in pdftops. CVE (2007), http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1384
7. Vulnerability type distributions in cev. CVE (2007), http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf
8. Signedness error in python-2.5.2. CVE (2008), http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1721
9. Brumley, D., cker Chiueh, T., Johnson, R., Lin, H., Song, D.: Rich: Automatically protecting against integer-based vulnerabilities. In: Proceedings of the 14th Annual Network and Distributed System Security, NDSS (2007)
10. Chen, P., Wang, Y., Xin, Z., Mao, B., Xie, L.: Brick: A binary tool for run-time detecting and locating integer-based vulnerability. In: International Conference on Availability, Reliability and Security, pp. 208–215 (2009)
11. Dipanwita, S., Muthu, J., Jay, T., Ramanathan, V.: Flow-insensitive static analysis for detecting integer anomalies in programs. In: Proceedings of the 25th conference on IASTED International Multi-Conference (SE 2007), pp. 334–340. ACTA Press, Anaheim (2007)
12. Emmerik, M.J.V.: Static Single Assignment for Decompilation. Ph.D. thesis (2007)
13. Evans, D., Guttag, J., Horning, J., Tan, Y.M.: Lclint:a tool for using specification to check code. In: Proceedings of the ACM SIGSOFT 1994 Symposium on the Foundations of Software Engineering, pp. 87–96 (1994)
14. Horovitz, O.: Big loop integer protection. Phrack Inc (2002), http://www.phrack.org/issues.html?issue=60&id=9#article
15. Howard, M.: Safe integer arithmetic in c (2006), http://blogs.msdn.com/michaelhoward/archive/2006/02/02/523392.aspx
16. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A safe dialect of c. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference (2002)
17. LeBlanc, D.: Integer handling with the c++ safeint class (2004), http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp
18. Lin, Z., Zhang, X., Xu, D.: Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In: Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN-DCCS 2008 (2008)
19. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Programming Language Design and Implementation, pp. 190–200 (2005)
20. Molnar, D., Li, X.C., Wagner, D.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proceedings of the USENIX 2009 Annual Technical Conference (2009)
21. Necula, G.C., McPeak, S., Weimer, W.: Ccured: Type-safe retrofitting of legacy code. In: Proceedings of the Principles of Programming Languages, pp. 128–139 (2002)
22. Seward, J., Nethercote, N.: Using valgrind to detect undefined value errors with bit-precision. In: Proceedings of the annual conference on USENIX Annual Technical Conference (2005)
23. Stalling, W.: Computer organization and architecture designing for performance. Prentice Hall, Inc., Englewood Cliffs (1996)
24. Wang, T., Wei, T., Lin, Z., Zou, W.: Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium, NDSS 2009 (2009)
25. Wojtczuk, R.: Uqbtng: a tool capable of automatically finding integer overflows in win32 binaries. In: 22nd Chaos Communication Congress (2005)