Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2016-CJ-001

2016-CJ-001

基于目标制导符号执行的静态缓冲区溢出警报自动确认技术

鲍铁匀，高凤娟，周严，李游，王林章，李宣东

信息安全学报 2016 年 4 月第 1 卷第 2 期

基于目标制导符号执行的静态缓冲区溢出警报自动确认技术

鲍铁匀[1,2,3], 高凤娟[1,2,3], 周 严[1,2,3], 李 游[1,2,3], 王林章[1,2,3], 李宣东[1,2,3]

[1]计算机软件新技术国家重点实验室(南京大学) 南京 中国 210023
[2]江苏省软件新技术与产业化协同创新中心 南京 中国 210023
[3]南京大学计算机科学与技术系 南京 中国 210023

摘要 缓冲区溢出漏洞是一类严重的安全性缺陷。目前存在动态测试和静态分析技术来检测缓冲区溢出缺陷: 动态测试技术的有效性取决于测试用例的设计, 而且往往会引入执行开销; 静态分析技术及自动化工具已经被广泛运用于缓冲区溢出缺陷检测中, 然而静态分析由于采取了保守的策略, 其结果往往包含数量巨大的误报, 需要通过进一步人工确认来甄别误报, 但人工确认静态分析的结果耗时且容易出错, 严重限制了静态分析技术的实用性。符号执行技术使用符号代替实际输入, 能系统地探索程序的状态空间并生成高覆盖度的测试用例。本文提出一种基于目标制导符号执行的静态缓冲区溢出警报确认方法, 使用静态分析工具的输出结果作为目标, 制导符号执行确认警报。我们的方法分为 3 步: 首先在过程间控制流图中检测静态分析警报路径片段的可达性, 并将可达的警报路径片段集合映射为用于确认的完整确认路径集合; 其次在符号执行中通过修剪与溢出缺陷疑似语句无关的路径, 指导符号执行沿特定确认路径执行; 最后在溢出缺陷疑似语句收集路径约束并加入溢出条件, 通过约束求解的结果, 对静态分析的警报进行分类。基于上述方法我们实现了原型工具 BOVTool, 实验结果表明在实际开源程序上 BOVTool 能够代替人工减少检查 59.9%的缓冲区溢出误报。

关键词 符号执行; 缓冲区溢出; 警报确认; 目标制导
中图法分类号 TP309.1 DOI 号 10.19363/j.cnki.cn10-1380/tn.2016.02.005

Automatically Validating Static Buffer Overflow Warnings based on Guided Symbolic Execution

BAO Tieyun[1,2,3], GAO Fengjuan[1,2,3], ZHOU Yan[1,2,3], LI You[1,2,3], WANG Linzhang[1,2,3], LI Xuandong[1,2,3]

[1]State Key Laboratory of Novel Computer Software Technology, Nanjing University, Nanjing 210023, China
[2]Jiangsu Novel Software Technology and Industrialization, Nanjing 210023, China
[3]Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China

Abstract Buffer overflow vulnerability is a kind of serious security defect. Currently there are dynamic and static approaches to detect buffer overflow. The effectiveness of Dynamic tools depends on design of test case, and they often introduce execution overhead. Static program analysis techniques have been widely used in buffer overflow detection, which often report a large number of false warnings. Manual validating the results of static analysis is time consuming and error-prone, which severely limits the usefulness of static analysis tools. Symbolic execution is a promising software testing and analysis technology, which systematically explore execution space of test program and generates test cases with high coverage. In this paper we propose a novel approach for automatic buffer overflow warnings validating based on symbolic execution.Our approach is consist of three steps: firstly detect the reachability of statements in static analysis path segment in inter procedural control flow graph and map the static path segment sets to complete path sets which are used to be validated; secondly guide the symbolic execution so that we only focus on the execution paths that cover the buffer overflow warnings generated by static program analysis through pruning useless path; finally construct warning path constraints according to buffer overflow vulnerability models at suspicios statements and classify results depend on the output of constraint solver. Based on the proposed technique we implemented a prototype tool BOVTool and our experimental results on real open source programs show that the percentage of false warnings which do not need to be manually validated is

通讯作者: 鲍铁匀(1991-), 男, 江苏无锡人,硕士研究生,主要研究方向为软件工程、软件测试。高凤娟(1992-), 女, 博士研究生, 研究兴趣为软件安全、web 安全. 周严(1991-), 男, 硕士研究生, 研究兴趣为软件自动化测试、程序静态分析. 李游(1986-), 男, 研究兴趣为软件测试、符号执行. 王林章(1973-), 男, 博士, 教授, 主要研究领域为软件工程、软件安全测试、模型驱动的软件测试及验证. 李宣东(1963-),男,博士,教授,主要研究领域为软件工程、软件建模与分析、软件测试与验证。

本课题得到国家自然科学基金(Nos. 91318301, 61321491, 61472179)资助。
收稿日期: 2015-11-22; 修改日期: 2015-12-23; 定稿日期: 2015-12-30

59.9% on average.

Key words symbolic execution; buffer overflow; warning validating; target guided

1 引言

软件安全漏洞一般是由于程序员的疏忽或者编程语言的局限性等原因遗留在代码中的特殊的缺陷。这些漏洞一旦被攻击者利用会造成非常严重的后果，能够极大削弱软件安全性。缓冲区溢出漏洞已经成为当前威胁极大的软件漏洞之一，据 CVE (Common Vulnerabilities and Exposures)公共漏洞数据库统计2011年和2013年发现的软件漏洞中缓冲区溢出漏洞占到了 19%和 17%左右[1]。缓冲区溢出漏洞在代码层面上属于内存操作类缺陷。造成缓冲区溢出缺陷的原因是程序的缓冲区中可以写入超出其长度的数据，没有进行越界检查而造成溢出，从而有可能破坏程序的堆栈，造成程序崩溃或使程序执行攻击者的指令，存在很高的安全隐患。

目前普遍主要采用两种方式来检测程序中的缓冲区溢出缺陷：静态分析[2-15]和动态测试[16-22]。动态测试指的是通过设计测试用例使得程序执行特定的路径以测试是否存在缺陷。动态测试因为其特性，检测的精准性高，但其弊端在于依赖测试用例以及需要执行开销。静态分析是指不运行软件的前提下对软件代码进行分析的过程 [23]。缓冲区溢出缺陷问题往往发生在特定的控制流路径上，所以静态分析工具一般来说是路径敏感的。静态分析往往事先设定缓冲区溢出特征，扫描待测试程序的源码或者字节码进行特征匹配，所以静态分析并不需要动态执行测试程序，因此不会引入运行时的开销。由于上述特性，静态分析工具在学术界和在工业界更受欢迎。目前比较常见的静态分析工具包括 HP Fortify[24], Splint[25], Klockwork[26]以及 Coverity[27]等等。

然而，静态分析工具在程序规模与结果精准性之间很难取得平衡，以 Fortify 静态分析工具为例，扫描 CVE 公共漏洞数据库中存在缓冲区溢出缺陷的程序，如表 1 所示，分析结果中包含了大量的警报需要人工确认，警报数量也随着程序规模上升而急剧增加。除此以外，静态分析工具常常产生大量的误报，其中可能包含真正的缓冲区溢出缺陷，人工确认这些警报非常耗费时间与精力。确认静态警报通常需要人工阅读相关代码片段并依据审阅者自身经验判断，整个过程依赖于审核者的专业能力，不易自动化，随着警报个数的增加，人工确认流程成本增高，并且极有可能引入误判。

表 1 实际程序使用 Fortify 扫描结果

程序名称	规模	警报总数	缓冲区溢出警报总数
gzip1.24	5.1K	237	19
tftp-hpa5.0	5.3K	78	12
net-tool-1.60	8.1K	1256	62
inspircd2.0.5	74.1K	142	11
udisks2.1.2	92.1K	437	8
tiff4.0.3	113.4K	1743	1117
freetype2.4.8	205.6K	479	16
firefox-33.0	683.3K	23990	4204

符号执行技术是一种传统的软件测试和分析技术，最早提出于 1976 年[28]，该技术使用符号代替具体输入，能够系统地探索程序的状态空间并生成高覆盖度的测试用例。从原理上看，符号执行收集所有可行的路径条件并生成测试用例。然而符号执行技术还存在一些不足：比如与环境(操作系统，网络，数据库等)交互，路径爆炸(可行路径的数量随着路径长度的增长而呈指数增长的趋势[29])，以及浮点数运算等问题。目前存在多种测试工具是基于符号执行的思想实现的，DART[30]结合随机测试和模型检查技术，检测每条路径上的不同的错误类型，然而其对于指针以及循环等结构无法有效处理。KLEE[31]能够模拟环境来解决环境交互的问题，然而其无法模拟整个环境执行。S2E[32]采用选择符号执行的技术，能够自动减少需要符号执行的代码数量，在确保稳定的情况下执行流程透明地在符号领域和实际领域往返。

本文提出一种静态分析制导符号执行的缓冲区溢出警报自动确认技术。我们基于静态测试工具警报，首先在控制流图上进行可达性分析，可达的前提下获取达到特定目标的路径集合；然后指导符号执行沿特定路径执行；最后当符号执行到达溢出疑似位置，分析是否触发缓存区溢出缺陷，并产生相应的测试用例。我们的确认技术能够将缺陷疑似路径分成不可执行路径、安全路径、可溢出路径以及未知路径，并在路径分类的基础上将静态分析警报分为**溢出**、**误报**以及**无法确认**三类。

本文的贡献主要在于：

(1) 提出面向缓冲区溢出缺陷的静态分析警报驱动的自动动态确认技术；

(2) 提出可达性分析用于筛选静态警报确认路径, 提出符号执行目标制导机制用于缩减符号执行状态空间以及基于符号执行的缓冲区溢出检测机制用于警报确认;

(3) 基于上述方法实现缓冲区警报确认工具 BOVTool, 在正确性验证程序和大规模实际程序两组基准程序上进行实例研究, 实验数据表明 BOVTool 能够利用目标制导机制有效引导确认路径执行, 并大量减少缓冲区溢出警报的人工确认数量。

本文的组织结构如下: 第 2 节详细描述我们的方法目标制导符号执行的自动确认技术; 第 3 节介绍原型工具的实现并进行实验和评估; 第 4 节分析静态分析, 动态测试, 静态结果验证等相关工作, 第 5 节总结全文并讨论未来的工作。

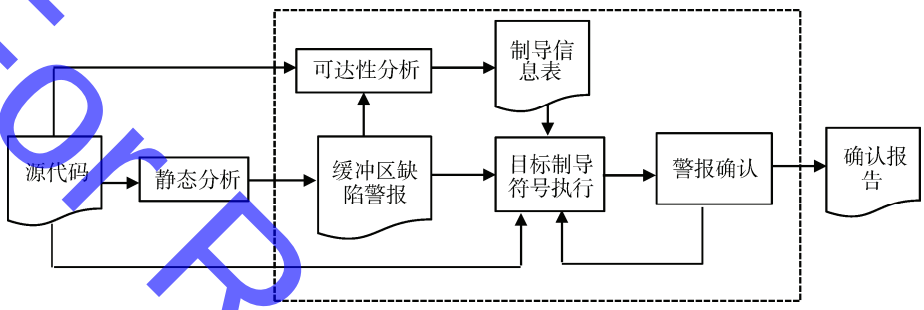


图 1 目标制导的符号执行确认缓冲区溢出缺陷流程

2 目标制导符号执行的自动确认技术

2.1 方法架构

本文提出的基于目标制导符号执行的缓冲区溢出确认方法如图 1 所示, 首先使用静态分析工具对待测试程序源代码进行分析, 获取程序中可能存在缓冲区溢出缺陷的语句类型, 位置以及潜在可能触发缺陷的语句组合作为静态分析警报。符号执行的确认过程分为 3 个步骤: 可达性分析、目标制导以及警报确认。首先抽象程序, 建立程序控制流图分析程序入口与静态警报的目标缺陷语句之间的可达性, 如果可达抽取路径信息构建制导信息表; 其次将制导信息表, 静态警报的目标缺陷语句以及程序源代码作为输入提供符号执行引擎, 指导符号执行过程沿设定的确认路径执行; 最后监测执行流程, 判断符号执行是否到达缺陷疑似点, 通过语句类型提取溢出条件约束, 连同路径约束进行约束求解, 分析求解结果, 最终得到静态警报的分类结果。

下面将用一个示例程序来说明我们的想法:

```
1 . # define MAX_LEN    24
2. # define    MIN_LEN    4
3. void usage ( ) {
4.          char des_buffer[MIN_LEN];
5.          char* src_buffer="source buffer";
6.          strcpy(des_buffer, src_buffer);
7.    }
8. void initialize( char* argv_string){
9.          char mapped_argv[MIN_LEN];
10.       if( strlen(argv_string) == 0)
11.             return;
12.       if( strlen(argv_string) >= MAX_LEN)
13.             return;
14.        if(argv_string[0] != '-')
15.             strcat(mapped_argv,'-');
16.       strcpy(mapped_argv, argv_string);
17.}
18. int main (int argc, char **argv) {
19.        initialize (*argv);
20.}
```

图 2 示例程序

如图 2 所示的代码片段用静态分析工具 Fortify 扫描后, 发现的缓冲区溢出静态警报有 3 处, 分别是 1:{4,6}; 2:{9,15}; 3: {18,19,9,16}, 这里大括号前面的内容表示警报编号, 大括号内的内容表示组成警报路径片段的程序源码语句行号。静态警报路径片段的起始点往往是缓冲区分配语句, 也有可能是函数调用语句, 终止点一般是缓冲区操作语句。由于程序中的大部分警报出现在 initialize 函数中, 我们为 initialize 函数构造其函数内控制流图以便更加清楚分析和展示我们的方法。如图 3 所示, initialize 函数(简称 i)中存在 4 条路径, 所有的分支以及分支约束都以标签的形式加以区别:

①strlen(argv_string)≠0;
②strlen(argv_string) == 0;
③strlen(argv_string) < MAX_LEN;
④strlen(argv_string)>= MAX_LEN;
⑤argv_string[0] == '-'

⑥argv_string[0] ≠ '-';

如果我们构造全局过程间控制流图, 第 1 条警报路径片段, 第 4 行和第 6 行不会映射到控制流图中, 因为他们所在的函数没有被调用即无法执行, 这里我们将其判断为不可达路径, 无需进行后续验证。

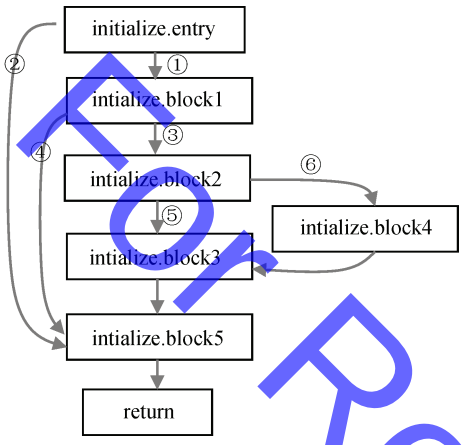


图 3　initialize 函数控制流图

对于第 2 条警报路径片段{9,15}, 第 9 和 15 行分别映射到基本块 i.entry 和 i.block4, 在控制流图中能够覆盖警报路径片段的基本块路径为

i.entry-> i.block1-> i.block2-> i.block4, 在确认该路径的过程中, 我们将第 15 行溢出疑似语句的溢出条件加入到路径约束中进行求解, 即 strlen(argv_string) ≠ 0 ^ strlen(argv_string) < MAX_LEN

^argv_string[0] ≠ '-' ^ strlen('-') +

strlen (argv_string)>size(mapped_argv), 我们无法找到满足这样的约束条件的 mapped_argv, 所以我们认为第 2 条静态分析警报安全路径, 而且由于第 15 行所在的语句只有一条路径能够到达, 所以我们认为第 15 行语句是误报。在实际程序中对于某个静态警报可能会包含多个可达路径, 只有每一条路径都被确认为安全路径, 我们的工具才会判定该缺陷语句为误报点。对于第 3 条警报路径片段第 9 和 16 行分别映射到基本块 i.entry 和 i.block5, 在控制流图中存在两条可达路径 i.entry-> i.block1-> i.block2-> i.block3-> i.block5 以及 i.entry-> i.block1-> i.block2-> i.block4-> i.block3-> i.block5, 出于确认正确性考虑, 两条路径都需要被验证, 这里我们选择前者为例, 路径约束和溢出条件为 strlen(argv_string) ≠ 0^ strlen (argv_string) < MAX_LEN^ argv_string[0] == '-'^

strlen(argv_string)>size(mapped_argv), 我们可以找到满足该约束的 argv_string 以及 mapped_argv。事实上 argv_string 来自于 main 函数参数, 没有越界检查确实有可能超过缓冲区 mapped_argv 的最大长度, 所以我们认为第 3 条警报路径片段是可溢出路径。静态分析的结果中只要有某条可达路径为可溢出路径, 我们的工具就会判定该缺陷语句为溢出点。在确认过程中, 我们的目标制导机制则用于解决如何避免执行示例中的 1 条无用路径, 而选择需要确认的 3 条路径执行。

2.2　可达性分析

控制流图构建　我们首先利用编译器为待测试程序生成中间指令集合, 并根据指令之间的跳转关系划分基本块, 生成函数内的控制流图; 其次我们从函数内控制流图出发, 依据函数调用指令分析调用关系构造过程间的完整控制流图; 最后我们逆转基本块的指向关系构造双向控制流图, 进行此步骤的目的出于对大规模程序的分析效率的考虑, 因为我们后续的路径寻找基于深度优先搜索, 从缺陷语句所在的基本块开始反向搜索能够有效减少遍历节点的数目。

算法 1: 从函数内控制流图集合生成过程间控制流图

输入: 函数内控制流图集合以及 *main* 函数的入口基本块 *BasicBlock*

输出: 过程间完整双向的控制流图

```
ConstructRCFG(BasicBlock B)
1:   CFGNode current = initialize(B)
2:   FOR each Instruction in B
3:      IF current instruction is function call (f is callee) THEN
4:         current->addSucc(f->entryBlock())
5:         f->entryBlock()->addPrev(current)
6:      IF f->entryBlock() has not been visited THEN
7:         ConstructRCFG (f->entryBlock())
8:            END IF
9:         current = f->returnBlock()
10:      END IF
11:  END FOR
12:  FOR each succ in B->succSet DO
13:      current->addSucc(succ)
14:      succ->addPrev(current)
15:      IF succ has not been visited THEN
16:         ConstructRCFG (succ)
17:      END IF
18:  END FOR
```

该算法描述了从函数内控制流图构建完整双向过程间控制流图的过程, 在实现过程中, 我们能够利用编译器获取函数内的控制流图, 图中的每个节点为称为 *BasicBlock*, *BasicBlock* 只存在后继关系,

为了能够构建过程间的双向控制流图, 我们将其封装为 *CFGNode*, 能够表示节点之间的前驱和后继关系, *CFGNode* 为我们下文中所说的基本块。我们用 *current* 指向当前遍历的控制流图节点(第 1 行), *initialize* 函数将 *BasicBlock* 封装为 *CFGNode*。在遍历当前节点的所有后继节点之前, 我们首先遍历 *current* 基本块中的每条指令检测是否存在函数调用指令(第 2-11 行), 如果存在的话(假设 f 为被调用函数), 将 *current* 的后继指向被调用函数的入口基本块 *entryBlock*(第 4 行), 随后如果该函数还没有访问过, 递归访问被调用函数, 当被调用函数返回的时候, 将 *current* 指向被调用函数的返回基本块(第 9 行), 这样被调用函数的返回基本块的后继指向调用指令所在基本块的后继, 算法的 12-18 行描述将 *current* 的后继指向基本块 *B* 的后继, 并对于每一个后继递归调用 *ConstructRCFG*, 此时 *current* 指向的可能是基本块 *B*, 也可能是基本块 *B* 中调用函数的返回基本块。这里我们在构建过程间完整控制流图的同时就已经添加反向指针(第 5 行, 第 14 行), 而非构建完成后再次遍历添加。

基本块映射 我们首先给出缓冲区溢出缺陷的静态分析警报ω的概念, 其中使用标签集合 L 标记和识别程序中的语句集合, ω由三元组构成(l_1,<$l_2,l_3,\cdots,l_{n-1}$>, l_n), 其中 $l_1,l_2,\cdots,l_n \in L$, l_1 为缓冲区定义语句的标签, l_n 为缓冲区操作语句的标签, 通常为缺陷疑似语句, < $l_2,l_3,\cdots,l_{n-1}$>为一系列静态分析与缓冲区相关的中间语句标签, < $l_2,l_3,\cdots,l_{n-1}$>可以为空, 它们与标签 l_1 以及 l_n 代表的语句共同构成缓冲区溢出警报路径片段。我们的方法能够处理的静态分析输出结果格式为可扩展标记语言(XML), 是大部分静态分析工具都能够支持的。

我们将缓冲区溢出警报路径片段中语句标签映射到双向控制流图的基本块中。我们以三元组(l_1,<$l_2,l_3,\cdots,l_{n-1}$>, l_n)标记静态分析警报, 对于其中的语句标签我们以<*f,l*>二元组唯一标记, 其中 *f* 和 *l* 分别为程序的文件名和行号。源代码中每个语句标签的二元组信息与每个基本块中的指令集合的静态信息进行匹配即可映射。我们记录被映射到的基本块, 由于这些基本块之间不一定可达, 下一部分我们进行可达性分析。

可达性分析 我们已经将静态警报片段中的语句位置映射到双向控制流图的基本块中, 我们需要确认映射到的基本块之间的可达性以及程序入口至基本块之间的可达性, 基本的搜索策略采用深度优先搜索, 我们在映射到的基本块两两之间, 以及程序入口基本块与 l_1 所在基本块之间调用 *SearchPaths* 进行可达确认。具体的算法如算法 2 所示:

算法 2: 生成起始基本块 *v* 和终止基本块 *des* 之间所有基本块路径集合

输入: 控制流图中的起始节点 *v* 和目标节点 *des*, *temp* 为深度优先遍历中的当前路径, *TempSet* 为用于保存的路径的集合

输出: *v* 和 *des* 之间所有基本块路径集合

SearchPaths(*CFGNode v, CFGNode des, SethPath TempSet , Path temp*)

```
1:  IF visited(v)==TRUE THEN
2:      return
3:  END IF
4:  temp.push(v)
5:  IF v == des THEN
6:      TempSet.add(temp)
7:      return
8:  ELSE
9:      visited(v) = TRUE
10: FOR each succseeor s_i of v DO
11:     IF visited(v)=FALSE THEN
12:     SearchPaths(s_i,des,TempSet,temp)
13:         temp.pop
14:     END IF
15:     visited(v)= FALSE
16: END FOR
```

算法 2 描述了如何在过程间控制流图中确认静态路径映射到的基本块之间的可达性, 事实上我们通过搜索基本块之间所有可达路径, 如果可达路径集合为空则说明不可达, 否则我们收集所有路径并进行后续步骤。我们通过深度优先搜索的策略遍历每个基本块节点。如果当前节点已经被访问过, 那么直接返回(第 1-2 行), 否则把当前节点加入到当前路径 *temp* 中(第 4 行)。如果当前节点为目标节点, 那么我们找到一条可达路径, 将其加入路径集合, 否则不为目标节点, 我们将其设为已经被访问过, 对于当前节点的每一个后继递归调用 *SearchPaths* 进行访问。在一个节点的后继都被访问完成之后, 我们将其设为没有被访问过, 目的是为了搜索所有可达路径。对于循环模块表现为控制流图中的环, 我们的处理策略是在静态分析时决定是否进入循环, 而循环的次数将依赖于符号执行支撑工具的处理。

路径选择与拼接 上个步骤中我们通过 *SearchPaths* 得到各个基本块之间, 程序入口至路径起始基本块的所有路径片段。这些路径片段由控制流图的基本块所组成, 我们需要从多个路径片段中选择确认的路径, 我们认为这些路径片段对于确认

效果来说是等价的, 可以根据实际情况设定不同的选择策略。考虑在确认过程中尽可能减少误报情况的出现, 我们的方法则是保留了所有可能的路径组合。拼接上述路径集合即可得到从待测试程序入口到缺陷疑似语句的完整可达路径。

制导信息表构建 制导信息表由潜在溢出缺陷路径ρ集合组成, ρ可以理解为从程序入口至缓冲区溢出点所在的基本块之间的可达路径所抽取出的分支入口信息。由于某个警报的缺陷语句可能存在多条可达路径, 那么ω和ρ存在一对多的对应关系。制导信息表作为符号执行的额外输入, ρ才是我们在真正确认验证的路径。最后警报分类的结果是基于ρ的分类结果, 这将在 2.4 章节说明。

2.3 目标制导

符号执行是一种传统的用于软件测试和分析的技术[28], 其目的是尽可能的遍历程序的状态空间, 并产生高覆盖度的测试用例。符号执行的基本思想是用符号代替实际输入, 在执行过程中遇到分支则复制已有的环境信息, 并收集相关路径约束。当执行到程序出口或发现错误时, 根据收集到的约束条件求解, 产生相应的测试用例。符号执行的整个执行过程实际上就是选择下一个执行状态并对该状态中所包含的指令进行解释的过程, 符号执行中的某条路径可以认为是由多个有序执行状态以及约束组成的集合。传统的搜索策略描述符号执行过程中如何选择下一个执行状态, 关注于如何提高程序覆盖度, 我们的目的则是探索并执行静态分析中的缓冲区潜在溢出缺陷路径。随着程序规模的增大, 符号执行存在状态爆炸的问题。为了使得符号执行能够避免路径爆炸, 也使得其能够确认特定的缺陷疑似路径, 我们设计了一种轻量型的机制能够实现目标制导, 这种机制的基本思想在于无用路径剪枝, 实现该机制的关键则在于如何确定无用路径以及如何尽早修剪无用路径。前者依据静态警报的结果, 与潜在溢出缺陷路径不相匹配的即为无用路径, 而符号执行中路径是由多个符号执行状态组成的, 即转换为不相匹配的状态检测。如果仅仅只有静态警报片段, 符号执行只有在某条路径执行完成之后才能判断是否覆盖了警报片段, 我们将潜在溢出缺陷路径作为额外输入提供给符号执行引擎, 在分支指令处分裂新的状态时抽取ρ的信息比较, 删除不相匹配的状态。

算法 3 描述了目标制导的符号执行技术路径剪枝算法, 在符号执行的过程中需要维护一个状态池 *ESVector* 用于保存当前所有执行的状态, 整个符号执行过程的终止条件为状态池中已经没有状态可以选择或者到达了设定的时间阈值(第 2 行), 每次执行的过程分为 3 个步骤(第 4-6 行): 首先从状态池中选择下一个状态, 然后解释状态内部指令, 在解释指令的过程中会有状态增加或者删除操作, *addedStates* 和 *removedStates* 分别用于临时保存即将变更的状态, 最后 *updateStates* 操作更新状态池。对于某条指令的解释过程体现在算法的第 7-16 行, 首先判定该指令的类型是否为退出或者错误触发类型, 是则将其加入 *removedStates* 并根据路径约束条件求解得到测试用例(第 8-11 行), 否则继续判断其类型是否为分支指令, 复制当前状态, 添加相反约束, 将新的状态加入 *addedStates*, 调用 *GuidedExecution* 模块进行路径剪枝(第 12-16 行)。具体的路径剪枝过程在算法 17-27 行, 我们将两个相反分支的入口信息与制导信息表中的信息对比, 删除不符合潜在溢出缺陷路径的分支(第 21-25 行)。

算法 3: 符号执行的目标制导算法

```
Vector<executionState> addedStates
//符号执行过程中增加状态集合
Vector<executionState> removedStates
//符号执行过程中删除状态集合
Vector<executionState> ESVector
//保存所有符号执行状态的集合
executionState initialState;
//符号执行的初始化状态
Vector<branchInfo> ValidatingPath;
1.  ESVector.add(initialState);
2.  WHILE ESVector.size>0 || !TIME OUT DO
3.      executionState ES = selectState()
4.      ES.executeInstruction()
5.      updateStates(ESVector, addedStates, removedStates)
6.  END WHILE
7.  executeInstruction()
8.  IF ES.instructionType = EXIT || FoundError THEN
9.      generateTestCase();
10.     removedStates.add(ES);
11. END IF
12. IF ES.instructionType = FORK THEN
13.     ES2=fork(ES);
14.     addedStates.add(ES2);
15.     GuidedExecution(ES, ES2);
16. END IF
17. GuidedExecution (executionState ES, executionState ES2)
18. IF ES! =NULL && ES2 ! =NULL THEN
```

19. <F_1,L_1>=*BranshesDebugInfo*(*ES*);
20. <F_2,L_2>=*BranshesDebugInfo*(*ES*2);
21. **IF** *ValidatingPath.contain*(<F_1,L_1>) && *! ValidatingPath.contain*(<F_2,L_2>) **THEN**
22. *removedStates.add*(*ES*2);
23. **END IF**
24. **IF** *! ValidatingPath.contain* (<F_1,L_1>) && *ValidatingPath.contain*(<F_2,L_2>) **then**
25. *removedStates.add*(*ES*);
26. **END IF**
27. **END IF**

为了确保剪枝过程的正确性，即不存在误删的情况，当相反分支的入口信息都出现或者都不出现在制导信息表中时，我们都不做处理。原因有以下两点：(1)我们的目标制导技术目前无法处理库函数调用，因为这部分信息在控制流图上是缺失的，库函数中的路径我们无法选择如何删减；(2)某种缺陷的触发依赖于循环的迭代次数，而我们的溢出确认路径不含包含循环信息。如果静态分析警报的溢出疑似位置出现在某个循环内部，那么符号执行在达到缺陷语句后继续执行直到下一次循环，之后的制导信息存在缺失的情况。

由于路径剪枝机制的局限性，我们目前只能够实现目标的近似制导，而非精确制导，实际情况可能是一组路径片段中包含了与静态结果相匹配的某条路径。但是从实验结果来看，我们实现的机制与传统的符号执行相比已经能够大大提高时间和空间效率，至于如何实现更加精确的制导以及控制符号执行过程中循环的执行次数，我们考虑将其作为后续工作。

2.4 缓冲区溢出警报确认

2.4.1 缓冲区溢出模型

为了扩展符号执行缓冲区溢出确认的功能，我们首先定义了缓冲区溢出模型。缓冲区溢出主要分为相关 API 的调用以及缓冲区直接访问两类。

缓冲区相关的 API 调用 为了使得我们的缓冲区溢出模型涉及的操作更具实用性和普遍性，我们参照了 C99 标准中可能存在缓冲区溢出缺陷的操作和 Linux 中常见系统调用，并根据参数的格式分成 8 个类型，在表 2 显示了不同类别的缓冲区操作的溢出条件。我们构造缓冲区溢出条件的基本思路是写入的字符串长度是否大于缓冲区本身的长度，但是在实际实现的过程中，我们很难获取符号化字符串的长度信息，我们使用 *len(b)*表示字符串 *b* 的字节长度，*len(b)=i*, 满足$\forall j$, $0 \leqslant j < i$, $P_b[j] \neq$'\0', $P_b[j]$ ='\0'。为了确认的统一性，*len(b)*的定义被扩展至符号化字符串以及常量字符串。

表 2 缓冲区溢出模型定义

类型	缓冲区操作	溢出条件
1	strcpy(des,src)	*len(src)>=size(des)*
2	strncpy(*des,src,n*), memcpy, memmove memset, snprintf, vsnprintf	*n>size(dest)*
3	strcat(*des,src*)	*len(src)+len(dest)>=size(des)*
4	strncat(*dest,src,n*)	*min{len(src),n}+len(dest)>=size(str)*
5	sprintf(*str,fomat,....*),vsprintf	*length(format string)>=size(str)*
6	fgets(*str,num,stream*)	*num>size(str)*
7	fead(*ptr,size,count,stream*)	*size*count>size(ptr)*
8	read(*fd,buf,count*)	*count>size(buf)*

缓冲区直接访问 对于直接数组或者指针访问，我们提出如下的约束模型：如果通过数组访问，表示为 *buf*[*i*]=*ch*，其中 *buf* 表示字符串，*i* 表示据起始地址的偏移量，*ch* 表示给对应地址的赋值。在上述模型中，缓冲区溢出的条件为 *i*>*size*(*buf*)。如果通过指针访问则表示为*(*buf*+*i*)=*ch*，在我们的模型中溢出条件同样为 *i*>*size*(*buf*)。

2.4.2 缓冲区溢出警报确认

相关变量符号化 与缓冲区溢出相关的变量可能来自于程序外部，也可能来自于程序内部，如果来自于程序内部则不存在符号化的过程。如果变量内容来自程序外部环境(main 函数参数，文件读取，网络传输)， 且符号执行工具无法模拟该环境来解决库函数的调用，那么我们调用 API 手动符号化相关变量。

跟踪符号执行状态 使用目标制导机制后符号执行在路径剪枝的过程中，我们跟踪并监测每个状态，在执行符号状态包含的指令之前获取其指令类型，如果发现其为函数调用指令，则获取调用函数的类型以及名称，如果其符合我们处理的类型，则进入以下步骤处理。如果是非函数调用指令，则分析其类型是否为指针或者数组访问指令，如果是则进

入下列处理步骤。

构建缺陷约束 在符号执行的过程中, 我们修改符号执行支撑工具引擎使得其执行到溢出疑似语句时收集当前的路径约束并且获取缺陷约束。如果是操作缓冲区的 API, 那么我们从符号执行状态中抽取出参数, 根据设定的缓冲区溢出模型构成溢出约束条件。如果是非函数调用并且为指针或者数组访问指令, 那么我们抽取出指令的起始地址以及偏移量等相关信息来构建溢出模型。最后我们将缺陷约束加入路径约束组成完整的路径缺陷约束。除此以外, 我们将当前状态复制出一个相同的分支, 一个分支用于构建缺陷约束, 求解获取分类结果, 在约束求解完成之后终止执行, 另一个分支状态则作为原有状态不受影响继续执行。

约束求解并分类 我们将制导信息表中的潜在溢出缺陷路径ρ分为以下 4 类。其中我们使用标签集合 L 标记和识别程序中的语句集合, 每个标签我们以<*f,l*>二元组唯一标记, 其中 *f* 和 *l* 分别为程序的文件名和行号, 使用φ表示约束。

不可执行路径 I: 潜在溢出缺陷路径中存在语句标签 l_i 和 l_j, 在控制流图上不可达, $l_i, l_j \in L$。

安全路径 S: 潜在溢出缺陷路径的路径约束 $\varphi_1 \wedge \varphi_2 \wedge \cdots\cdots \wedge \varphi_n$ 与缓冲区溢出缺陷的静态分析警报ω的标签 l_n 处 φ_0 溢出约束条件满足 $\varphi_1 \wedge \varphi_2 \wedge \cdots\cdots \wedge \varphi_n \wedge \varphi_0 = false$, 则该溢出确认路径称为安全路径。

可溢出路径 O: 潜在溢出缺陷路径的路径约束 $\varphi_1 \wedge \varphi_2 \wedge \cdots\cdots \wedge \varphi_n$ 与缓冲区溢出缺陷的静态分析警报ω的标签 l_n 处 φ_0 溢出约束条件满足 $\varphi_1 \wedge \varphi_2 \wedge \cdots\cdots \wedge \varphi_n \wedge \varphi_0 = true$ 有解, 则该溢出确认路径称为可溢出路径, 约束 $\varphi_1 \wedge \varphi_2 \wedge \cdots\cdots \wedge \varphi_n \wedge \varphi_0$ 的解可触发缓冲区溢出缺陷。

未知路径 U: 在设定的时间内, (1)没有或无法判定执行到标签 l_n 处, 无法得到溢出约束条件 φ_0; 或者(2)潜在溢出缺陷路径的路径约束 $\varphi_1 \wedge \varphi_2 \wedge \cdots\cdots \wedge \varphi_n$ 与缓冲区溢出缺陷的静态分析警报ω的标签 *b* 处 φ_0 溢出约束条件满足 $\varphi_1 \wedge \varphi_2 \wedge \cdots\cdots \wedge \varphi_n \wedge \varphi_0 \neq false$ 且 $\varphi_1 \wedge \varphi_2 \wedge \cdots\cdots \wedge \varphi_n \neq true$.

先前的研究工作表明程序中 9%-40%的路径可以通过静态分析可以判定为不可达[33], 不可达路径意味着动态运行过程中该路径不可能被执行。我们通过静态分析的缓冲区溢出警报路径片段中语句之间的可达性, 提前筛选出静态警报中不可能动态执行的路径片段, 在确认过程中即可忽略上述路径, 提高确认效率。

安全路径和可溢出路径是通过符号执行进行静态警报确认结果的不同分类, 安全路径我们认为该路径不存在缓冲区溢出的可能, 可溢出路径我们认为符号执行执行该路径生成的测试用例可以导致缓冲区越界, 即有被攻击的可能, 但是溢出的数据覆盖其他内存部分是否导致严重的后果, 具体还是要取决于特定测试用例的内容。

静态分析工具报告中警报往往可能对应不止一条可达路径, 上述的分类分析只是针对确认路径, 我们设定如下的判断准则用于判定如何从潜在溢出缺陷路径集合到静态警报的判定结果。

判定准则 1: 缓冲区溢出静态警报ω, 以及其分析得到对应的潜在溢出缺陷路径集合{ρ}, ∃ρ∈{ρ},ρ满足 **O**, ω为缓冲区溢出缺陷。

判定准则 2: 缓冲区溢出静态警报ω, 以及其分析得到对应的潜在溢出缺陷路径集合{ρ}, ∀ρ∈{ρ}, ρ满足 **S** 或者ρ满足 **I**, ω为静态分析缺陷误报。

判定准则 3: 缓冲区溢出静态警报ω, 以及其分析得到对应的潜在溢出缺陷路径集合{ρ}, ∃ρ∈{ρ}, ρ满足 **U**, 且∃ρ∈{ρ}, ρ满足 **O**, ω为无法确认。

准则 1 的意思指: ω所对应的集合{ρ}中, 只要有一条路径确认为可溢出, 那么ω确认为缓冲区溢出缺陷; 准则 2 的意思指: ω所对应的集合{ρ}中, 如果所有待验证路径都为误报, 那么ω确认为静态分析缺陷误报; 准则 3 的意思为不存在可溢出的路径且存在未知路径, 那么ω无法确认。

3 实现与评估

3.1 原型工具实现

我们在 Linux 3.11.0 系统下实现了基于符号执行的静态缓冲区溢出警报确认的原型工具 BOVTool。在本文中我们使用 HP Fortify v3.2 作为缓冲区溢出静态分析工具, 静态可达性分析基于 LLVM2.9[34]平台, 我们选择 KLEE 作为符号执行工具, 修改其核心源码以实现目标制导以及缓冲区溢出确认功能. 所有的实验都在 Intel Corei5-2400 3.10GHz 处理器的双核工作站执行。

我们选择使用 LLVM 平台进行编译以及分析的原因有以下两点: (1)LLVM 平台前端编译器编译能够生成函数内控制流图, 便于后续的过程间控制流图的构建和分析; (2)KLEE 的底层实现就是对 LLVM 中间代码的解释过程, 这使得我们的前端分析与后台确认都基于同一个对象。

BOVTool 的静态分析依赖于 LLVM 平台的 PASS 模块, LLVM 的优化和转换工作就是由一个或者多个 PASS 模块完成的。我们实现新的 PASS 模

块用于构建和遍历双向的过程间控制流图。我们首先使用 llvm-gcc 对待测试程序进行编译, 生成的中间代码是控制流图分析以及符号执行的输入。我们对过程间控制流图分析后生成制导信息表, 连同静态分析警告作为符号执行的额外输入。

我们的原型工具自动化执行程度相对较高, 但是在实际的实验过程中, 仍然存在以下需要人工干预的步骤: (1) 源程序需要修改 makefile 用于 llvm-gcc 编译器编译, llvm 编译时可能会出错, 有时需要修改源代码使得编译通过; (2) 由于宏定义的存在, 某些缺陷疑似语句可能不会编译进入符号执行输入文件, 需要人工修改宏定义; (3) 某些函数如 getenv(获取环境变量赋值给 char* 变量)等 klee 不支持, 需要手动使用 klee_make_symbolic 对该变量符号化。

我们的工具扩展了符号执行的目标制导以及缓冲区溢出检测两个功能模块, 我们希望实验数据能够回答以下几个问题:

Q1: 工具对缓冲区溢出静态警报的确认准确性如何?

Q2: 工具能够确认警报的比例如何?

Q3: 工具在确认过程中的时间和空间效率如何?

Q4: 工具目标制导的效率如何?

3.2 实验 1: 准确性确认程序

我们从 GNU COREUTIL 库[35]中选取 8 个测试程序, 这些程序相对于真实系统规模较小, 便于我们人工确认分类结果是否准确。在待测试的程序中, 除了本身存在的缺陷以外, 我们人工植入若干真正的缓冲区溢出缺陷代码片段以及能够造成静态分析工具误报的代码片段。前者的代码片段植入比较简单, 我们随机选择存在缺陷的缓冲区溢出片段插入到程序随机某处同时保证程序能够成功编译。后者的植入相对困难, 因为我们无法得知静态分析工具的实现细节, 即何种特定的代码模式能够触发静态工具产生误报。我们通过对 COREUTILS 中的 chmod.c 的静态结果分析发现, 程序中使用 x2realloc 等非标准库函数动态分配内存空间时, 经过人工确认在某种特定条件会引起静态分析工具的误报, 我们将这种代码模式实例化成与程序上下文信息无关的代码片段, 随机植入到待测试程序的不同位置并使用我们的工具进行确认。我们为程序的每条确认路径执行时间设置了阈值, 因为植入位置的随机性, 可能导致某些路径的缺陷疑似点过于远离程序入口而导致在时间阈值内无法执行到。

表 3 准确性确认程序分类结果

(1)程序名称	(2)基本信息			(3)路径分类				(4)位置点分类			(5)正确性确认	
	#L	#W	#P	#IP	#SP	#OP	#UP	#OS	#FS	#US	FS ∩T	OS ∩F
chmod	602	12	12	2	2	7	1	7	4	1	0	0
pr	1904	8	15	1	4	3	4	3	3	2	0	0
pwd	406	12	16	3	7	3	3	3	7	2	0	0
sort	3306	22	27	2	7	8	8	8	7	7	0	0
su	567	6	8	0	3	1	3	1	3	2	0	0
ls	4572	31	42	5	17	4	10	4	19	8	0	0
pr	2940	29	31	5	10	11	3	11	15	3	0	0
df	1014	18	18	1	8	9	0	9	9	0	0	0

表 3 中列举了我们用于实验的 8 个测试程序的实验数据。在表 3 基本信息中, #L, #W 以及#P 分别表示测试程序的规模(以行数表示), 静态分析报出的位置点个数以及缺陷疑似路径数。第三和第四部分分别是对缺陷疑似路径和位置点进行分类。路径的分类结果以 4 类表示(#IP, #SP,#OP,#UP 分别表示不可执行路径, 安全路径, 可溢出路径以及未知路径), 位置点的分类结果以 3 类表示(#OS,#FS,#US 分别表示溢出语句, 误报语句, 未知语句集合)。第五部分正确性确认表示人工确认结果。

结论 1: BOVTool 对缓冲区溢出静态警报的确认准确性高。在实验数据的第五部分中, T 和 F 分别表示人工检查静态分析结果后的真实和误报的缓冲区溢出语句集合, FS∩T 和 OS∩F 分别表示工具检测的误报语句集合与真实溢出语句集合的交集, 以及工具检测的真实溢出语句集合与溢出误报语句集合的交集, 数据表明两个集合的交集均为 0。这说明我们的工具不存在将真实溢出语句划分为误报, 也不存在将误报溢出语句划分为真实溢出语句的情况。因为未知语句集合的存在, 我们没有将两个集合是否相等来作为工具准确性判定标准。这里我们并没有再次对路径分类的结果进行人工确认, 因为从路径

分类到语句分类的判定准则决定，一旦路径分类的结果出现错误，位置语句的分类结果也会出现错误。

除了在小型程序中植入错误验证以外，在大规模实际程序中我们也对 BOVTool 的分类正确性进行验证，验证结果如表 4 中第 5 部分正确性确认部分所示，我们事先知道了程序中缺陷具体位置，T 表示真实存在的缓冲区溢出语句集合，FS 表示 BOVTool 确认误报语句的集合，两者取交集表示是否存在分类错误的情况，两个集合并不存在交集。对比 BOVTool 确认的缓冲区溢出缺陷个数与待测试系统中真实存在的溢出个数，两者并不匹配，部分测试程序确认的个数小于程序中真实存在的个数。我们人工检查静态分析警报，发现这是由于静态分析工具存在严重的漏报情况。而 Fortify 等静态分析工具如何设定更加精准的缺陷特征匹配机制来减少漏报情况的发生并不在我们考虑的范畴之内。

表 4　大规模实际程序确认结果

(1)程序名称	(2)基本信息			(3)路径分类			(4)位置点分类			(5)正确性确认
	#L	#W	#P	#IP∪#SP	#OP	#UP	#OS	#FS	#US	FS∩T
polymorph-0.40	0.3K	19	11	19	0	0	11	0	0	0
bc1.06	9.7K	15	5	11	1	3	3	1	1	0
net-tool-1.60	8.1K	1256	62	763	251	242	36	6	20	0
wwwcount2.3	8.3K	112	20	51	1	61	5	1	14	0
gzip1.2.4	5.1K	237	19	148	4	85	12	1	6	0
sendmail8.12.7	78.1K	4854	96	3067	1	1786	51	1	44	0

3.3　实验 2: 大规模实际程序

我们通过 COREUTILS 中若干人工植入缺陷的程序确认，保证工具确认的正确性，接下来我们将确认已经存在缓冲区溢出大规模实际程序作为实验确认对象。我们通过在大规模实际程序中比较不同版本，查看修复日志以及检索程序主页公布的缺陷等方式定位缓冲区溢出在程序中的位置。

结论 2: BOVTool 能够有效减少人工确认警报数量。表 4 中#IP∪#SP 以及#OP 表示确认的误报路径和可溢出径条数, #OS 和#FS 分别表示溢出语句和误报语句集合。我们可以发现代表未知路径#UP 和未知语句#US 的个数往往是相对较少的，图 4 显示了 6 个测试程序从语句和路径两种粒度上能够确认得到的有效分类数量在所有静态分析结果的数量中所占的比例，对于不同的程序而言 BOVTool 可以有效确认 45.5%-100%的警报路径，平均来说可以确认 60.1%的警报语句，而对于 COREUTILS 中的 8 个测试程序，平均来说可以确认 81.9%的警报语句，这意味着我们的工具能够有效减少人工检查的警报数量。

我们分析实验结果中存在未知路径以及未知语句的情况，原因有: (1)在有限的资源(比如内存)或者有限的时间内，符号执行无法覆盖到缺陷疑似的程序位置点，因而我们无法获取完整的路径约束，无法做出判断; (2)编译过程中静态信息的缺失，导致符号执行无法判断是否已经覆盖到警报中的缺陷疑似语句; (3)依赖的约束求解器没有能力求解缺陷路径约束。

结论 3: BOVTool 的目标制导机制是有效的。表 5 中#SA, #RA, #SV 分别表示静态分析阶段，制导路径分析阶段以及符号执行确认阶段; #VT 和#VC 表示不使用目标制导确认时间和内存。图 6 表示利用目标制导技术以后符号执行确认过程中相比于全路

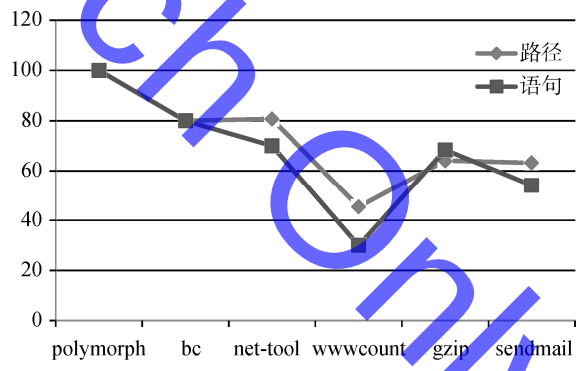


图 4　减少人工确认数量的百分比

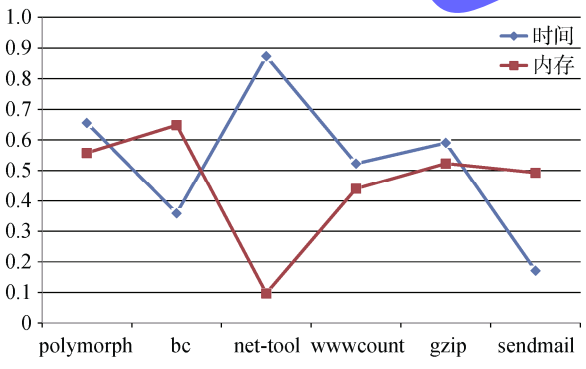


图 5　使用目标制导后减少的时间和内存

表 5 确认时间(单位 秒)和内存开销(单位 MB)

(1) 程序名称	(2) 时间消耗			(3) 内存消耗			(4) 非制导	
	#SA	#RA	#SV	#SA	#RA	#SV	#VT	#VC
polymorph-0.40	83.7	0.11	101.2	413.7	2.4	36.1	294.2	87.1
bc1.06	289.3	3.1	125.9	541.7	22.2	77.4	201.1	283.4
net-tool-1.60	206.5	39.8	282.7	465.9	68.0	394.5	2547.7	512.3
wwwcount2.3	291.1	9.1	159.3	490.9	19.4	225.1	352.6	435.9
gzip1.2.4	175.5	16.66	169.2	431.4	15.1	64.2	453.7	166.2
sendmail8.12.7	6240.4	145.8	5758.2	865.7	109.2	1435.1	7124.1	3046.9

径覆盖减少的时间和内存。时间节省在 net-tool 程序中最高可达 87.3%, 内存节省在 bc-1.06 中最高可达 64.9%。平均来看我们的制导技术可以优化时间和空间分别为 52.8%和 46.0%。通过实验数据我们发现: (1)不同的程序之间时间和内存节省相差较大, 这可能由于待测试程序中库函数调用数量不同, 因为符号执行对库函数有独特的处理方式, 这一部分并没有在我们的目标制导机制中有效的解决。(2)对于同一个测试程序, 目标制导技术没有在时间和内存节省上同时是最优的, 这可能是由于目标制导机制尽管删除部分符号执行状态, 相比于这部分状态中的指令的解释执行时间减少, 时间更多花费于路径约束的求解。

3.4 实验讨论

我们的方法也存在以下几点限制: (1)在使用 LLVM 构建控制流图的过程中, 我们发现有 3 类函数调用不能够被连接到过程间的控制流图: ①被调用的函数只有声明, 但是没有定义。这种函数虽然能够成功编译但是我们并不能够找到其函数体; ②被调用的函数是函数指针, 我们仍然无法找到其调用的函数, 因为我们仅从中间代码无法得到动态运行信息; ③被调用的函数是内联汇编表达式, 同样我们无法得到其函数体[36]。(2)对于循环结构我们并没有给出更加适合的处理方式, 导致我们抽取出的路径可能无法完成精确制导。我们仍然使用所依赖的符号执行工具中原有的循环处理策略, 为了保证探索所有路径空间, 从循环不执行到上界次数穷尽执行, 而在我们确认警报的背景下, 这种策略可以优化; (3)确认的方法实现基于符号执行支撑工具 KLEE, 我们对警报分类依赖于路径约束和溢出条件的约束求解结果, 而 KLEE 的约束求解能力受限于所依赖的 STP 求解器。除此以外, KLEE 能够通过模拟环境来解决环境交互的问题, 程序对库函数的调用会被重定向到设定的模型中, 这些模型能够理解程序的语义并产生正确的约束, 然而其并不能够模拟整个环境中所有情况, 即有些库函数的调用无法被识别, 除此以外, 库函数关联的模型所产生的路径数量可能远大于程序本身的路径数量, 而这部分路径我们无法通过制导的方式删减。

实验数据表明我们的方法能够有效减少静态分析工具警报缓冲区警报的人工确认数量, 并且我们的确认方法正确性比较高, 没有存在将真实溢出缺陷分类为误报或者将误报分类为溢出缺陷的情况。但是我们的方法仅有在有限的测试用例集上评估了确认方法的有效性和准确性, 因此不能声称分类系统对所有的实际程序都有绝对的准确性。当然我们在实验过程中仅用#OS 和#FS 的数目来衡量我们的方法能够减少的人工确认静态分析警报的成本。虽然实验结果表明我们能够有效减少人工确认的数量, 但是数据结果无法衡量减少的数目如何影响测试人员确认的时间以及如何影响人工确认警报的难度。在后续工作中我们也考虑设计其他度量准则来衡量我们方法减少确认的时间和难度。

4 相关工作

与我们工作相关的主要是缓冲区溢出的静态分析, 动态测试, 静态分析制导动态测试, 以及静态结果验证等工作。

4.1 静态分析

静态分析往往通过缺陷特征匹配的方式来检测缓冲区溢出缺陷。部分工具对源代码进行分析, 如 ITS4[2]扫描 C 或者 C++源代码并将其分割成词法记号, 在词法流中匹配缓冲区溢出的缺陷模型以检测可能存在的溢出漏洞; 类似的工具 FlawFinder[3]能够输出细节更多的测试报告并且能够支持更多的源代码类型。上述工具原理简单且能够运用于大规模程序但是仅仅考虑词法信息不够完善。另外部分工具则考虑了语法信息: BOON[4]关注字符串类型数据的操作, 通过设定定义约束语言来对变量边界建模, 检测是否存在某些操作能够使得字符串越界; Splint[5]则是使用各类轻量型的静态分析技术, 对于所有缓

冲区操作库函数的可疑操作都给出警告。由于缓冲区溢出往往发生在特定的路径中，部分工具考虑路径敏感分析: ARCHER[6]对源代码采用路径敏感的过程间符号分析检测变量内存边界，其主要限制在于考虑程序中所有可执行路径，然而大部分路径与缓冲区溢出缺陷触发是无关的; Marple[7]同样通过路径敏感分析将路径分类，并通过驱动分析使其能够运用于大规模程序，缺点在于缺少环境信息，库函数调用等动态执行信息而存在大量误报。AEG[8]首先使用符号执行在源代码粒度上检测缺陷，然后在字节码粒度上使用符号执行生成的测试用例动态执行来确认。TaintScope[9]与 IntScope[10]使用路径敏感分析方法在中间表示层上使用符号执行检测溢出发生位置。Yamaguchi[11]等人提出一种新颖的程序特性图来表示源代码，特性图中包含了抽象语法树，控制流图，程序依赖图等，他们能够使用程序之间的继承关系来对不同的常见缺陷更加清晰的建模。Chucky[12]则是通过污染传播的静态分析方式检测安全关键对象比如内存或者缓冲区的缺失引用。

目前也有相关一部分工作将符号执行应用于二进制级别以及其他语言检测。MergePoint[13]则在二进制上融合了静态符号执行与动态符号执行，能够应用于大规模应用软件的测试。Saxena[14]等人则将符号执行应用于 Javascript 中脆弱性检查，比如代码注入等。BitBlaze[15]结合了静态，动态分析以及程序验证技术，能够抽取安全相关特性用于检测缓冲区相关的内存问题。我们的方法能够与现有的静态分析方法集成，作为有效的补充，提高静态分析工具的实用性。

4.2 动态分析

动态测试缓冲区溢出采用的方法往往在待测试程序中插入特殊的代码片段以检测缺陷的发生并给出合适的处理方式比如停止程序执行。有部分动态检测工具关注返回地址来检测缓冲区溢出，StackGuard[16]编译器在栈数据和返回地址之间放置 canary，检测其完整性以鉴别是否发生缓冲区溢出攻击。StackShield[17]则是通过全局变量记录每个函数的返回地址，检测缓冲区溢出攻击转变为其返回地址是否已经修改。数组边界检查[18]定义了基指针以及每个指针的访问区域，是否存在某个指针的内存访问越界。Ccured[19]需要程序编写人员为指针类型进行注解，通过辨别不同种类的指针，阻止不合理的指针使用，保证程序的操作不会访问不合适的内存区域。Array Bounds Checking[20]定义基本指针和每个指针的合法区域，通过验证内存访问是否在合法区域内，该工具能够进行越界检查。Wagner[21]通过设计非确定性有穷状态自动机来对应用程序行为规约以及特征化预期的系统行为，如果真实行为违背设定的规约工具能够给出警告。Splat 工具[22]能够自动生成测试用例用于缓冲区溢出的检测，该工具基于制导的随机测试与符号长度抽象技术缩小执行空间并能够保持缺陷检测的能力。我们的方法基于符号执行，与动态测试相比较，能够有效节省测试资源，并且能够自动生成测试用例。

4.3 静态分析结果验证

静态分析结果包含了大量误报严重影响工具的实用性，目前也存在许多研究工作研究静态工具的分析结果验证。Kim[37]等人通过分析程序的不同版本之间缺陷的修复历史来对静态分析结果进行分类统计。Dillig[38]等人提出一种基于诱发推测的算法来分析静态分析中丢失信息而对静态分析结果进行诊断。Ruthruff[39]等人通过警报信息与信息相关联的代码来建立回归模型预测静态分析结果的类别。与我们的工作最相似的是 Li Mengchen[40]于 2013 年提出的静态分析工具内存泄漏的动态确认技术，该方法使用混合执行技术，根据静态分析信息记录并更新内存对象状态并对静态内存泄漏结果进行确认，然而该方法需要人工指定源代码中的符号化输入，人工对符号化变量的标记包含了工具使用者对程序语义的理解，对于复杂的输入数据，需要人工提前设计可以触发某些缺陷路径的测试用例，加重了测试人员的负担，而我们的方法大部分情况下能够实现自动化测试输入符号化，不同于他们的工作需要设计特定的测试输入用于触发缺陷疑似路径，我们的方法依赖于符号执行探索程序空间，通过路径剪枝的方法在状态空间中执行并验证缺陷疑似路径。Babic[41]等人的工作是使用静态分析制导动态执行自动为二进制代码生成测试用例，他们首先使用种子测试构建下推自动机来反映整个程序的控制流，其次使用静态分析检测潜在的漏洞，最后使用带权重的下推自动机驱动符号执行自动生成测试用例。

5 总结

静态分析方法被广泛应用于实际程序中的缓冲区溢出缺陷检测。尽管静态检测工具能够发现程序中潜在的缺陷，但是静态分析报告时常输出数量庞大的误报需要人工确认。人工确认缺陷位置往往是枯燥，耗时并且也存在确认错误的可能。我们提出面向缓冲区溢出缺陷的目标制导符号执行方法，基于静态分析警报，首先在控制流图上进行可达性分析，

可达的前提下获取达到特定目标的路径集合; 然后指导符号执行沿特定路径执行; 最后当符号执行到达溢出疑似位置, 分析是否触发缓存区溢出缺陷, 并产生相应的测试用例。我们将静态警报分为误报, 溢出以及无法确认三种类型。人工确认时主要关注我们的方法无法确认的静态警报。实验结果表明我们的工具能够有效减少静态缓冲区警报中的误报, 提高静态分析工具的实用性。

我们的工作目前针对的是 C 语言中静态分析警报中缓冲区溢出缺陷的确认, 然而我们的方法也能扩展到 C 语言中的其他常见的、且容易存在静态分析误报缺陷类型比如内存泄露等, 关键在于研究缺陷发生的机理以及如何跟踪符号执行过程并构造缺陷模型。除此以外, 我们的方法也可以扩展到其他语言比如 java, python 等, 当然这首先依赖于这些语言的符号执行支撑工具实现与开发。

致 谢 在此向对本文的工作给予建议的老师、参与本文实验的所有同学、给我们提出建议的评审专家表示感谢。

参考文献

[1] http://www.cvedetails.com/vulnerabilities-by-types.php.

[2] J. Viega, J.T. Bloch and Y. Kohno. “A Static Vulnerability Scanner for C and C++ Code”, *Proceedings of the 16th Annual Computer Security Applications Conference(ACSAC’00)*, pp.257-269,2000.

[3] WHEELER, D. Flawfinder home page. Web page: http://www.dwheeler.com/flawfinder.

[4] D. Wagner, JS. Foster, EA Brewer and A. Aiken. “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities”, *Proceedings of the Network and Distributed System Security Symposium (NDSS’00)*, 2000.

[5] D. Evans and D. Larochelle. “Improving security using extensible lightweight static analysis”, *IEEE software*, vol.19, pp.42-51, 2002.

[6] Y. Xie, A. Chou and D. Engler. “Archer: using symbolic, path-sensitive analysis to detect memory access errors”, *ACM SIGSOFT Software Engineering Notes*. vol.28, No.5, pp.327-336, 2003.

[7] W. Le and M.L. Soffa.“Marple: a demand-driven path-sensitive buffer overflow detector”, *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (FSE/ESEC’08)*. ACM, pp.272-282,2008.

[8] T. Avgerinos, S.K. Cha, BLT. Hao and D. Brumley. “AEG: Automatic Exploit Generation”, *Proceedings of the Network and Distributed System Security Symposium (NDSS’11)*, vol.11, pp. 59-66, 2011.

[9] T. Wang, T. Wei, G. Gu and W. Zou. “TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection”, *IEEE Symposium on Security and Privacy(SP’10)*, pp. 497-512, 2010.

[10] T. Wang, T. Wei, Z. Lin and W. Zou. “IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution”.*Proceedings of the Network and Distributed System Security Symposium(NDSS’09)*, 2009.

[11] F. Yamaguchi, N. Golde, D. Arp and K. Rieck. “Modeling and discovering vulnerabilities with code property graphs”. *IEEE Symposium on Security and Privacy(SP’14)*, pp.590-604, 2014.

[12] F. Yamaguchi,C. Wressnegger, H. Gascon and K. Rieck.“Chucky: exposing missing checks in source code for vulnerability discovery”.*Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, pp. 499-510,2013.

[13] T. Avgerinos, A. Rebert, SK. Cha and D. Brumley. “Enhancing symbolic execution with veritesting”, *Proceedings of the 36th International Conference on Software Engineering (ICSE’14)*. pp. 1083-1094,2014.

[14] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant and D. Song. “A symbolic execution framework for javascript”, *IEEE Symposium on Security and Privacy(SP’10)*, pp.513-528,2010.

[15] D. Song, D. Brumley, H. Yin, J. Caballero, … and P. Saxena. “BitBlaze: A new approach to computer security via binary analysis”, *Information systems security*. Springer Berlin Heidelberg, pp.1-25,2008.

[16] C. Cowan, C.Pu, D. Maier, J. Walpole, P. Bakke,S. Beattie, A. Grier, P. Wagle, and Q. Zhang. “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”. *Proceedings of the 7th USENIX Security Symposium*, vol.98, pp.63-78, 1998.

[17] Vendicator. Stack Shield technical info file v0.7. http://www. angelfire.com/sk/stackshield/info.html.

[18] E. Haugh, M. Bishop. “Testing C Programs for Buffer Overflow Vulnerabilities”.*Proceedings of the Network and Distributed System Security Symposium(NDSS’03)*. 2003.

[19] G. C. Necula, S. Mcpeak, W. Weimer. “CCured: Type safe retrofitting of legacy code”. *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL’02)*, vol.37, pp.128-139,2002.

[20] R.W.M. Jones, P.H.J. Kelly. “Backwards compatible bounds checking for arrays and pointers in C”. *Automated and Algorithmic Debugging (AADEBUG’97)*, 1997.

[21] D. Wagner, D. Dean. “Intrusion detection via static analysis”. *IEEE Symposium on Security and Privacy (SP’01)*. pp.156-168, 2001.

[22] R.G. Xu, P. Godefroid, R. Majumdar. “Testing for buffer overflows with length abstraction”. *Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA’08)*. ACM,

pp.27-38, 2008.

[23] H. Mei, Q.X. Wang, L. Zhang, J. Wang. “Software analysis: A road map”.*Chinese Journal of Computers*, vol.32, No.9, pp.1697-1710, 2009.
(梅宏, 王千祥, 张路, 王戟, “软件分析技术进展”, *计算机学报*, 2009, 32(9): 1697-1710.)

[24] HP Fortify. http://www.fortify.com.

[25] Splint. http://www.splint.org/.

[26] The Klocwork static analysis tool. http://www.klocwork.com.

[27] The Coverity static analysis tool. http://www.coverity.com.

[28] L. Clarke. “A system to generate test data and symbolically execute programs”. *IEEE Transactions on Software Engineering*, pp. 215-222, 1976.

[29] P. Joshi, K. Sen, M. Shlimovich. “Predictive testing: amplifying the effectiveness of software testing.” *European Software Engineering Conference*. pp. 561-564, 2007.

[30] P.G.N. Klarlund and K. Sen. “DART: directed automated random testing". *ACM Sigplan Nofices*.Vol.40.No.6 ,2005.

[31] C. Cadar, D. Dunbar and D. R. Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”,*USENIX Symposium on Operating Systems Design and Implementation(OSDI'08)*.vol.8, pp.209-224. 2008.

[32] V. Chipounov, V. Kuznetsov, G. Candea. “S2E: a platform for in-vivo mult-ipath analysis of software sysfems”. *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS'11)*. Vol. 47. No. 4. pp. 265-278, 2012.

[33] R. Bodik, R. Gupta and M. L. Soffa. “Refining data flow information using infeasible paths”, *ACM SIGSOFT Symposium on the Foundation of Software Engineering/ European Software Engineering Conference (ESEC/FSE'97)*. Springer Berlin Heidelberg, pp. 361-377, 1997.

[34] The LLVM Project. http://llvm.org/.

[35] Coreutils GNU core utilities. http://www.gnu.org/software/coreutils/

[36] Chris Lattner, LLVM Language Reference Manual. http://llvm.org/releases/2.9/docs/LangRef.html#moduleasm.

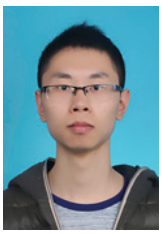
[37] S. Kim and M.D. Ernst. “Prioritizing warning categories by analyzing software history”.*Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, vol. 27, 2007.

[38] I. Dillig and T. Dillig, A. Aiken. “Automated error diagnosis using abductive inference” *ACM SIGPLAN Notices*, vol.47, No.6, pp. 181-192, 2012.

[39] J.R. Ruthruff, J. Penix, J.D. Morgenthaler, S. Elbaum and G. Rothermel. “Predicting accurate and actionable static analysis warnings: an experimental approach”.*Proceedings of the 30th international conference on Software engineering(ICSE'08)*. ACM, pp.341-350, 2008.

[40] M. Li, Y. Chen, L. Wang, G. Xu. “Dynamically validating static memory leak warnings”.*Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'13)*. ACM, pp.112-122, 2013.

[41] D. Babić, L. Martignoni, S. Mccamant and D. Song. “Statically-Directed Dynamic Automated Test Generation.” *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11)*. ACM, pp. 12-22, 2011.



鲍铁匀 于 2013 年在南京大学计算机科学与技术专业获得学士学位。现在南京大学计算机科学与技术系攻读硕士学位，研究领域与研究兴趣包括软件工程，软件安全测试。Email: baotieyun@seg.nju.edu.cn

高凤娟 于 2014 年在电子科技大学计算机科学与技术专业获得学士学位。现在南京大学计算机科学与技术专业攻读博士学位。研究领域为软件工程，软件测试。研究兴趣包括：软件安全，web 安全。Email: fjgao@seg. nju.edu.cn

周严 于 2014 年在南京大学计算机科学与技术专业获得学士学位。现在南京大学计算机科学与技术专业攻读硕士学位。研究领域为软件测试。研究兴趣包括：软件自动化测试，程序静态分析。Email: zhouy@seg.nju. edu.cn

李游 于 2008 年在南京大学计算机科学与技术专业获得学士学位。现于南京大学计算机系软件工程组攻读博士学位。研究兴趣：软件测试，符号执行。Email: leo86@seg.nju. edu.cn

王林章 于 2005 年在南京大学取得计算机软件与理论专业博士学位。现任南京大学计算机科学与技术系教授。研究领域为软件工程、信息安全。研究兴趣包括: 软件安全性测试、模型驱动的软件测试及验证、自动化软件测试工具。Email: lzwang@nju.edu.cn

李宣东 于 1994 年在南京大学取得计算机软件与理论专业博士学位。现任南京大学计算机科学与技术系教授。研究领域为软件工程、信息安全。研究兴趣包括: 软件建模与分析、软件测试与验证、自动化软件测试工具。Email: lxd@nju.edu.cn