CN 43-1258/TP
ISSN 1007-130X

计算机工程与科学
Computer Engineering & Science

第 38 卷第 12 期 2016 年 12 月
Vol. 38, No. 12, Dec. 2016

文章编号:1007-130X(2016)12-2536-06

结合静态分析与动态符号执行的软件漏洞检测方法*

蔡　军,邹　鹏,熊达鹏,何　骏

(装备学院复杂电子系统仿真实验室,北京 101416)

摘　要:动态符号执行是近年来新兴的一种软件漏洞检测方法,它可以为目标程序的不同执行路径自动生成测试用例,从而获得较高的测试代码覆盖率。然而,程序的执行路径很多,且大部分路径都是漏洞无关的,通常那些包含危险函数调用的路径更有可能通向漏洞。提出一种基于静态分析的有导动态符号执行方法,并实现了一个工具原型 SAGDSE。该方法通过静态分析识别目标程序中调用危险函数的指令地址,在动态符号执行过程中遇到这些指令地址时收集危险路径约束,再通过约束求解生成走危险路径的测试用例,这些测试用例将更可能触发程序漏洞。实验结果表明了该方法的有效性。

关键词:软件漏洞检测;静态分析;动态符号执行;危险路径

中图分类号:TP309　　**文献标志码**:A

doi:10.3969/j.issn.1007-130X.2016.12.021

A software vulnerability detection method based on static analysis and dynamic symbolic execution

CAI Jun, ZOU Peng, XIONG Da-peng, HE Jun

(Science and Technology on Complex Electronic System Simulation Laboratory, Academy of Equipment, Beijing 101416, China)

Abstract: Dynamic symbolic execution is a software vulnerability detection method emerging in recent years, which can automatically generate test cases for different execution paths of the target program, so it can obtain high test code coverage. However, there are so many execution paths of a program, and most of them are unrelated to vulnerabilities, and those paths containing dangerous function calls are more likely to lead to vulnerabilities. We propose a guided dynamic symbolic execution method based on static analysis, and implement a tool prototype named SAGDSE. This method firstly identifies the program instructions that call dangerous functions via static analysis, and then collects the constraints of dangerous paths during the dynamic symbolic execution process when encountering these instructions. Finally it generates test cases that go through these dangerous paths by solving the constraints. These test cases are more likely to trigger program vulnerabilities. Experimental results verify the effectiveness of the proposed method.

Key words: software vulnerability detection; static analysis; dynamic symbolic execution; dangerous path

1　引言

当今时代是信息时代,随着全球信息化的迅猛发展,信息系统已经成为世界各国经济、军事和社会发展的重要基础设施。与此同时,信息窃取、资源被控、系统崩溃等各类信息安全事件层出不穷,给国民经济、国家安全和社会稳定带来严重威胁。信息化越深入,信息安全的重要性就越突出。然而,造成这些信息安全事件的一个根本原因就是信

* **收稿日期**:2015-07-09;**修回日期**:2015-11-13
基金项目:国家 863 计划(2012AA012902);"核高基"国家科技重大专项基金(2013ZX01045-004)
通信地址:101416 北京市怀柔区 3380 信箱 93 号
Address: Mailbox 3380-93, Huairou District, Beijing 101416, P. R. China

息系统或软件自身存在可被利用的漏洞[1]。因此,软件漏洞检测日益成为信息安全领域的一个研究热点。

动态符号执行是近年来一种新兴的软件漏洞检测方法。其核心思想是将具体执行与符号执行相结合,在程序真实运行过程中,判断哪些代码需要经过符号执行,哪些代码可以直接运行[2]。使用动态符号执行检测软件漏洞的一般过程如图1所示:给定一个种子输入(也叫初始输入),交给目标程序执行,在程序读取输入数据时,将输入符号化,然后边执行程序边收集路径约束(遇到条件跳转时),程序执行完后得到一个路径约束集合,将约束集合中的约束逐个取反,交由约束求解器求解生成新的输入(即测试用例),再以这些测试用例作为种子输入,重复上述过程,生成新的测试用例,如此迭代往复;在程序执行每一个测试用例的同时监控其执行状态,如果有异常发生(例如程序崩溃),就表示程序可能存在漏洞,通过进一步对异常原因进行分析,确认是否为漏洞及漏洞的类型。

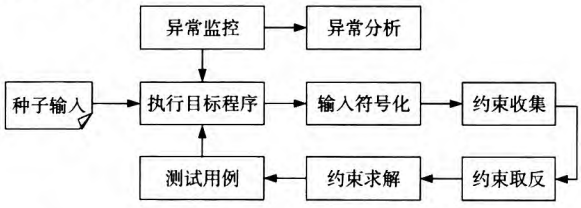

Figure 1　General process of detecting software vulnerabilities using dynamic symbolic execution

图1　使用动态符号执行检测软件漏洞的一般过程

与其它漏洞检测方法如模糊测试、静态符号执行相比,动态符号执行具有测试覆盖率高、分析结果准确等优点,也存在难以克服路径爆炸问题的缺陷。如前文所述,动态符号执行是一个迭代的过程,每生成一个测试用例将驱动目标程序走一条不同的路径;理论上,使用动态符号执行方法,可以驱动目标程序走完所有可能的执行路径。然而,当软件大小达到一定规模时,路径数量将相当庞大,加之约束求解器求解能力的限制,即便不考虑时空开销,也几乎不可能走完所有路径。事实上,大部分路径都是漏洞无关的,如果能直接找到可能通向程序漏洞的路径,将大大提高检测效率。

基于上述考虑,本文提出了一种基于静态分析的有导动态符号执行方法,并实现了一个工具原型SAGDSE(Guided Dynamic Symbolic Execution based on Static Analysis)。该方法的核心思想如下:(1)编写Python脚本,通过IDA Pro的IDA-Python插件定制静态分析流程,识别目标程序包含哪些危险函数以及调用危险函数的指令地址;(2)根据静态分析得到的指令地址列表通过动态符号执行搜索包含危险函数调用的危险路径。

2　相关工作

动态符号执行[3,4]是一种动态漏洞检测方法。目前主流的动态漏洞检测方法包括模糊测试、动态符号执行和动态污点分析三种方法。模糊测试[5-7]的基本思想是构造大量测试用例,交给程序执行,如果程序崩溃或是挂起,就有可能是一个潜在的漏洞。它通过两种方法来构造测试用例:一是基于变异,即对已有数据样本应用变异技术来创建测试用例;二是基于生成,即通过对目标文件格式或协议建模来从头创建测试用例。模糊测试的缺点是随机性较强,构造的测试用例可能会走单一路径,代码覆盖率低。动态污点分析方法[8-10]将用户输入作为污点源,追踪污点源在目标程序执行过程中的传播,如果有安全敏感操作在操作污点数据,就有可能存在漏洞,该方法的缺点是不能自主构造测试用例。动态符号执行面向程序路径来构造测试用例,与前两种方法相比,在获得高代码覆盖率方面具有明显优势。

符号执行用抽象符号代替程序变量,根据程序的语义,在每条路径上通过符号计算引擎对抽象符号做等语义操作,模拟程序执行[2]。早期的符号执行都是静态符号执行,需要程序源码,由于缺乏程序运行时信息,静态符号执行在模拟程序执行时遇到了很多困难,且容易产生误报。动态符号执行是在静态符号执行的基础上发展而来的,从2005年最早出现至今,已经涌现出了一些优秀的动态符号执行工具,如DART[11]、KLEE[12]、Fuzzgrind[13]和Fuzzball[14]等,但是由于符号执行的最大瓶颈即路径爆炸问题始终未能得到很好的解决,这些工具的适用范围也比较有限。本文的研究不是为了解决路径爆炸问题,而是试图通过将动态符号执行技术与其它技术结合来检测漏洞,充分发挥动态符号执行在路径搜索方面的优势。

3　有导动态符号执行

3.1　概述

软件漏洞种类很多,不同种类的漏洞,其产生的机理也不同,相应的漏洞检测方法也不同。没有

哪一种漏洞检测方法能够适用于所有类型的漏洞检测,通常每种漏洞检测方法只能检测有限的几种类型的漏洞。动态符号执行主要适用于缓冲区溢出、整数溢出、除0等输入相关类漏洞(即可以由某个特别构造的输入触发的漏洞)的检测。

要检测某一类漏洞,通常先提取该类漏洞的漏洞模式。漏洞模式是研究人员在通过对大量已知漏洞的产生原理进行深入分析,并在归纳总结出其一般规律的基础上,抽象出的存在安全缺陷的代码段在二进制代码或者汇编代码表现形式上具有的典型特征[1]。例如,缓冲区溢出类漏洞模式的典型特征就是常常与不安全函数调用有关,不安全函数主要包括C函数中的一些没有判断输入长度的内存和字符串操作函数(*strcpy*、*strcat*、*sprintf* 等);整数溢出类漏洞模式则常常与内存操作函数(*malloc*、*calloc*、*realloc* 等)有关。因此,对于缓冲区溢出和整数溢出类漏洞的检测,只需重点关注那些包含不安全函数和内存操作函数调用的路径。本文统称这些安全相关函数为危险函数,称包含危险函数调用的程序路径为危险路径。

本文所提有导动态符号执行的目的就是要通过动态符号执行搜索目标程序的危险路径。找到危险路径后,可以重点对这些危险路径进行测试,而对于危险路径之外的其它路径则不用再做进一步的分析测试,由于缩小了需要测试的路径范围,进而提高了漏洞检测效率。有导动态符号执行的工作流程如图2所示。首先确定要检测什么类型的漏洞,建立相应的漏洞模式,确定哪些函数将有可能触发这些漏洞;接下来通过静态分析,识别目标程序中调用危险函数的指令地址;再接下来开始动态符号执行,在遇到调用危险函数的指令时,输出当前已经收集到的路径约束集合,这个约束集合即为包含该危险函数调用的路径的约束;最后当动态符号执行停止时,输出所有危险函数对应的危险路径约束。通过求解危险路径约束,即可得到相应的测试用例。需要说明的是:一条路径对应很多测试用例,一次约束求解只能得到一个测试用例,可通过多次求解得到走同一条危险路径的不同测试用例。

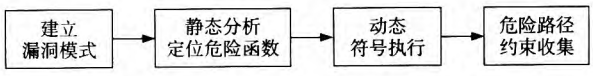

Figure 2 Workflow of guided dynamic symbolic execution

图2 有导动态符号执行工作流程

本文工具原型SAGDSE是在开源软件Fuzzgrind[13]的基础上开发的。Fuzzgrind是一个典型的动态符号执行工具,它从给定目标程序一个正常的输入文件开始执行,分析程序执行过程中遇到的分支指令,收集路径约束,通过逐个求解约束,修改正常文件的部分字节生成新的文件来驱动目标程序走新的路径。Fuzzgrind的缺点是只能运行在版本比较陈旧的Ubuntu 9.04系统上,且依赖于较低版本的Valgrind(一个开源的二进制插桩平台)[15]。SAGDSE借用了Fuzzgrind的基本动态符号执行功能,将运行平台移植到了Ubuntu 12.04上,并匹配了较新版本的Valgrind,再结合静态分析来搜索危险路径。

3.2 二进制程序静态分析

静态分析指在不运行程序的前提下,对程序的源码或是二进制代码进行的分析。由于在实际应用中大多数软件均以二进制代码形式存在,研究针对二进制代码的漏洞检测技术具有更强的实用价值。通过静态分析,可以对程序内部结构有一个大致的了解,还可以找到可能存在漏洞的位置,便于我们在用动态分析方法检测漏洞时做到有的放矢,提高效率。静态分析的最大缺点是误报率高,即通过静态分析分析出来的漏洞在实际运行时不一定真实存在,需要经过动态方法来确认,本文方法实质是一种动静结合的漏洞检测方法。

SAGDSE使用IDA Pro来实施静态分析。IDA Pro是一款强大的二进制代码静态分析工具,它提供了丰富的静态分析手段,能够识别各种不同体系架构下的二进制文件格式,支持多种平台。IDA Pro采用的是开放式架构,允许用户通过IDA自带的IDC脚本语言开发自己的静态分析工具。2004年,Erdelyi和Carrera[16]发布了IDAPython,这是一款IDA Pro插件,提供了IDC脚本引擎的Python接口,通过这款插件,研究者只需编写纯粹的Python脚本,就可以在IDA下定制自动化静态分析流程。SAGDSE主要使用IDA Pro来识别目标程序中有哪些危险函数,并识别调用这些危险函数的指令地址。

具体的算法流程如图3所示:首先,根据漏洞模式初始化一个危险函数列表 *Danger_Funcs*,即确定要重点关注哪些函数;接着从列表中逐个取出待分析的危险函数,当列表为空时分析完毕;对于每一个危险函数 *func*,调用IDAPython提供的函数 *LockByName* (string *FunctionName*) 获取 *func* 的调用地址,如果返回的是合法地址,则再调用 *CodeRefsTo* (long *Address*,*bool Flow*)函数获取指向 *func* 调用地址的代码应用列表,也就是调用 *func* 的指令地址列表;最后打印这些指令的地

址,并将指令标红。

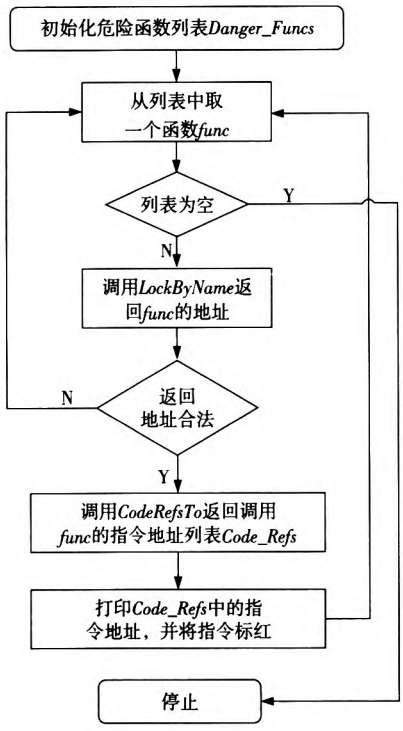

Figure 3　Flow chart of the static analysis algorithm

图 3　静态分析算法流程图

图 4 是对 png2swf 0.9.2 在 IDA Pro 中执行本文的静态分析算法后得到的分析结果(该图只显示了部分结果)。图 4 中标记为灰色的指令就是调用危险函数的指令,可以看到一共显示了两条指令"*call _malloc*"和"*call _strcpy*";IDA Pro 输出窗口(Output Window)中黑色方框里的内容就是调用危险函数的指令地址,可以看到调用 *strcpy* 函数的指令有 5 条,地址分别为"0x0804b007、0x0804b6e3、0x0804b71a、0x0805e9c1 和 0x0805ea68";调用 *malloc* 函数的指令有 4 条。

3.3　静态分析指导下的动态符号执行

通过上一步的静态分析,定位了目标程序中的危险函数,接下来就要使用动态符号执行方法搜索包含这些危险函数调用的危险路径。例如,对于图 4 中的地址为"0x0804b6e3"的调用 *strcpy* 函数的指令"*call _strcpy*",需要找到程序在执行这条指令之前经过了哪些路径分支节点,相应的这些分支节点的约束条件构成的约束公式即为这条指令的路径约束。

具体的危险路径搜索算法流程如图 5 所示,输入为通过静态分析得到的危险函数调用表 *DFunc_Map*,路径约束集合 *PC* 和危险路径约束表 *DCons_Map* 初始为空,*DCons_Map* 中的每个元素为一个二元组〈*dfunc*,*pc*〉,其中 *dfunc* 为危险函数,*pc* 为该危险函数依赖的路径约束;开始动态

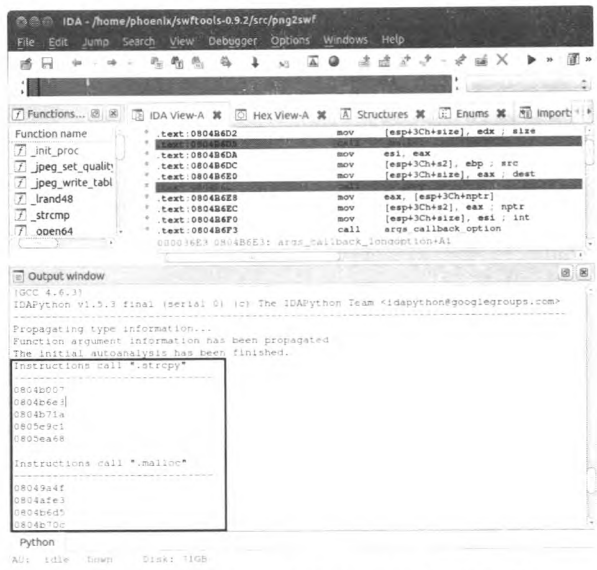

Figure 4　Static analysis results of png2swf

图 4　png2swf 的静态分析结果

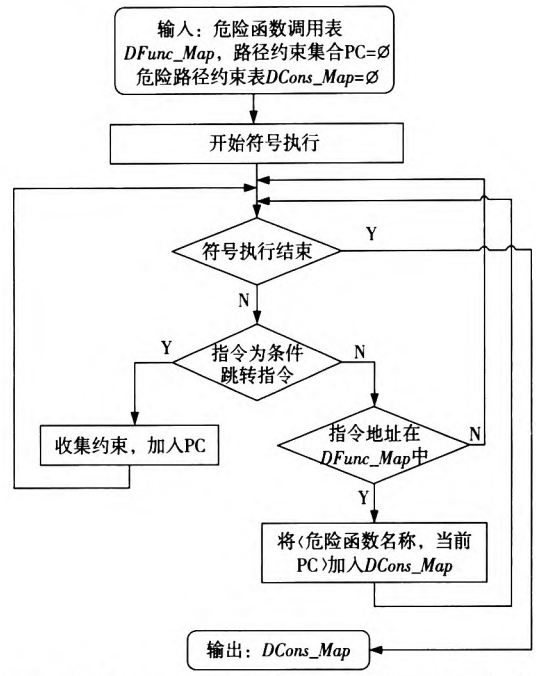

Figure 5　Flow char of the dangerous path search algorithm

图 5　危险路径搜索算法流程图

符号执行后,首先将输入符号化,即为输入的每一个字节指派一个符号,然后逐条分析正在执行的程序指令,记录指令对符号的操作,如果指令为条件跳转指令,且跳转条件与符号相关,则将符号约束加入到 PC 中;如果指令地址在 *DFunc_Map* 中,则表示此处调用了危险函数,则将〈危险函数名称,当前 PC〉加入到 *DFunc_Map* 中。最后当动态符号执行停止时,输出 *DFunc_Map*。

4　实验分析

本节对本文方法进行实验验证。实验环境为 Ubuntu 12.04 32 位操作系统。实验分为两个部

分,首先用一个简单程序对SAGDSE进行测试,看其能否正确找到危险路径,然后使用真实应用程序来进行测试。

4.1 简单程序测试

测试程序为ELF格式的二进制程序test_strcpy,其部分源代码如下所示:

```
void function(char *str){
  char buffer[6];
  strcpy(buffer, str);
}
int main(int argc, char *argv[]) {
  char buf[5] = { 0 };
  int fd, i, num = 0;
  fd = open(argv[1], O_RDONLY);
  for (i = 0; i<=3; i++)
    read(fd, &buf[i], sizeof(char));
  if (buf[0] == 'g') num++;
  if (buf[1] == 'o') num++;
  if (buf[2] == 'o') num++;
  if (buf[3] == 'd') num++;
  if(num == 2) return 2;
  if (num == 4) {function(buf);return 4;}
  return 0;
}
```

该程序从输入文件读取4个字符,看其是否分别与"good"的4个字符相同,根据字符相同的个数得到不同的结果,当4个字符都相同时,将调用危险函数strcpy。

给定初始输入文件的内容为"aaaa",使用SAGDSE对test_strcpy进行测试,一共生成了15个测试用例,其中只有第14个测试用例对应危险路径,具体执行结果如表1所示,表中的'NUL'代表空字符。

Table 1 Test results of *test_strcpy*

表1 *test_strcpy*测试结果

测试用例编号	1	2	3	4	5
内容	gaaa	'NUL'oaa	'NUL' 'NUL' oa	'NUL' 'NUL' 'NUL'd	'NUL' 'NUL'od
测试用例编号	6	7	8	9	10
内容	'NUL' ooa	'NUL'o 'NUL'd	goaa	g 'NUL' oa	g'NUL' 'NUL'd
测试用例编号	11	12	13	14	15
内容	g'NUL' od	gooa	go 'NUL'd	good	'NUL' ood

下面分析这个结果的准确性,主要从两个方面进行分析:一是总的路径条数是否应该为15,二是第14号测试用例是否会走危险路径。对于第一个方面,分析源码可以得知"return 2"的路径条数应为$C_4^2=6$(即有两个字符相同的情况),"return 4"的路径条数应为$C_4^4=1$,"return 0"的路径条数应为$C_4^0+C_4^1+C_4^3=8$,故总的路径条数应为6+1+8=15,与实际检测到的相符;对于第二个方面,由于第14号测试用例的内容为"good",满足"num==4",因而将执行到*strcpy*函数,是危险路径。收集到的危险路径约束如图6所示,可以看到当前PC中一共有4个约束,分别与符号"*input*(0)"、"*input*(1)"、"*input*(2)"和"*input*(3)"(分别对应输入文件的第1、2、3、4个字节)相关,且每个约束的分支均为True分支。

Figure 6 Flow chart of the dangerous path search algorithm

图6 危险路径搜索算法流程图

4.2 真实应用程序测试

使用三款真实应用程序对SAGDSE进行了测试,测试结果如表2所示。以swfbytes为例进行说明,种子文件为一个SWF文件,大小为3 551 B,最终找到5条危险路径,危险路径约束的输出结果保存在文本文件"Dcons.txt"中,部分危险路径约束如图7所示。可以看到真实应用程序的危险路径约束比简单应用程序明显复杂。另外,在实验中还发现,静态分析得到的危险函数调用并不都对应危险路径,因为它们可能与输入无关。

Table 2 Test results of real applications

表2 真实应用程序测试结果

测试软件名称版本	输入文件格式	输入文件大小	软件可执行文件大小	找到的危险路径数量
jpeg2swf 0.9.2	jpg	412 B	156.4 KB	1
png2swf 0.9.2	png	108 B	145.4 KB	3
swfbytes 0.9.2	swf	3.6 KB	87.6 KB	5

综上,SAGDSE能够准确搜索程序路径,并识

Figure 7　Part of the dangerous path constraints of swfbytes

图 7　swfbytes 的部分危险路径约束

别危险路径,输出相应的危险路径约束。收集到危险路径约束后,可随时通过约束求解生成新的测试用例,为进一步的漏洞检测奠定基础,如可以进一步对危险路径进行模糊测试,直至发现漏洞。

5　结束语

本文提出了一种基于静态分析的有导动态符号执行方法,并实现了一个工具原型 SAGDSE。该方法不依赖于程序源代码,通过静态分析识别目标程序的内部信息,根据这些内部信息,通过动态符号执行来搜索危险路径。由于危险路径只占程序路径的很少一部分,如果只是对危险路径进行重点测试,相比全路径测试效率更高。实验表明,SAGDSE 能够准确搜索到程序的危险路径,并输出相应的危险路径约束。SAGDSE 的不足在于危险路径查找算法的效率还有待提高。

参考文献:

[1] Wu Shi-zhong, Guo Tao, Dong Guo-wei, et al. Software vulnerability analysis techniques[M]. Beijing: Science, 2014. (in Chinese)

[2] Wang Tie-lei. Research on binary-executable-oriented software vulnerability detection[D]. Beijing: Peking University, 2011. (in Chinese)

[3] Cadar C, Godefroid P, Khurshid S, et al. Symbolic execution for software testing in practice: Preliminary assessment[C]// Proc of the 33rd International Conference on Software Engineering, 2011: 1066-1071.

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

[5] Sutton M, Greene A, Amini P. Fuzzing: Brute force vulnerability discovery[M]. Upper Saddle River: Addison-Wesley Professional Press, 2007.

[6] Takanen A. Fuzzing: The past, the present and the future[C]// Actes du 7ème Symposium Sur la Sécurité des Technologies de l'information et des Communications (SSTIC), 2009: 202-212.

[7] Pak B S. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution[D]. Pittsburgh: Carnegie Mellon University, 2012.

[8] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software[EB/OL]. [2015-07-09]. http://repository.cmu.edu/cgi/viewcontent.cgi?article=1042&context=ece.

[9] Kang M G, McCamant S, Poosankam P, et al. DTA++: Dynamic taint analysis with targeted control-flow propagation [C]// Proc of the 18th Annual Network & Distributed System Security Symposium (NDSS), 2011: 1.

[10] Kemerlis V P, Portokalidis G, Jee K, et al. libdft: Practical dynamic data flow tracking for commodity systems[J]. Acm Sigplan Notices, 2012, 47(7): 121-132.

[11] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing[J]. Acm Sigplan Notices, 2005, 40(6): 213-223.

[12] Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs[C]// Proc of the 8th USENIX Symposium on Operating Systems Design and Implementation, 2008: 209-224.

[13] Sogeti ESEC Lab. Fuzzgrind[EB/OL]. [2015-07-09]. http://esec-lab.sogeti.com/pages/fuzzgrind.html.

[14] Martignoni L, McCamant S, Poosankam P, et al. Path-exploration lifting: Hi-fi tests for lo-fi emulators[J]. Computer Architecture News, 2012, 40(1): 337-348.

[15] Armour-Brown C, Borntraeger C, Fitzhardinge J, et al. Valgrind[EB/OL]. [2015-07-09]. http://valgrind.org/.

[16] Erdelyi G. IDAPython[EB/OL]. [2015-07-09]. http://code.google.com/p/idapython/.

附中文参考文献:

[1] 吴世忠,郭涛,董国伟,等. 软件漏洞分析技术[M]. 北京:科学出版社,2014.

[2] 王铁磊. 面向二进制程序的漏洞挖掘关键技术研究[D]. 北京:北京大学,2011.

作者简介:

蔡军(1982-),男,湖北天门人,博士生,CCF 会员(E200040476G),研究方向为网络与信息安全。**E-mail**: cjgfkd@163.com

CAI Jun, born in 1982, PhD candidate, CCF member(E200040476G), his research interest includes network, and information security.