Branch Obfuscation Using Binary Code Side Effects*

Hong Lin[1], Xiaohua Zhang[1], Ma Yong[2], Baohui Wang[3]
[1]Network and Information Center North China Electric Power University Beijing, China
[2]College of Information Technical Science, Nankai University, Tianjin 300071, China
[3]Software College Beihang University, Beijing, China
{linh& zh_sober}@ncepu.edu.cn, mayong@mail.nankai.edu.cn, wangbh@buaa.edu.cn

Abstract - The path constraints are leaked by binary conditional jump instructions which are the binary representation of software's internal logic. Based on the problem of software's path constraints leaking, reverse engineering using path-sensitive techniques such as symbolic execution and theorem proving poses a new threat to software intellectual property protection. In order to mitigate path information leaking problem, this paper proposed a novel branch obfuscation scheme that uses binary code side effects to hide path constraints and takes advantage of remote trusted entity to protect software's control flow graph, without changing software's functionality. The experimental results show that this branch obfuscation technique could effectively protect software's path constraints against state-of-the-art reverse engineering, yet practical in terms of performance.

Index Terms - code obfuscation, binary code side effects, exception handling, symbolic execution, trusted entity

1. Introduction

At run time, binary conditional jump instructions disclose software's path information. Researchers could easily collect path constraints from software's execution trace using concolic execution technique and accurately reason about software's internal logic.

This reverse engineering method based on the software's path information leaking problem has been widely applied in security analysis, such as software testing, vulnerability discovering, malicious code analysis, protocol analysis and other fields. This method is a double-edged sword, and it is also used for nefarious purposes—such as code theft, software tampering, crack and piracy, as shown in Fig. 1, which is a significant threat to software intellectual property protection.

Our goal is to maintain security of some confidential path information that increases the complexity of reverse engineering. We present a novel branch obfuscation approach that hides the explicit path constraints using the implicit binary instruction's side effects and deploys key branch entry points on a remote trusted entity.

Our approach replaces conditional jump instructions by instructions that have side effects. If current execution environment satisfies original path constraints, the instructions' side effects would cause an exception. Otherwise, the program will execute sequentially without exception. Exceptions will be caught by underlying operation system that invokes our registered handling function to carry out appropriate control transfer implicitly. Our approach also exploits the code mobility to hide software's key branch entry points on a remote trusted entity that a full binary version of the program is not present in memory at run time.

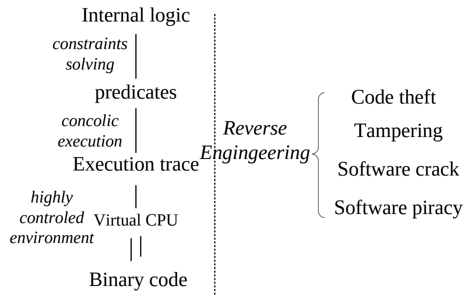


Fig. 1 Reverse engineering using path constraints

The main contribution of this paper is as follows:

1) We proposed a novel branch obfuscation approach to mitigate path information leaking. This approach hides key branch entry points on a remote trusted entity which is out of attacker's control.

2) We extended binary obfuscation approach based on exceptions. This approach not only supports unconditional control transfer instructions, such as jmp, but also supports conditional control transfer instructions, which contain software's path constraints.

3) We designed and implemented a branch obfuscator. The test result shows that branch obfuscation method is effective to protect software's path information, yet practical in terms of performance.

The remainder of this paper is organized as follows. Section 2 discusses related work of code obfuscation. Section 3 presents our proposed framework for branch obfuscation. Section 4 details the design and implementation of our obfuscator. Section 5 shows the evaluation results of branch obfuscation. Finally we conclude in section 6.

2. Related Work

In the man-at-the-end (MATE) scenario [1], adversaries have no restriction on the tools and techniques to use to reverse engineer and tamper with the software. Code obfuscation is the most viable method to fight against reverse engineering [2][3], which aims at increasing the code complexity to make it hard for adversaries to comprehend software's internal logic. Researchers have proposed many

* Supported by National Science and Technology Support Program "Key Technology Research and Demonstration for the Enhanced Search Engine", Project No. 2011BAH11B01

strategies for code obfuscation, and novel solutions are still under investigation.

A Symbolic Execution and Its Applications

In recent years, symbolic execution has advanced a lot. It is usually combined with dynamic taint analysis and theorem proving, and is becoming a powerful technique in security analysis of software programs.

Automatic testing leverages forward symbolic execution to achieve high code coverage and automatic input generation [5-11]. Most of these applications automatically generate inputs to trigger well-defined bugs [12-17], such as integer overflow, memory errors, null pointer dereference, etc. Recent work shows that symbolic execution can be used to generate succinct and accurate input signatures or filters to block exploits [18-21]. Previous work has also proposed several improvements to enhance white-box exploration on the programs that rely on string operations [22] and lift the symbolic constraints from the byte level to the protocol level [23]. Malware analysis leverages symbolic execution to capture information flows through binaries [24-27].

B. Binary Obfuscation

Barak et al. showed that some functions cannot be obfuscated [4], and other papers claimed that perfect obfuscation is impossible. However, in the practical application, binary obfuscation techniques have been recently proposed to increase reverse engineering complexity [28-30].

Many binary obfuscation approaches are proposed to fight against static analysis. Popov et al. obfuscates binary code by changing unconditional control transfers into signals (traps) and inserting dummy control transfers and "junk" instructions after the signals [31]. Sharif et al. presented conditional code obfuscation scheme to prevent symbolic execution by introducing hash function [32]. Wang et al. proposed linear obfuscation method which introduces unsolved mathematics conjectures into branch conditions to combat reverse engineering using symbolic execution and theorem proving [33].

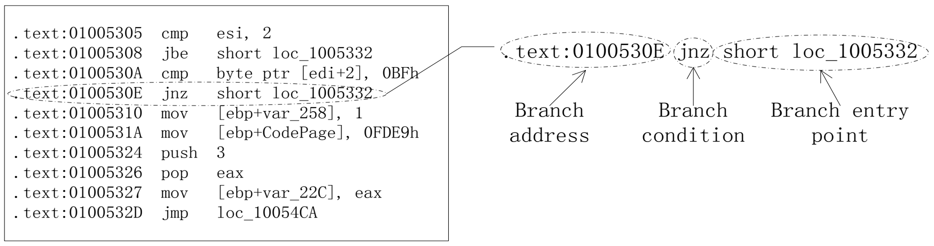
C. Obfuscation Using Code Mobility

Falcarin et al [34], Ceccato et al [35] and Wang et al. [36] proposed binary obfuscation approaches exploiting code mobility that the code in the untrusted environment is not complete. At run time, code arrives from a trusted network entity, which reduces adversaries' visibility on the whole binary code. However, the above two methods need client frequently interact large amount information with trusted server.

3. Overview of Branch Obfuscation

In the software's execution trace, the branch information is disclosed by the conditional jump instructions. Each conditional jump instruction contains 3 aspects: branch address, branch condition and branch entry point, as shown in Fig 2. Here, the branch address is the memory address of the conditional jump instructions; the branch condition is the trigger condition of control transfer; and the branch entry point is the target memory address of conditional jump when the branch condition is met.

Previous control flow obfuscation research focuses on the branch entry point confusion, such as control flow degradation, branch inversion, branch function and so on. They obfuscate branch entry points by introducing indirect jumps, because indirect jumps are difficult to be analyzed by static reverse engineering techniques. However, the current dynamic reverse engineering tools can collect and deduce software's path information accurately, which greatly weakens the strength of branch entry point obfuscation.



```
.text:01005305  cmp   esi, 2
.text:01005308  jbe   short loc_1005332
.text:0100530A  cmp   byte ptr [edi+2], 0BFh
.text:0100530E  jnz   short loc_1005332
.text:01005310  mov   [ebp+var_258], 1
.text:0100531A  mov   [ebp+CodePage], 0FDE9h
.text:01005324  push  3
.text:01005326  pop   eax
.text:01005327  mov   [ebp+var_22C], eax
.text:0100532D  jmp   loc_10054CA
```

Fig 2 The branch information in software's execution trace

As shown in Fig. 3, this paper proposes a novel control flow obfuscation strategy, which obfuscates branch address, branch condition and branch entry point.

A. Obfuscate Branch Address and Condition

Each conditional jump instruction makes the program produce 2 execution paths according to whether the branch condition is met. Instead of jump instructions, we use instruction's side effects and exception handling function to implicitly simulate software's control flow transfers at run time.

The x86 instruction set is a complex set that most instructions have side effects. Instruction side effect can implicitly change the CPU status and then implicitly affect the execution of subsequent instructions. Evaluating each instruction's side effects and reasoning about various mutual influences among different instructions are very complex. Therefore, we exploit the implicitly side-effect instructions to replace conditional jump instruction to increase the difficulty of reverse analysis and reasoning.

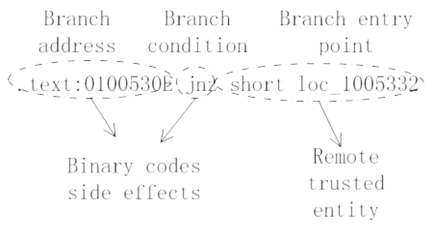


Fig 3 Branch obfuscation overview

Fig. 4 gives an example of branch obfuscation. The Fig. 4(a) is the original control flow graph, and the Fig. 4(b) is the obfuscated control flow graph that the conditional jump instructions are replaced by the side-effect instruction “div”.

B. *Obfuscate Branch Entry Point*

When an instruction raises an exception, the operation system will store the address of the instruction and invokes appropriate exception handling functions. Fig. 5 (a) shows the normal exception handling process. Note that if the exception is handled, OS returns control to the same instruction address where the exception is caught.

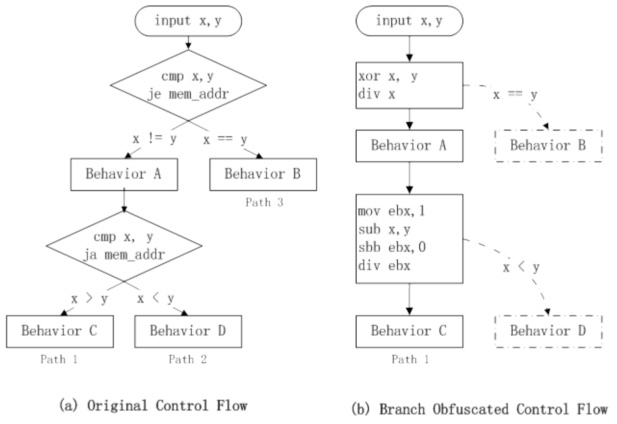


Fig 4 An example of branch obfuscation

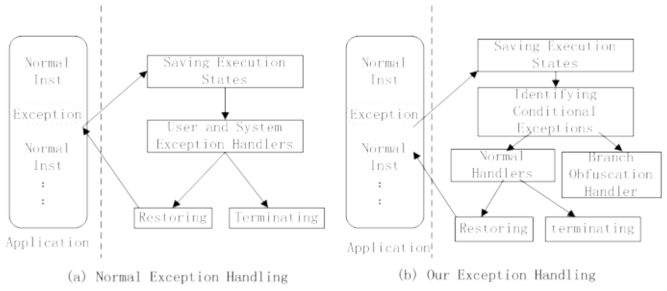


Fig 5 Control transfer using exception handling

Fig. 5(b) is our exception handling process. The essential difference is that we return control to a different address which is the branch entry point of the original conditional jump instruction. All branch entry points are stored in a mapping table that maps branch address to branch entry points. This mapping table is deployed on a remote trusted entity, which is out of the control of adversaries as shown in Fig. 6. The branch entry point is delivered from the remote trusted entity at run time, which limits adversaries’ knowledge. Below we will give the details of how branch obfuscation is implemented.

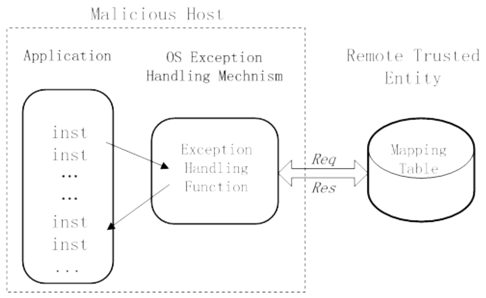


Fig 6 Mapping table in the remote trusted entity

4. Design and Implementation

Having explained the basic idea, we turn to the implementation details in this section. Conditional jump instruction’s function consists of two aspects: judge whether the jump conditions are met according to the current execution environment; transfer control flow to the specified branch entry point according to the result of the judgment. In this section we use instruction’s side effects and exception handling function to simulate conditional jump instructions.

A. *Hide Branch Condition Using Instruction’s Side Effects*

To hide original jump conditions, we should construct an equivalent one that is more difficult to understand and reverse engineer. There are two aspects to design obfuscated code: judging whether the current execution environment satisfies the jump condition; and raising exception using instruction’s side effects when jump condition is met.

Judge jump condition. Whether the conditional jump instructions will jump depends on the EFLAGS Register’s status flags. Only five of the flags can be used in this way as shown in TABLE I. The flags are used by CPU to indicate the status of current execution environment, which is only one bit (either 1 or 0).

TABLE I Status flags in the EFLAGS register

Flag	Position	Status Flags
CF	0	Carry Flag
PF	2	Parity Flag
ZF	6	Zero Flag
SF	7	Sign Flag
OF	11	Overflow Flag

Unfortunately, there is no binary instruction that is able to directly access the EFLAGS register. We found some binary instructions could indirectly access CPU status flags in the EFLAGS register as shown in Fig. 7.

Fig. 7 gives three examples of indirectly accessing the EFLAGS register to judge jump condition. Fig. 7(a) uses conditional jump instruction to check CPU status. Fig. 7(b) uses flag transfer instruction “lahf” to fetch EFLAGS register to general register. And we can also use the "pushf" and "pushfd" instructions to fetch EFLAGS register to the stack.

Another method is to use conditional transfer instruction to judge the original jump conditions as shown in Fig. 7(c). In addition, we can also directly judge the jump conditions through arithmetic instructions without accessing EFLAGS register as shown in Fig. 7 (d).

```
cmp ebx, eax
je target_addr
```

(a)

```
cmp ebx, eax
lahf
and eax, 100h
shr eax, 14
mov ebx, eax
div ebx
```

(b)

```
cmp ebx, eax
cmove ebx,
invalid_addr
mov ebx, dword
ptr [ebx]
```

(c)

```
xor ebx, eax
div ebx
```

(d)

Fig 7 Indirectly access CPU status flags

As shown in Fig. 7, jump conditions can be judged through a variety of indirect and implicit ways, which can effectively protect software's path information at run time.

Construct exception codes. After judging jump conditions, we use instruction's side effects to raise exceptions at run time when the original jump condition is met, as shown in Fig. 8. The exception trigger conditions are equivalent to the original jump conditions, and the instructions are commonly used instructions that are difficult to identify.

Fig. 8(a) is the original code snippets that path information is explicit. Fig. 8(b) is an example that uses the side effect of instruction "div" to raise exception when the jump condition in the Fig. 8(a) is met. When the jump condition is not met, the dividend value is not 0 that program will normally execute; When the original jump condition is met, the dividend value is 0 that will cause "dividing by zero" exception and transfer control to exception handling function.

```
cmp eax, ebx
ja target_addr
```

(a)

```
cmp eax, ebx
lahf
and eax, 100h
shr eax, 8
mov ebx, eax
div ebx
```

(b)

```
mov ecx, random_value
sub ebx, eax
sbb ecx, random_value-1
shl ecx, 31
mov eax, dword ptr[ecx]
```

(c)

Fig 8 Examples of exception codes

Fig. 8(c) is another example that simulates the jump condition in Fig. 8(a). The "mov" is the most widely used instruction in the binary codes. If the target memory address of instruction mov is invalid, "mov" will raise a memory access exception. In Fig. 8(c), if the original jump condition is met, the target memory address of mov is 0x00000000 that is not readable, and mov will generate an exception at run time. Otherwise, the target memory address is 0x80000000 that is valid, then the control transfers to the next instruction without exception.

It is very complicated for adversaries to reason about all potential exceptions in the binary code. In the execution trace, there are a large number of binary instructions, and the side effects of each instruction have mutual influence and interaction. So analyzing instruction's side effects to collect path information is not practical.

B. Transfer control flow implicitly

When an obfuscated instruction raises an exception, a sequence of actions occurs, and the end result is that control is transferred back to exception address. Our exception handling process is different from the traditional exception handling process that the control are transferred to specified instruction rather than back to the exception address as shown in Fig. 5. This process includes two aspects: exception handling function and mapping table on the remote trusted entity.

Exception handling Function. The main role of exception handling function is to cause the operation system to transfer control to specified address. We do this when an exception is raised from an instruction that we have inserted in the binary. However, other instructions in the original program might raise exceptions too. To tell the difference, we use a mapping table that contains all memory address of our obfuscation instructions and the corresponding branch entry points. If an exception is raised from our obfuscation instructions, we transfer the control to the jump address. If not, we transfer the control to the default exception handling functions, as shown in Fig. 5.

Mapping table. After obfuscating path information, obfuscator needs to build the table that maps exception address to branch entry points as shown in Fig. 9. Suppose that N conditional jump instructions have been obfuscated, and then there are N rows in the mapping table, one for each obfuscation instruction.

To make it hard to reverse engineer the contents of this table, we deploy the mapping table on a remote trusted entity as shown in Fig. 10. An incomplete application is deployed to the final user's computer, containing exception handler to access the mapping table. The trusted entity is a complete secure machine or device placed somewhere on the network. The exception handler can dynamically interact with the trusted entity at run time to get control transfer's target address.

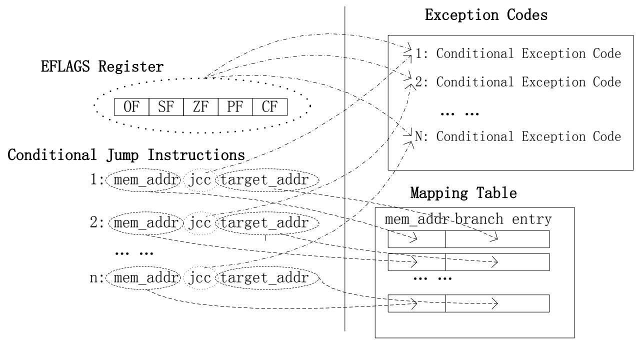

Exception Codes
1: Conditional Exception Code
2: Conditional Exception Code
... ...
N: Conditional Exception Code
EFLAGS Register
OF SF ZF PF CF
Conditional Jump Instructions
1: mem_addr jcc target_addr
2: mem_addr jcc target_addr
... ...
n: mem_addr jcc target_addr
Mapping Table
mem_addr branch entry
... ...

Fig 9 Generate exception code and mapping table

The network communication between the trusted entity and the program is created by exception handler through a network socket, includes two logical channels: a request channel used to send exception's address, and a response channel used to receive branch entry point. If the returned target address is null, our exception handler will transfer control to default exception handling functions.

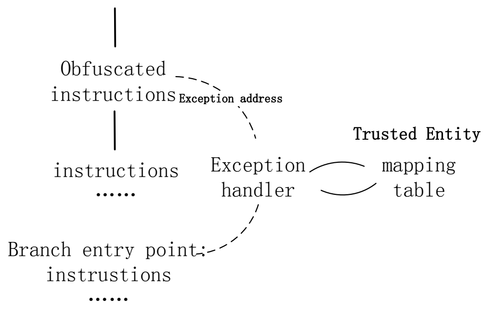


Fig 10 Execution process of branch obfuscation

As a trusted entity beyond the attacker's control, the attacker couldn't control the whole binary code, limiting the path information collection and preventing reverse engineering based on path-sensitive analysis, such as concolic execution and theorem proving.

5. Experiment

We used two code snippets to test branch obfuscation using binary code's side effects as shown in Fig. 11. Our experiments were run on an Intel Core2 Q9400 CPU with 4 GB RAM running Ubuntu 10.04. The programs were compiled with gcc version 4.3 at optimization level -O3. In the experiment, we use Crest and BitBlaze to reverse engineer the test samples, while they are using the Yices and STP as the path constraint solving tools.

Crest [1] works by inserting instrumentation code into a target program to perform symbolic execution concurrently with the concrete execution. The generated symbolic constraints are solved to generate input that drives the test execution down unexplored program paths. The analysis results are shown in TABLE II and III.

```
void time_check(int month,
                int day){
s1:  if (month > 6){
s2:    if (day > 15){
s3:      behavior_a();
       }
s4:    else{
s5:      behavior_b();
       }
     }
s6:  else{
s7:    behavior_c();
     }
   }
```

(a) Branch conditions based on date

```
void ip_filter(int i_1,int i_2,
               int i_3, int i_4){
s1:  if ((i_1 == 192){
s2:    if(i_2 == 168){
s3:      if (i_3 < 5){
s4:        valid_ip();
s5:        return;
         }
       }
     }
s6:  invalid_ip()
s7:  return;
   }
```

(b) Branch conditions based on IP

Fig 11 Branch conditions based on date and IP

[1] http://code.google.com/p/crest/

Before branch obfuscation, Crest could find 4 and 6 different execution paths from the two test samples as shown in TABLE II, using 4 different heuristic path selection methods. After branch obfuscation, Crest only found 1 execution path rather than the original 4 and 6 as shown in TABLE III. In the obfuscated codes there are no conditional jump instructions, only commonly used instructions with implicit side effects, and Crest is not able to reason about instruction's side effects. Therefore, the Crest didn't collect any path information and only found 1 execution path.

TABLE II Test result on original code using Crest

	DFS	CFG	uniform	random
date_check	4	4	4	1
ip_filter	6	6	6	1

TABLE III Test result on obfuscated code using Crest

	DFS	CFG	uniform	random
date_check	1	1	1	1
ip_filter	1	1	1	1

BitBlaze [2] is a powerful binary analysis platform that features a novel fusion of static and dynamic analysis techniques, dynamic symbolic execution, and whole-system emulation. The experimental results show that after the branch obfuscation, BitBlaze cannot collect useful path information from execution traces with the same reason as the Crest. Therefore BitBlaze is unable to effectively traverse the path space of the obfuscated codes.

As shown in TABLE IV and V, we recorded the number of executed instructions before and after path obfuscation using BitBlaze. Table data indicates that branch obfuscation using binary code's side effects only increases a small amount of execution overhead, which is practical in performance.

TABLE IV time_check path information

time_check	behavior_a	behavior_b	behavior_c
Original	9426	9425	9422
Obfuscated	9701	9565	9440

TABLE V ip_filter path information

ip_check	valid()	invalid()
Original	11631	11838
Obfuscated	12035	12129

6. Conclusion

This article proposed a branch obfuscation strategy based on the binary code side effects to hide explicit path information into implicit instruction's side effects. In addition, we use operating system exception handling mechanism to implicitly carry out control transfer and deploy key branch

[2] http://bitblaze.cs.berkeley.edu/

entry points on the remote trusted entity that is beyond adversaries' control. In the x86 complex instruction set, most instructions have implicit side effects, and side effects of different instructions have mutual influence and interaction, so analyzing instruction's side effects to collect path information is very complex and not practical. The experimental results show that this obfuscation strategy effectively mitigates software path information leaking problem and only adds a small amount of execution cost.

References

[1]. P. Falcarin, C. Collberg, M. Jakubowski, "Guest Editors' Introduction: Software Protection," IEEE Software, 2011, 28: 24-27

[2]. C. Collberg, C. Thomborson, D. Low, J. Newsome, D. Song, H. Yin "A Taxonomy of Obfuscation Transformations," Department of Computer Science, The University of Auckland, Technical Report 148, 1997

[3]. C. Collberg, C. Thomborson. "Watermarking, tamper-proofing, and obfuscation – tools for software protection," IEEE Transaction on Software Engineering, 2002, 28(8): 735-746

[4]. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang. "On the (Im) possibility of Obfuscating Programs," Advances in Cryptology (CRYPTO 01), LNCS 2139, Springer, 2001. 1-18

[5]. P. Godefroid, M. Y. Levin, D. Molnar. "Automated whitebox fuzz testing," Proceedings of the Network and Distributed System Security Symposium. Rosten, VA: Interne t Society, 2008. 1-11

[6]. C. Cadar, D. Engler. "Execution generated test cases: How to make systems code crash itself," Proceedings of Int SPIN Workshop. Berlin: Springer, 2005. 2-23

[7]. C. Cadar, D. Dunbar, D. Engler. "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," Proceedings of USENIX OSDI'08. Berkeley, CA: USENIX, 2008.

[8]. G. Lee, J. Morris, K. Parker, G. A. Bundell, P. Lam "Using symbolic execution to guide test generation," Software Testing, Verification & Reliability, 2005,15(1):41-61

[9]. C. Cadar, V. Ganesh, P. Pawlowski, D. L. Dill, D. R. "EXE: automatically generating inputs of death," Proceedings of the 2006 ACM Conference on Computer and Communications Security (CCS).

[10]. P. Godefroid, N. Klarlund, K. Sen. "DART: Directed automated random testing," Proceedings of the ACM Conference on Programming Language Design and Implementation, 2005.

[11]. K. Sen, D. Marinov, G. Agha. "CUTE: A concolic unit testing engine for c," Proceedings of the 13th International Symposium on the Foundations of Software Engineering, 2005.

[12]. D. Brumley, P. Poosankam, D. Song, J. Zheng. "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," Proceedings of IEEE Symposium on Security and Privacy. New York, NY: IEEE,2008. 143-157

[13]. D. Molnar, X. C. Li, D. A. Wagner. "Dynamic test generation to find integer bugs in x86 binary linux programs," Proceedings of USENIX Security Symposium. Berkeley, CA: USENIX, 2009. 67-82

[14]. V. Felmetsger, L. Cavedon, C. Kruegel, G. Vigna. "Toward Automated Detection of Logic Vulnerabilities in Web Applications," Proceedings of the 19thUSENIX Security Symposium. Berkeley, CA: USENIX,2010.143-160

[15]. T. Wang, T. Wei, Z. Lin, W. Zou. "IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution," Proceedings of the 16th Annual Network and Distributed System Security Symposium. Rosten VA: Internet Society, 2009.

[16]. D. Brumley, J. Newsome, D. Song, H. Wang, S. Jha. "Towards automatic generation of vulnerability-based signatures," Proceedings of 2006 IEEE Symposium Security and Privacy. Piscataway, NJ:IEEE, 2006. 2-16

[17]. C. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, D. Song. "MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery," Proceedings of the 20th USENIX Security Symposium. Berkeley, CA: USENIX, 2011.

[18]. D. Brumley, C. Hartwig, Z. Liang. Automatically identifying trigger-based behavior in malware. Book Chapter in "Botnet Analysis in Defense" 36 (2007)

[19]. D. Brumley, H. Wang, S. Jha, D. Song. "Creating Vulnerability Signatures Using Weakest Preconditions," Proceedings of the 20th IEEE Computer Security Foundations Symposium. Piscataway, NJ: IEEE, 2007. 311-325

[20]. D. Brumley, C. Hartwig, M. Kang, Z. Liang, J. Newsome, P. Poosankam, et al. "BitScope: Automatically Dissecting Malicious Binaries," School of Computer Science, Carnegie Mellon University, Technical Report CS-07-133, 2007

[21]. M. Costa, M. Castro, L. Zhou, L. Zhang, M. Peinado. "Bouncer: Securing software by blocking bad input," Proceedings of the 2007 ACM Symposium on Operating Systems Principles (SOSP) , 2007.

[22]. J. Caballero, S. McCamant, A. Barth, D. Song. "Extracting models of security sensitive operations using string-enhanced white-box exploration on binaries," Technical Report UCB/EECS-2009-36, EECS Department, University of California, Berkeley, 2009.

[23]. J. Caballero, Z. Liang, P. Poosankam, D. Song. "Towards Generating High Coverage Vulnerability-Based Signatures with Protocol-Level Constraint-Guided Exploration," Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection. Berlin: Springer, 2009. 161-181

[24]. P. M. Comparetti，G. Salvaneschi，E. Kirda, C. Kolbitsch, C. Kruegel, S. Zanero. "Identifying Dormant Functionality in Malware Programs," Proceedings of the 2010 IEEE Symposium on Security and Privacy. Piscataway, NJ: IEEE, 2010. 61-76

[25]. A. Moser, C. Kruegel, E.Kirda. "Exploring Multiple Execution Paths for Malware Analysis," Proceedings of IEEE Symposium on Security and Privacy. Piscataway, NJ: IEEE, 2007 231-245

[26]. H. Yin, D. Song, M. Egele, C. Kruegel, E. Kirda. "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," Proceedings of ACM Conference on Computer and Communication Security. New York, NY: ACM, 2007. 116-127

[27]. D. Brumley, C. Hartwig, Z. Liang. "Automatically Identifying Trigger-based Behavior in Malware," In: Lee W, Wang C, Dagon D, eds. Botnet Detection. Berlin: Springer, 2008. 65-88

[28]. B. Lynn, M. Prabhakaran, A. Sahai. "Positive Results and Techniques for Obfuscation," Advances in Cryptology-EUROCRYPT, Lecture Notes in Computer Science(LNCS), 2004，3027:20-39

[29]. N. Kuzurin, A. Shokurov, N. Varnovsky, V. Zakharov. "On the Concept of Software Obfuscation in Computer Security," Lecture Notes in Computer Science(LNCS), 2007, 4779: 281-298

[30]. P. Beaucamps, E. Filiol. "On the possibility of practically obfuscating programs towards a unified perspective of code protection," Journal in Computer Virology, 2007, 3(1): 3-21

[31]. I. Popov , S. Debray, G. Andrews. "Binary obfuscation using signals," Proceedings of the USENIX Security Symposium. Berkeley, CA: USENIX, 2007. 321-333

[32]. M. Sharif, A. Lanzi, J. Giffin, W. Lee. "Impeding malware analysis using conditional code obfuscation," Proceedings of the Network and Distributed System Security Symposium. Rosten,VA: Internet Society, 2008. 321-333

[33]. Z. Wang, J. Ming, C. Jia, D. Gao . "Linear Obfuscation to Combat Symbolic Execution," Proceedings of the European Symposium on Research in Computer Security (ESORICS2011), Springer, 2011, 210-226

[34]. P. Falcarin, S. Carlo, A. Cabutto, N. Garazzino, D. Barberis. "Exploiting Code Mobility for Dynamic Binary Obfuscation," Proceedings of the 2011 World Congress on Internet Security, Piscataway, NJ: IEEE, 2011. 114-120

[35]. M. Ceccato, P. Tonella. "CodeBender: Remote Software Protection Using Orthogonal Replacement," IEEE Software, 2011, 28(2): 28-34

[36]. Z. Wang, C. Jia, M. Liu, X Yu. "Branch Obfuscation Using Code Mobility and Signal," Proceedings of the 7th IEEE International Workshop on Security, Trust, and Privacy for Software Applications (STPSA'12), Izmir, Turkey, 2012