软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2018,29(1):1−22 [doi: 10.13328/j.cnki.jos.005375]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

精准执行可达性分析:理论与应用*

杨 克[1,2], 贺也平[1,2,3], 马恒太[1], 王雪飞[1]

[1](中国科学院 软件研究所 基础软件国家工程研究中心,北京 100190)

[2](中国科学院大学,北京 100049)

[3](计算机科学国家重点实验室(中国科学院 软件研究所),北京 100190)

通讯作者: 杨克, E-mail:yangke2015@iscas.ac.cn; 贺也平, E-mail:yeping@iscas.ac.cn

摘 要: 精准执行可达性分析探究计算机程序状态之间的可达性关系,通过分析软件的文档、源代码或二进制程序并进行必要的测试验证,以求出在既定限制下从初始状态到特定代码位置的目标状态的准确触发输入和执行路径.精准执行可达性分析在定向测试、静态分析结果核验、错误复现和漏洞POC构造等领域均有广泛的应用.对近年来国内外学者在该研究领域取得的相关研究成果进行了系统的分析、提炼和总结.首先,指出了精准执行可达性分析对应的约束求解问题,以双向符号分析和程序归纳为主线介绍了其主要研究方法,讨论了相关技术难点;其次,对目前已经存在的精准执行可达性应用进行了分类分析;进而,指出精准执行可达性分析应用中程序分析、归纳和约束求解等方面存在的挑战;最后,对可能的解决办法以及未来发展方向进行了展望.

关键词: 程序分析;可达性分析;定向测试;双向符号分析;程序归纳

中图法分类号: TP301

中文引用格式: 杨克,贺也平,马恒太,王雪飞.精准执行可达性分析:理论与应用.软件学报,2018,29(1):1−22. http://www.jos.org.cn/1000-9825/5375.htm

英文引用格式: Yang K, He YP, Ma HT, Wang XF. Precise execution reachability analysis: Theory and application. Ruan Jian Xue Bao/Journal of Software, 2018,29(1):1−22 (in Chinese). http://www.jos.org.cn/1000-9825/5375.htm

Precise Execution Reachability Analysis: Theory and Application

YANG Ke[1,2], HE Ye-Ping[1,2,3], MA Heng-Tai[1], WANG Xue-Fei[1]

[1](National Engineering Center of Fundamental Software, Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

[3](State Key Laboratory of Computer Science (Institute of Software, The Chinese Academy of Sciences), Beijing 100190, China)

Abstract: The research of precise execution reachability analysis focuses on figuring out the reachability between program states. It tries to find witness inputs and the execution traces that pass through the setting-up target state of certain code location by performing necessary test and verification on executable files, source code and documentation. Precise execution reachability analysis has been applied to direct testing, bug reproduction, construction of proof of concepts of vulnerabilities, verification for result of static analysis and so on. This paper provides a survey of this area. First, the corresponding constraint solving problem of precise reachability analysis is cited. Next, existing typical methods and technical difficulties about bidirectional symbolic analysis and program induction, and some technical difficulties are discussed. Then, the applications of current precise reachability analysis are classified and summarized. Furthermore, the challenges on program analysis, program induction and constraint solving are provided. Last but not least, the possible solution and future research are suggested.

• 基金项目: 国家科技重大专项(2014ZX01029101-002)
Foundation item: National Science and Technology Major Project (2014ZX01029101-002)
收稿时间: 2017-04-18; 修改时间: 2017-05-31; 采用时间: 2017-08-18; jos 在线出版时间: 2017-10-09
CNKI 网络优先出版: 2017-10-09 16:20:51, http://kns.cnki.net/kcms/detail/11.2560.TP.20171009.1620.002.html

Key words: program analysis; reachability analysis; targeted testing; bidirectional symbolic analysis; program induction

计算系统的可达性分析研究在给定状态迁移规则下,确定一个状态集合是否可以从系统中特定的初始状态集合到达.由于计算机程序准确刻画了计算机在不同输入下的状态迁移行为,一般研究能否以及如何构造输入使得某段计算机程序执行到目标状态,称其为程序的执行可达性分析,简称执行可达性分析.莱斯定理[1]指出,图灵机可接受语言的一切非平凡性质是不可判定的.执行可达性也是一个不可判定性质,因此需要根据具体的应用场景,对执行可达性分析结论的完备性和可靠性做出折衷.

传统的执行可达性分析的应用场景主要是静态缺陷检测、一般性测试、编译和软件安全属性的形式化验证等,其仅对目标非安全状态进行了最基本的描述,例如活代码、竞争、死锁、溢出、空指针解引用和内存泄露等.这些应用场景注重分析的全面性,对目标状态所在的程序代码位置不做限制,对到达目标状态的执行路径的不同执行轨迹不作区分,也并不关心其所引发的额外攻击行为和危害.这类执行可达性分析应用可以帮助捕获一些软件错误的基本信息,或发现一些疑似缺陷.

但是,这些信息只展示了一个错误在某错误类中的共性特征,无法反映该错误在该程序中的特有性质,例如有多少种触发方式,能否被远程利用、能否造成代码执行、如何修复等.因此,缺陷检测报告中的程序缺陷需要被复现和调试以进一步分析、评估其对程序可靠性和安全性的影响,最终正确地修复.对缺陷较为集中的程序模块还需进行更加深入、细致的定向测试,下面通过两个例子来说明进一步对触发缺陷状态的执行路径进行深入辨别和分析的必要性.

图 1 显示了 GnuTLS(<3.1.22,3.2.x-3.2.11)中存在的一个缺失检查错误(CVE-2014-0092)和一条触发路径(阴影部分代码).函数 *_gnutls_verify_certificate2* 通过调用函数 *check_if_ca* 来验证证书发行商的有效性,缺少了对负数返回值(*check_if_ca* 返回 0 和负数表示验证失败)的检查.导致该问题的根源是 *check_if_ca* 函数的返回值设计不合理,负数没有带来特殊的信息反而容易引发使用上的错误.修复补丁除了修改第 3 行的对 *check_if_ca* 的检查外,也在 *check_if_ca* 中第 18 行的"goto cleanup;"前增加"*result*=0;"的设置.由于触发这一错误的路径不止一条(图 1 中经过 25 行的路径就是另外一条).*check_if_ca* 函数中有 7 处"goto cleanup;"语句,修复时很容易漏掉某个导致 *result* 为负数的分支,这有可能导致其他使用 *check_if_ca* 函数的地方发生类似的错误.这段代码直到 GnuTLS-3.3.17 才得到了相对妥当的处置,最终将 *check_if_ca* 的返回值类型更正成了 Bool 型.

```
1  int _gnutls_verify_certificate2 (...)
2  {   ...
3      if (check_if_ca(...)==0) {
4        result=0;
5        goto cleanup;
6      }
7      ...
8      result=1;   //summary no exception
9  cleanup:
10     return result;
11 }
12 int check_if_ca (...)
13 {   ...
14     result=_gnutls_x509_get_signed_data(...);
15     if (result<0)
16     {
17       gnutls_assert ();
18         goto cleanup;
19     }
20     ...
21     result=gnutls_x509_get_signed_data(...)
22     if ((result<0...
23     {
24       gnutls_assert();
25       goto cleanup;
26     ...
27     result=0;
28 cleanup:
29     return result;
30 }
```

Fig.1 A missing check in GnuTLS (CVE-2014-0092)

图 1 GnuTLS 的一个缺失检查错误(CVE-2014-0092)

该案例表明:只获取一条触发错误的执行路径是不够的,还需要全面分析不同触发路径,这有助于全面认识和修复缺陷.应用在传统测试中的执行可达性分析技术大都依赖于崩溃时的调用栈,未能区分图 1 所示的这种程序错误的多路径触发现象.

图 2 显示了 iwconfig(wireless tools-26)的一个缓冲区溢出漏洞的漏洞利用的示意图.iwconfg 从 main 函数经 *print_info* 将可执行程序 iwconfig 的第 1 个参数 *argv*[1]传递到 *get_info* 函数的 ifname,中途没有作合法性检

查,就直接交给第7行的*strcpy*进行字符串复制.普通的测试或动态分析只能根据崩溃时操作系统的信号以及调用栈来区别不同的执行路径,对溢出多少字节以及是否执行了嵌入在输入内容中的攻击代码(shellcode,用虚线标记)不加以区分.但对漏洞利用而言,这些问题十分重要.如果不考虑操作系统的地址随机化(ASLR)、数据执行保护(DEP)以及各种栈溢出防护机制,希望通过该错误构造一个代码执行攻击,如借助溢出打开一个终端,那么所构造的输入(*argv*[1])应该仿照图 2 中下部的十六进制内容那样,将原返回地址所在的 4 个字节(用下划线标记)覆盖为 shellcode 起始地址,从而在 *get_info* 函数返回时修改 EIP 指针转到 shellcode 起始位置,执行打开终端的汇编指令.填充字节(由 0x01 组成)、返回地址("60 f3 ff bf"即 0xbffff360)和 shellcode("31 c0 50 68...")共同组成了触发输入.图 2 左下第 1 列是十六进制运行时栈地址,左下第 2 列是从 0 开始的十六进制长度计数,每行 16 个字节.右上角显示了栈中关键数据的内存布局.从上到下为内存地址增长方向.完成该攻击需要准确地判定运行时栈中的 *get_info* 返回地址距离输入 iframe 的长度以及 iframe 在内存中的位置,进而构造恰当的输入内容.

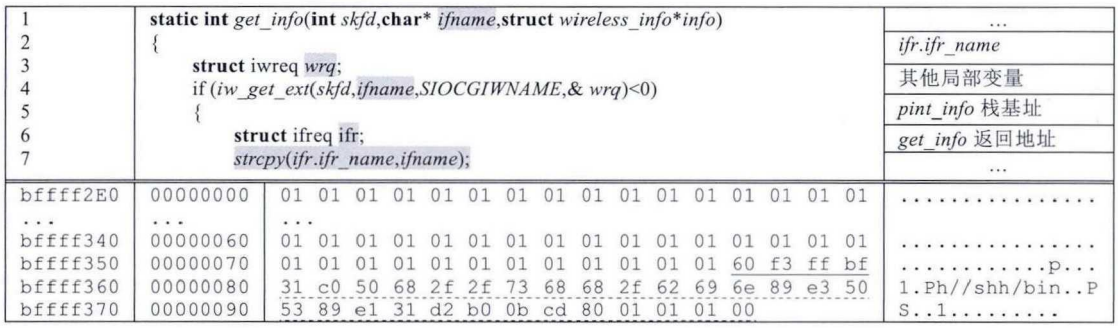

Fig.2 An input to exploit buffer overflow bug in iwconfig (CVE-2003-0948)

图 2 iwconfig 缓冲区溢出漏洞 CVE-2003-0948 的漏洞利用示意

以上两个案例显示:深入分析一个错误,需要站在不同的角度对触发路径进行细致的分类和分析,从而更加全面地理解其触发条件、影响和危害.这类有针对性的分析和测试工作包含定向测试、错误复现、静态分析结果核验、漏洞利用构造等.相比传统的执行可达性分析问题,这类应用中的执行可达性分析问题有着更加明确的执行目标,对触发路径增加了更强的限制,并要求给出准确的触发输入和触发路径.

本文将这类对程序代码位置确定的目标状态寻找定制化执行路径和触发输入的问题称为精准执行可达性分析问题.表 1 显示了精准执行可达性分析常见应用相比传统执行可达性分析应用所增强的限制条件.

Table 1 Difference of restrictions between normal and precise execution reachability analysis

表 1 精准/传统执行可达性分析应用在问题限制上的对比

传统	精准	增加的限制条件
非定向测试	定向测试	生成触发特定位置代码(或一组特定位置的代码)的执行的差异化测试用例
静态缺陷检测	静态分析结果的核验	测试用例触发的动态执行路径与静态分析得到的路径片段相吻合
动态缺陷检测	错误复现	调用栈、崩溃信息或错误日志需与错误报告一致
动态缺陷检测	漏洞利用构造	触发特定位置的缺陷,在满足运行时环境的限制下完成给定的攻击行为

与传统执行可达性分析相比,求解精准执行可达性分析问题需要特别考虑如下两个关键问题.

(1) 如何将附加限制条件进行合理的表示和建模,与程序代码和运行时信息进行有效的关联和匹配,并实现对定制约束和程序执行约束的关联求解;

(2) 如何根据触发目标和其他附加限制条件对分析过程进行有效的引导,缩小搜索和分析范围.

相应地,精准执行可达性分析也存在一些特有的挑战.

- 第一,复杂场景的综合约束带来关联分析挑战.目标执行状态或定制的约束条件中可能包含的约束是多维度的,可能涉及高层软件设计和领域知识,可能涉及源代码实现,也可能涉及底层编译得到的二进制文件以及运行时信息(内存布局、进程/线程调度),还可能关系到操作系统和硬件等其他环境因素.

如何自动获取相关信息,将不同抽象层面的动态和静态约束有效地关联起来,建立适合的数学模型进行求解是一个难题;

- 第二,目标状态的精确程序代码位置约束,给引导路径测试带来挑战.静态程序分析对完整控制流图的构造困难[2]以及外部调用等不可见逻辑区,造成了以目标状态为起点的逆向最弱前置条件推理的中断.面向非固定位置目标的正向路径分析所使用的路径探索策略[3,4]以提升覆盖率为导向,缺乏对可达性关系的深入分析[5].并发和带有随机数的程序给执行过程注入了更多的状态和不确定性.这种夹杂在程序中的复杂性和信息的不确定性,给有效地指导动态分析带来了挑战.

较难建模的约束常见于漏洞利用自动构造研究中,除了完全按照人工调试和分析的步骤实现自动化(如AEG[6,7])外,最近也开始有一些针对可用数据进行建模和分析以提高分析效率的尝试,如 FlowStich[8]构造带有时间维度的数据流图(2D-DFG)考察变量随执行时间的活跃性,辅助寻找可利用的指针和数据配对.目前的研究大都局限于对缓冲区溢出自动利用研究,而且这些分析也很零散,不成体系.更多形式的约束场景和执行可达性关联建模问题仍有待挖掘.

现有的解决精准执行可达性分析问题的方法大都采用正向符号执行和逆向最弱前置条件推理相结合的方式,在起点和目标之间形成的纺锤形路径空间的两端尽早剪枝,通过寻找和调整邻近交汇的双向路径来发现到达目标状态的可达路径.这种方法又被称为"双向符号分析(bidirectional symbolic analysis)"[9].在这类方法中,通常利用启发式方法和最短距离(即到达目标状态的最短路径上的语句或指令数目)指导路径探测方向.其合理性探讨及更有效的策略研究还有待开展.另外,双向符号分析依然需要解决循环、递归带来的路径爆炸问题、外部 API 调用的建模问题、指针分析问题、运行时决定的函数调用和间接跳转的控制流识别等问题.这些问题也是传统执行可达性分析中的研究难点.

为了深入理解精准执行可达性分析在理论和应用上面临的挑战和问题,寻找新的研究方法,本文从理论和应用两方面对该领域现有的研究工作进行了综述研究.主要工作是:

(1) 对精准执行可达性分析基本理论和研究现状进行总结,指出精准执行可达性研究在程序分析方面存在的技术难点;

(2) 对最近出现的精准执行可达性研究的应用成果进行分类分析,指出现有技术在实践中的不足,以及需要克服的技术问题.

1 精准执行可达性分析相关理论

为了使得研究问题更加清晰,这里尝试给出一个相对具体的精准执行可达性分析问题定义.借用符号执行研究中的常用记号[10],用代码位置 l 以及变量到符号表达式的映射表 σ:$Var\rightarrow SymbolicExp$ 构成的二元组表示抽象执行状态(l,σ),用 PC_p 表示使得执行路径 p 上的分支条件成立的变量约束.执行路径 p 由依次执行的代码位置序列构成 $p=\langle l_1,l_2,\ldots,l_n\rangle(n>0)$.这里沿用类似的符号体系.为方便描述问题的解,用代码位置 l 以及变量到具体值的映射表(下文称为具体赋值)σ_c:$Var\rightarrow ConcreteValue$ 构成的二元组表示具体执行状态(l,σ_c).所有具体赋值组成的全集用 Σ_c 表示.

精准执行可达性分析问题形式地定义为:在先后经过程序 P 的入口代码位置 l_s 和目标代码位置 l_t 的执行路径集(用 $\mathbb{P}=Path(P,l_s,l_t)$ 表示)中,找出执行路径 p,并求出对应的触发输入 $\boldsymbol{X}\in\{0,1\}^m,m\in\mathbb{Z}^+$,使$(p,\boldsymbol{X})$满足如下 5 条.

(1) 输入满足限制 $M(\boldsymbol{X})$=True;

(2) 入口状态 $s=(l_s,\sigma_c^{(s)})$ 满足约束 $S(\sigma_c^{(s)})$=True,且具体赋值 $\sigma_c^{(s)}$ 由输入 $\boldsymbol{X}$ 部分初始化,即 $I_s(\boldsymbol{X})\subseteq\sigma_c^{(s)}$;

(3) 目标状态 $t=(l_s,\sigma_c^{(s)})$ 满足约束 $T(\sigma_c^{(t)})$=True;

(4) 路径 p 是可执行的(满足控制流上的分支条件),即 $PC_p(\sigma_c^{(t)},\boldsymbol{X})$=True,入口状态变量取值表 $\sigma_c^{(s)}$ 在路径 p 和输入数据 $\boldsymbol{X}$ 的作用下得到目标状态的变量取值表 $\sigma_c^{(t)}$,即 $\sigma_c^{(t)}=f_p(\sigma_c^{(s)},\boldsymbol{X})$;

(5) 满足额外的路径限制 $C(p)$=True.

其中,$M:\{0,1\}^m\to\{\text{True,False}\}$表示对输入的限制断言,如长度、内容等,$\sigma_c^{(s)},\sigma_c^{(t)}\in\Sigma_c$,分别表示入口状态 s 和目标状态 t 的具体赋值;$S:\Sigma_c\to\{\text{True,False}\}$,$T:\Sigma_c\to\{\text{True,False}\}$,$I_s:\{0,1\}^m\to\Sigma_c$表示输入 $\boldsymbol{X}$ 对 s 中部分变量的初始化函数;$C:\mathbb{U}\to\{\text{True,False}\}$为对执行路径的约束断言,其中,$\mathbb{U}$为程序 P 中的执行路径全集.$C(p)$=True 表示对执行路径的额外限制,如路径 p 必须调用或绕过某些函数等;路径约束 PC_p 是执行路径 p 关于初始赋值的二值函数(断言/逻辑公式),$PC_p:\Sigma_c\times\{0,1\}^m\to\{\text{True,False}\}$,其限定了控制流;$f_p:\Sigma_c\times\{0,1\}^m\to\Sigma_c$是由执行路径 p 形成的计算函数,其限定了数据流.PC_p,f_p称为基本约束,其余约束称为附加约束.

精准执行可达性分析问题实际就是求集合 $\{(p,\boldsymbol{X})\mid p\in Path(P,l_s,l_t),PC_p(\sigma_c^{(s)},\boldsymbol{X})=\text{True},\sigma_c^{(t)}=f_p(\sigma_c^{(s)},\boldsymbol{X}),M(\boldsymbol{X})=\text{True},I_s(\boldsymbol{X})\subseteq\sigma_c^{(s)},S(\sigma_c^{(s)})=\text{True},T(\sigma_c^{(t)})=\text{True},C(p)=\text{True}\}$,是一个约束求解问题.

1.1 分析方法

精准执行可达性分析专注于特定软件逻辑切面.该切面限定了有效分析区域,由实际应用场景决定.例如,测试某段补丁代码时只关心该段代码影响到的执行路径和功能逻辑,而复现错误时关心的是能够引发相同错误信息或日志的执行路径.为了避免无效分析,需要分析方法根据应用限制,有效地剥离和识别这些区域,避免分析不相关的代码.相比目标代码位置不确定且无额外定制约束的传统执行可达性分析问题,精准执行可达性分析问题对分析路径规划的有效性和语句、状态间触发关系推理的准确性提出了更高的要求.

由于目标状态和定制约束常涉及到程序的执行细节,因此,路径的规划和选择、触发条件的推理和触发输入的求解也必须足够细致和准确.传统程序分析方法[11]也可以用于精准执行可达性分析,但并非全部适用.许多基于抽象的分析方法是路径不敏感的,难以适应精细化执行路径规划的要求.例如,模型检测和抽象解释等技术通过建立状态机、逻辑语言模型和数据流方程等抽象模型,从而简化了分析.由于抽象模型缺失了对部分执行细节的描述,其需要具体执行信息作为补充[12,13],否则会造成可达性误报.交互式定理证明工具依赖于手工增加的断言和注释,这种做法代价过大,因为精准执行可达性分析的应用场景大都是局部性和临时性的.而且,借助循环不变式来代替循环等做法也会带来误报.误报会严重影响可用性[14-16],对于只有一个触发目标的精准执行可达性分析问题而言,更是要极力避免的.另外,这些静态分析方法大都是针对代码位置非固定的程序属性的验证进行设计,同等地对待程序中的执行路径.但在精准执行可达性分析应用中,只关心与目标代码和目标状态相关的执行路径,应当尽早发现明显不可达的路径,并设计快速逼近触发目标的路径导航和探索机制.

为了生成可靠的触发输入,一般结合测试技术,以动态执行获得的运行时信息作为反馈,进一步修正静态分析的结果.目前,用于保障精准执行可达性分析精度的主要技术是双向符号分析.双向符号分析主要用到两个经典程序分析方法:正向符号执行[17]和逆向最弱前置条件推理[18].逆向最弱前置条件推理帮助发现一些可达目标状态的中间状态,正向的符号分析通过启发式的方式触探这些中间状态,以求根据交汇路径发现可达路径.符号分析对执行路径触发条件的刻画较为准确.但是,细粒度的路径分析会遇到著名的路径爆炸问题,这引发了对路径搜索策略的优化的探讨以及程序归纳的研究.目前,通过构造精简的抽象模型克服程序复杂性,并结合详细程序信息和符号分析指导精准执行可达性分析也有了一些进展[12,13],受限于篇幅,本文未展开介绍.

1.1.1 符号执行和约束求解

精准执行可达性分析与传统执行可达性分析对正向符号执行和约束求解的用法相同,这里仅简要加以介绍.符号执行[17]技术主要用于收集一条路径 p(用有序的代码位置序列表示)是可执行路径需要满足的条件,用路径约束 PC_p 表示.通过验证 PC_p 的可满足性(即进行约束求解),可以很方便地判断执行路径的可行性.具体地,符号执行用代数表达式来替换由输入初始化的变量的取值,并使用代数符号运算代替真实数值完成程序运算,通过记录分支条件以及与分支条件相关的变量计算关系,从而得到路径约束 PC_p.下面以一个简单的例子来说明这一计算过程.

例 1:求触发图 3 中阴影部分代码对应执行路径的初始变量赋值.

沿图 3 阴影部分代码对应的路径 p=⟨1,2,3,4,5,6⟩进行符号执行,计算 PC_p.图 3 右侧 3 列记录了抽象执行状态(l,σ)和路径约束 PC_p 的更新过程(这里,采用参考文献[10]中的记号).输入变量的初始值设置为一个代数符号,

例如 y 初始化为 y_0.图 3 对输入变量采用懒惰初始化的方式,即,输入变量在被使用时才初始化.每执行一步,就根据变量赋值更新符号状态(l,σ),在遇到分支条件时更新 PC_p(图 3 最右侧列加粗部分所示).最终得到的路径约束 PC_p 为 $y_0\neq 0\wedge y_0(y_0+1) \bmod 3\neq 0$,简化后,即 $y_0 \bmod 3=1$.也就是说,y 需为摸 3 余 1 的整数,才能触发该执行路径.

1	**if** (y!=0){	l=1	$\sigma=\{y\rightarrow y_0\}$	PC_p: $y_0\neq 0$
2	$x=y+1$;	l=2	$\sigma=\{y\rightarrow y_0, x\rightarrow y_0+1\}$	PC_p: $y_0\neq 0$
3	$z=x*y$;	l=3	$\sigma=\{y\rightarrow y_0, x\rightarrow y_0+1, z\rightarrow (y_0+1)y_0\}$	PC_p: $y_0\neq 0$
4	**if** (0==z%3)	l=4	$\sigma=\{y\rightarrow y_0, x\rightarrow y_0+1, z\rightarrow (y_0+1)y_0\}$	**PC_p: $y_0\neq 0\wedge y_0(y_0+1) \bmod 3\neq 0$**
5	**return** z;			
6	**return** −1;	l=6	$\sigma=\{y\rightarrow y_0, x\rightarrow y_0+1, z\rightarrow (y_0+1)y_0\}$	PC_p: $y_0\neq 0\wedge y_0(y_0+1) \bmod 3\neq 0$
7	}			

Fig.3 A symbolic execution procedure on a path

图 3 一条路径的符号执行过程

PC_p 是定长的约束公式,因此,判定路径 p 可行性的问题就是定长约束公式的可满足问题.与命题逻辑公式的可满足性问题 SAT(satisfiability)不同,其约束公式是结合了代数运算和数据结构知识作为背景理论的一阶逻辑公式,属于 SMT 问题[19].表 2 给出了一些在单路径执行可达性验证中常用理论的计算复杂性类,可以看出,许多常见的约束公式的求解是 NP 问题.关于 SMT 求解技术和常用求解器可以参考文献[19,20].

Table 2 Complexity type of constraint solving for some context theory

表 2 一些背景理论公式求解问题的计算复杂性类

理论	计算复杂性类
未解释函数的等式(EUF)	P
线性方程组、实数线性规划	P
差分逻辑	P
线性/混合整数规划	NP complete
位向量运算公式	NP complete
数组逻辑	NP complete
非线性约束	NP

1.1.2 最弱前置条件推理

精准执行可达性分析问题的目标状态对应的代码位置已知,因此,从目标状态开始进行逆向的触发条件推理便成为一种最为自然的方法.具体地,对于某条语句带来的程序状态迁移,可以根据后置条件(描述迁移后的状态变量的约束公式)和语句推测前置条件(描述迁移前的状态变量的约束公式).一般研究最弱前置条件(weakest precondition)的计算[18],满足如下两条的约束公式 pre 是语句 s 的关于后置条件 $post$ 的最弱前置条件.

a) 如果 s 执行前的程序状态满足 pre,则 s 执行后的程序状态满足 $post$;

b) 如果 s 执行前的程序状态满足$\neg pre$,则 s 执行后的程序状态满足$\neg post$.

从集合的角度来看,pre 与 $post$ 分别约束了输入集合 $I_{pre}\in I$ 和输出集合 $O_{post}\in O$,对于 $s:I\rightarrow O$,有 $I_{pre}=\{x\in I|s(x)\in O_{post}\}$.在不考虑别名的条件下,最弱前置条件算子一般用 $WP(s,post)$表示,其表示触发 s 的执行并造成后果 $post$(后置条件)的最低前提.在给定别名指派条件α时,则使用 $WP_\alpha(s,post)=\alpha\wedge WP(s,post)$表示[21],其中,$s$ 可以是一条指令,也可以是一个复杂的语句,还可以是一个基本块;后置条件 $post$ 是 s 执行后各相关变量的取值约束.例如,$WP_{(*x==*y)}(*x=*x+*y+1,*x>10)=2*(*x)+1>10$.在实际应用中,存在一些较难推理的语句,如复杂的循环、递归等,此时,一般辅以启发式的动态测试以确保分析准确.表 3 给出了在不考虑别名、循环和函数调用情形下的最弱前置条件推导规则,其中,“;”表示顺序执行.

Table 3 Rules for weakest precondition reasoning

表 3 最弱前置条件推算表

s	$WP(s,R)$
$x:=E$	$post[x:=E]$
$S;T$	$WP(S,WP(T,R))$
if B then S else T	$(B\wedge WP(S,R))\vee(\neg B\wedge WP(T,R))$

最弱前置条件计算还被用于检查死代码以及冗余断言造成的程序错误[22].

从功能上讲,逆向最弱前置条件推理已经可以完成可达性分析,但是仍然存在两个效率问题:首先,并不是所有的语句都需要推算最弱前置条件;其次,逆向分析遇到循环或递归时,直接展开分析会导致路径爆炸.解决前者涉及到路径剪枝和搜索策略的设计,而解决后者则主要靠对程序中重复运算的归纳.

1.1.3 双向分析和搜索优化

为了减少不必要的分析,逆向分析只需在与触发目标相关的数据流和控制流上进行.一般通过逆向程序切片[23,24]抽取影响目标状态的程序片段.在实际应用中,其与最弱前置条件计算和可满足性求解同时进行[25].另外,可达性分析并非需要沿着切片代码逐句进行,由表及里、由粗到精,懒惰地展开复杂函数,可以帮助快速剪除不可达分支.例如,设图 4 代码片段中 *mem_op* 函数在处理 *p_data* 时有可能引发缓冲区溢出,但是图 4 中的代码片段不会触发这个错误,因为 *p_data→type* 的值不匹配.此时,如果延迟分析第 4 行中的包含复杂运算的函数调用,则首先进行基本的控制条件的校验,就可以快速剪枝.

```
1   int recover(TData*p_data){
2       ...
3       if (p_data→type!=-1){
4           complex_operation(p_data);
5           mem_op(p_data,other);
6       }
7       ...
8   }
9   int handle_error(int type,char*message,TData*p_data){
10      ...
11      case MEM_ERR:
12          p_data→type=-1;
13          info(message);
14          return recover(p_data);
15      ...
16  }
```

Fig.4 An infeasible execution towards target caused by inconsistent control condition

图 4 控制流条件不一致导致的目标代码不可达实例

求解器无法处理的非线性运算、作用未知的外部 API 调用以及由于别名和共享变量造成的较大范围上下文依赖等场景阻碍了最弱前置条件的推算.对这些情况,使用正向分析作为补充,可适当缓解.这就需要合适的机制引导正向分析逼近目标状态,并避开已经发现的执行路径.随机[26]、自适应[27]或者基于搜索的技术[28-30]大都通过输入特征与执行特征之间的统计关联性来指导定向测试.这些方法的测试用例生成速度很快,但是对未探测代码区的分析不够精准.由于搜索空间巨大,其对触发条件相对苛刻的深层次执行路径的探测概率极低[31].相反,混合符号执行(EGT 或 concolic testing)[10]恰恰能做到对静态路径触发条件的细致分析,并较为准确地生成对应的输入用例.现有的精准执行可达性分析技术[9,32-34]选择距离目标状态最近的分支点进行条件取反和约束求解,从而逐步获得更加逼近目标的测试用例.常用的距离指标是最短执行距离[9,32-34],即,对分支点到目标状态的最短执行语句数的一种静态估计,其精度可根据需要进行调整.距离计算依赖于静态控制流图的构建.

当正向分析使用符号执行、逆向分析采用最弱前置条件计算时,这种方法被称为双向符号分析[9].双向符号分析一方面通过最弱前置条件计算,确定一些距离程序入口较近的中间目标;另一方面,在遇到难以推理的代码片段时(如非线性运算、未建模的 API 调用、复杂的别名指派等),采用正向混合符号执行技术,用真实执行得到的具体值代替符号值参与约束求解,从而启发式地触探中间目标.正向符号分析沿着距离中间目标由近及远的方式进行试探,当其与逆向分析交汇时,借助约束求解判定交汇路径的可行性.

由于执行路径空间巨大,正向路径搜索也需要优化.例如:记录导致路径不可达的 API 调用模式作为知识库,帮助后续分析快速剪枝[25];使用污点分析技术,识别各个分支条件中的变量是否受输入控制[35-37],只对与输入数据有关的分支条件进行约束求解.另外,为了有效地指导回溯,常借助约束求解器计算“不满足核(unsatisfiable core)”来识别使得约束公式不成立的关键部分,进而有选择性地回溯到特定分支[38].我国华中农业大学郭曦等

人[5]通过启发式的优先检查抵达目标代码位置的邻近路径,他们的实验结果显示,这种方法能够更快速地寻找到触发目标状态的执行路径.而对于触发同样位置的执行路径分布问题,则有待进一步探索.最近,在理论研究领域,已经有人提出保障双向分析交汇的算法[39],其值得在精准执行可达性分析中尝试.

1.1.4 程序归纳

程序归纳是将程序片段所表示的运算过程进行抽象和总结,进而得到简洁的功能模型或其计算问题的描述,从而借助问题自身的性质和特点简化程序分析和推理的归纳方法.程序归纳是编程的逆过程,也就是根据代码内容识别出它完成了什么功能或它在求解什么问题.在执行可达性分析中,识别代码片段的功能有助于合并同类路径、复用分析结果以提高分析效率;识别代码片段解决的数学问题有助于利用已知问题的性质,推理出触发(目标状态的)输入应该满足的条件,指导触发输入的构造.这里主要介绍功能归纳.

一般用"摘要(summary)"来描述一个代码片段的功能.将从相同的入口 s 到达相同的出口 t 的执行路径集 F 划分为 $n(n>0)$个互不相交的等价类 $F_1,F_2,\ldots,F_n$,以每一个执行路径等价类 $F_i(0\leqslant i\leqslant n)$的前置条件 pre_i(输入变量需满足的条件)和后置条件 $post_i$(输出变量关于输入变量的取值表达式)的二元组作为该类路径的简要功能描述,形成一个路径等价类的描述表,将这个描述表称为对路径集 F 的摘要,形式地用逻辑公式表达为

$$\phi_F=\bigvee_{i=1}^{n}(pre_i\wedge post_i) \tag{1}$$

其中,前置条件 pre_i 是计算路径形成的对输入变量的约束公式;后置条件 $post_i$ 是输出变量的约束,包含输出变量关于输入的赋值表达式.如果 F 是从 s 到 t 的所有执行路径组成的路径集,称其为从 s 到 t 的完全摘要;否则,称为从 s 到 t 的部分摘要.根据路径集 F 的摘要计算其最弱前置条件的公式为

$$WP(F,post)=\bigvee_{i=1}^{n}(pre_i\wedge post[post_i]) \tag{2}$$

上述摘要公式和最弱前置条件公式并未考虑别名,如需考虑不同的别名情况,则应对不同别名条件下的最弱前置条件进行析取,并且保证后续的最弱前置条件推算要沿着形成别名的路径进行.此时,逆向分析依赖于上文,需要与正向分析结合使用.该问题将在第 1.2.2 节加以讨论.

在进行逆向分析时,一些不可达路径类可以快速地从程序流程图上识别出来,因此并不需要计算某段代码的完全摘要,只需按照分析需要归纳可达路径片段簇的摘要即可.逆向分析多次经过的路径片段或路径片段簇的摘要,可以被记录下来方便重用.设某循环需要满足后置条件 $post$,则其沿着该循环的第 i 类路径(设 $post_i$ 为"$x:=E$")进行逆向推理得到的前置条件为 $pre_i\wedge post[x:=E]$.

下面以一个对线性查找 C 程序代码进行循环归纳的例子来说明逆向程序执行可达性分析中用到的对程序摘要的归纳过程.

例 2:现欲触发图 5 所示程序第 9 行的执行,求第 8 行的变量 input 需要满足的条件.

```
1   int search(int*array,int n,int key){
2       int i;
3       for (i=0; i<n && array[i]!=key; i++);
4       return (i<n?i:-1);
5   }
6   int foo(...){
7       ...
8       if (search(input,length,bad_value)≥0)
9           goto error;
10      ...
11  }
```

Fig.5 A code snippet that makes decision according to the result of search

图 5 一段根据查找结果进行分支决策的代码

图 5 中,*search*()函数实现了数组查找功能.查找到关键字 *key* 就返回 *array* 索引值;否则返回–1.而只有返回–1 才能触发第 9 行"goto *error*;".因此,满足条件的 *input* 的约束是

$$input!=NULL\wedge 0<length\leqslant len(input)\wedge\exists i\in\mathbb{N},0\leqslant i<length,input[i]=bad\_value,$$

其中,$len(input)$表示 $input$ 的长度($input$ 内存空间大小除以整型数宽度).

为了求得该约束,首先,建立 *search*()函数的程序流程图(如图 6 所示).

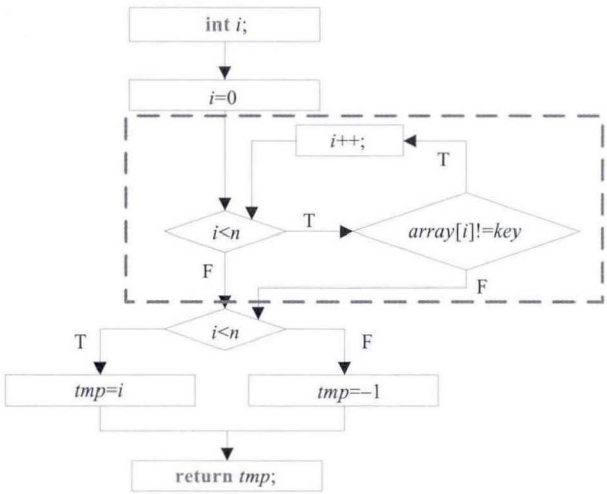

Fig.6 Program flow chart of function "*search*" in Example 2

图 6 例 2 中 *search* 函数的程序流程图

其次,识别流程图上的强连通分量以及其出入口,它们分别代表了循环体以及循环体的出入口.

然后,进行逆向最弱前置条件推理.图 5 中第 8 行的约束映射到图 6 中,即 $tmp\geqslant 0$.将第 4 行的三元运算符分支语句记为 s_4,其对应于图 6 中虚线框下方的内容,下标 4 代表行号.对其作最弱前置条件分析($i<n\wedge WP(tmp=i,tmp\geqslant 0)$)$\vee(i\geqslant n\wedge WP(tmp=-1,tmp\geqslant 0))$,得到 s_4 最弱前置条件为 pre_{s4}:$0\leqslant i<n$.分析程序流程图上的两个循环出口条件,只有后置条件为 $i<n\wedge array[i]==key$ 的出口满足.

有了出口条件,只需在循环体(强连通分量)中寻找出口条件 $i_k<n\wedge array[i_k]==key$ 的这类内部执行路径,对其进行摘要分析,进而计算该类路径的最弱前置条件.根据图 6,这类路径必经入口条件为 $i_0<n$,且循环体只有一条语句:"*i*++;",因此,它们只是在执行该语句的次数 k 上有所差异,其摘要可表示成 $\varphi_k\wedge\psi_k$ 的析取形式,并期待循环次数与 $\varphi_k\wedge\psi_k$ 具有特定的关系式.

最后,迭代这个循环过程,并记录每轮循环的前置条件和变量更新赋值关系,见表 4,其中,*LoopCount* 表示循环次数.

Table 4 Loop induction of Example 4

表 4 例 4 中循环的归纳

LoopCount=0		*LoopCount*=1		*LoopCount*=*k*−1		*LoopCount*=*k*	
前置条件	操作	前置条件	操作	前置条件	操作	前置条件	操作
$i_0<n$, $array[i_0]==key$	exit loop	$i_0<n$, $array[i_0]!=key$	$i_1=i_0+1$;	$i_0<n$, $array[i_0]!=key$	$i_1=i_0+1$;	$i_0<n$, $array[i_0]!=key$	$i_1=i_0+1$;
		$i_1<n$, $array[i_1]==key$	exit loop	…	…	…	…
				$i_{k-2}<n$, $array[i_{k-2}]!=key$	$i_{k-1}=i_{k-2}+1$;	$i_{k-2}<n$, $array[i_{k-2}]!=key$	$i_{k-1}=i_{k-2}+1$;
				$i_{k-1}<n$, $array[i_{k-1}]==key$	exit loop	$i_{k-1}<n$, $array[i_{k-1}]!=key$	$i_k=i_{k-1}+1$;
						$i_k<n$, $array[i_k]==key$	exit loop

按照循环规律,对 *LoopCount*=*k* 的前置条件和操作进行猜想,借助数学归纳法,考察循环次数增 1 带来的前置条件和操作的增量式影响,可证明表 4 中的公式.对 k=0 和 k>0 两种情况,收集每轮循环的前置条件以及操作

集合,将中间变量表示成初始值的运算式:

- 当 k=0 时,φ_0 为 $array[i_k]==key\wedge i_0<n$,$\psi_0$ 为 $i_{out}=i_0$;
- 当 $k\in\mathbb{Z}^+$时,φ_k 为 $array[i_0]!=key\wedge array[i_0+1]!=key\wedge\ldots\wedge array[i_0+k-1]!=key\wedge array[i_k]==key\wedge i_0+k<n$,$\psi_k$ 为 $i_{out}=i_k=i_0+k$.

根据前面的推导,语句 s_4 的最弱前置条件 $post_{s_4}$ 为 $0<i_k<n$,按照公式(2)计算图 5 第 3 行 for 循环(s_3)的前置条件:

$$\begin{aligned}WP(s_3,post_{s_3})&=WP(s_3,post_{s_4})\\&=(\varphi_0\wedge post_{s_4}[\psi_0])\vee(\vee_{k\in\mathbb{Z}^+}(\varphi_k\wedge post_{s_4}[\psi_k]))\\&=(0\leqslant i_0<n\wedge array[i_0]==key)\vee\\&\quad(k\in\mathbb{Z}^+\wedge((\forall j\in\{0,1,\ldots,k-1\},array[i_0+j]!=key)\wedge array[i_k]==key\wedge 0\leqslant i_0+k<n)).\end{aligned}$$

同样的方法,逆向分析至图 5 中代码第 8 行的 *search*()函数调用前,相当于将 i_0=0,n=*length*,*array*=*input*,*key*=*bad_value* 代入上式,有:

$$\begin{aligned}WP(s_8,post_{s_8})&=(0<length\wedge input[0]==bad\_value)\vee\\&\quad((\forall j\in\{0,1,\ldots,k-1\},input[j]!=bad\_value)\wedge input[k]==bad\_value\wedge 0<k<length).\end{aligned}$$

该公式等价于:$\exists k\in\mathbb{N},0\leqslant k<length,input[k]==key$.结合语言先验知识,增加两个约束:*input*!=*NULL* 和 *length*≤*len*(*input*),最终得到 *input*!=*NULL*∧*length*≤*len*(*input*)∧$\exists k\in\mathbb{N},0\leqslant k<length,input[k]==key$.

借助抽象解释理论中给出的不动点迭代方法可以求得循环不变式 $i<n\wedge array[i]!=key$,但其无法归纳出循环入口的 i_0 与输出的 i_k 的具体关系.若直接采用符号执行技术,则难以穷尽执行所有可达路径片段.即便恰好遇到一条满足图 6 所示可见代码的约束的执行路径,随着对第 7 行展开部分以及调用 func 的更高层代码推理的深入,对数组的附加条件越来越苛刻,这条偶然发现的路径片段未必能满足要求.当回溯时,盲目执行的效率低下.

在例 2 的分析中,$i_k=i_0+k$ 的约束和"∃"量词的归纳是两个关键归纳步骤.前者是输入变量、输出变量与循环次数的关系,通过"猜想"和"归纳法"证明得到;后者是对数组元素约束的归纳合并,需要与量词公式相关的理论作支撑.前面提到 SMT 求解器对于量词公式的支持还不够好,因此,类似问题的求解也有难度.事实上,"归纳猜想"与"演绎推理"是定理证明中的核心问题,现有的循环摘要[40−43]算法只能解决特定类型循环的识别和归纳,由于不可能穷尽所有的算法和编程模式,这种各个击破的思想仍存在局限性.

从另一个角度来看,分析 *search*()函数并识别其功能的过程实际上是一种代码模式识别.人工神经网络在模式匹配方面取得了很好的应用效果,近年来,该方法已经跨越图像、音视频和自然语言,在逻辑性较强的领域取得突破,如围棋比赛[44]等.对于自动程序摘要问题,其能力值得探索.

例 2 只是对线性查找的归纳,由于不同程序的应用领域各不相同,抽象出来的数学问题也各不相同,需有相应的形式语言和求解方法的支持.在操作系统等并发程序中,执行可达性分析需要考虑互斥、同步、锁、信号量、事件、中断等并发系统特性,该类程序的归纳更具挑战性[45,46].

1.2 分析难点

在精准执行可达性分析中,特有的分析难点是复杂场景约束的关联分析困难以及向固定代码位置引导动态测试的困难.传统的执行可达性分析中的常见程序分析难点,在精准执行可达性分析中仍然存在.所不同的是,精准执行可达性分析问题中的目标状态的代码位置约束和对执行路径定制化约束,为约减分析范围创造了条件.需要剥离出与目标状态有控制和数据依赖关系的部分,并在其上完成更细致的分析.表面上看分析范围缩小了,但对分析精度要求更高了.在传统的执行可达性分析中可以选择性回避的困难点,现在被重新提上分析议程.

1.2.1 复杂场景约束的关联分析

精准执行可达性分析问题中的定制化约束随应用而定.表 1 中的大部分应用的约束都比较简单且容易表达和判定,唯一复杂的是漏洞利用的构造问题.严格来讲,它属于一个编程问题,许多漏洞利用技术都是图灵完

备的,也就是说,一个漏洞利用可以完成计算机可做的任何事情.具体漏洞的利用所涉及到的软件系统和应用领域知识也非常丰富,可以从运行时内存分布等底层信息直到高层的应用逻辑.如何将不同抽象层次上的约束关联起来并有效地表示或建模,以有效地辅助漏洞利用构造是一个挑战.这一问题在内存相关漏洞的利用方面尤为明显.

例如,对缓冲区溢出构造漏洞利用,为了劫持数据流,需要考虑输入中溢出的字节与返回地址的关联.为了绕过数据执行保护(DEP),需要考虑栈中填充的一连串跳转地址与运行时加载到内存中的代码片段(return Gadget)的关联(ROP 攻击[47−51]);为了绕过控制流完整性保护(CFI)[52−57],需要将溢出变量的取值与被缝合变量的地址相关联,还需要将一连串被错位的数据流和指针赋值与最终劫持到的目标数据相关联(DOP[8,58]攻击).

在现有的漏洞自动利用工作中,AEG[6,7]针对源码可见程序的栈缓冲区溢出,按照与人工调试的相同的自动流程识别数据填充返回地址,其通过查找可执行文件中的符号表来定位源码中特定变量或函数的地址.FlowStitch[8]专门定义了 3 类可缝合数据,并构造了 2D-DFG 来表示不同数据流上变量的活跃关系,以辅助被劫持指针和待缝合数据的配对.采用 ROP 方式进行栈溢出漏洞利用的 Q 系统[59]则针对从 Gadget 到攻击向量的组装过程专门设计了一套编程语言 QooL.目前,自动内存漏洞利用研究还不够完善,对释放后重用、数据竞争等问题导致的漏洞自动利用缺乏研究,还未能形成较为系统的内存数据关联和建模方法论.

1.2.2 数据结构识别、分析与归纳

在存在别名的情况下,最弱前置条件计算依赖于上下文.例如在图 7 所示的代码中,虽然只有两条执行路径,但是第 6 行的 *size* 变量与第 1 行~第 5 行中的任何一行代码都有着依赖关系.如果要对 *size* 进行推断,则需要综合考虑第 1 行~第 5 行指针别名信息才能做出决定.也就是说,首先识别出这种依赖结构和别名,才能进行准确的逆向分析.

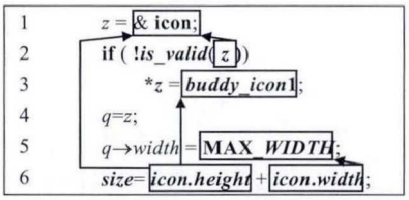

Fig.7 A C code example that shows indirect data dependency

图 7 指针带来的间接数据依赖 C 代码示例

适当的正向符号执行可以缓解图 7 所示的逆向依赖分析问题,但仍需先确定正向分析的起点或分析范围,否则又会陷入盲目的正向穷尽搜索.由于指针可以跨越不同的函数传播,最坏情况下,需要综合分析程序的所有代码.全局变量和锁也会产生类似的依赖问题.另外,上下文依赖的指针能够形成上下文依赖的控制流迁移,如动态函数指针和二进制程序中的间接跳转[2],这在基于事件和消息的程序中尤为常见.上下文敏感(context-sensitive)的指针分析是不可判定问题[60],一般通过限定正向分析跨越的函数层数或对象层数来平衡计算的精度与速度[61].

精准执行可达性分析只关心影响目标状态的别名,且其不仅需要知道准确的指向关系,还需要知道别名信息的有效作用域以及产生别名的路径条件.经典的别名分析方法[62,63]是面向全局指向关系的计算,而且是一种可能性分析,其存在别名误报.需求驱动的指向分析[64]专为部分变量的别名计算而设计,其理论基础是上下文无关文法可达性.该类方法中,对指向关系进行摘要的做法[65,68,69]可以减少可达性推断中的重复计算.现有的方法[65−68]能够做到上下文敏感和字段敏感,但很难做到路径敏感.可达性推理常涉及堆数据结构在计算过程中的高级性质,如判定链表是否存在环、二叉树是否平衡等.这涉及到形态分析[70−73],该类问题也是 NP 问题.精准执行可达性分析的按需分析模式给形态分析提出了新的需求,旧有的全局形态属性验证技术不再适用.

1.2.3 算法的识别、分析与归纳

逆向推算最弱前置条件的过程中可能会遇到一些带有循环/递归的复杂计算过程,这些计算过程会导致执行路径数量的爆炸式增长.识别这一过程所用的算法并确定在不同输入条件下的功能和计算性质,能够为后续

问题的求解提供帮助.现有的研究还很难做到这一点,目前,这方面的研究集中在对基本程序结构的摘要上.

从不同的语法结构来划分,程序摘要包括函数摘要、循环摘要、分支摘要、语句块摘要等.其核心思想是:通过记录摘要,重复利用之前的分析结论以提高性能.微软公司研究员 Godefroid 最早提出函数摘要[74]以及组合符号执行[75],帮助符号执行中函数分析进行结果复用.卡内基·梅隆大学的 Avgerinos 提出了对 if-else 嵌套语句的分支合并策略(分支语句摘要),以减少符号执行对分支路径的探索次数[76].中国科学院软件研究所的易秋萍设计了一种基于最弱前置条件的分支摘要的方法[77],避免了 KLEE[78]在产生(fork)符号状态时以相同的状态进入公共子分支.循环摘要技术[40-43]主要依靠循环中线性变化的归纳变量对变量的计算结果进行推断,按照摘要完整性,其可分为完全循环摘要[40,41]以及部分循环摘要[42,43].现有的摘要技术只能处理针对整数加减和字符比较等基本数据类型运算的周期性循环,这限制了其应用范围.对非周期线性变化的循环(如二分查找等)的摘要问题以及含有非线性运算的循环摘要问题还未能解决.另外,循环不变式也可以用来代替循环,帮助排除不可达路径.循环不变式可以借助抽象解释中的不动点算法计算[79-82],但并非所有的循环都有不变式.

传统的可达性分析关心整体代码的覆盖率,因而需要完整的摘要来划分等价类,以减少同类路径的重复测试.但是精准执行可达性分析只关心如何触发位置确定的单一目标状态,只需摘要影响目标状态的局部切片中的复杂计算即可.目前,专门针对最弱前置条件计算的程序摘要技术还较少.

程序摘要和组合分析技术仍处于研究起步阶段,还有很多基本算法和有规律性的计算结构的摘要问题有待解决,程序片段摘要的组合归纳还没有形成完善的理论.对于所有的算法是否都存在相对简明的摘要,或哪类程序或算法存在简明摘要的问题,尚未给出理论解释.另外,程序摘要只是程序执行路径簇的功能识别和表示法.程序归纳还包括识别该路径簇解决的程序问题,这类更高层次的归纳分析还有待探索.

1.2.4 外部接口的建模

软件并非独立运行,软件与软件、软件与硬件之间存在着依赖关系.通常,一款软件的源代码并不包含其所用到的外部库的程序逻辑,发布的应用程序可执行文件中也并不包含所使用的动态链接库以及系统调用的内部实现逻辑.操作系统、基础库、中间件、驱动程序等 API 接口繁多,种类复杂,与其交互的应用程序和硬件也种类各异,即便是应用程序,网络通信、图形以及命令行上的交互接口也形式各异.源代码或二进制程序符号分析如果不能准确地识别这些 API 接口,则难以准确地推理触发路径和触发输入.

现有的针对源码的分析技术大都针对特定的 API 设计识别方法,构造特定的 API 分析模型,因此只能分析特定交互形式的程序.例如,KLEE 符号执行引擎对μCLibc 库进行了建模,其专门针对命令行程序进行了测试接口设计;商业反汇编工具 IDA Pro 内建 API 的指令序列模式来识别外部 API 调用信息,其所分析的二进制程序接口也是常用的系统调用和基础库.借助 QEMU 等系统模拟器可以记录全系统的执行轨迹,但目前的研究[83-86]还都未能利用待分析程序源代码或可执行文件中的静态信息进行引导,这使得动态分析陷入不相关代码的约束求解和条件翻转试探中,极大地降低了分析效率.

无论是对源代码还是对二进制可执行文件,外部 API 识别和摘要方法都依靠手工一对一设计.由于并非所有的 API 功能都存在简洁摘要,而且软件库时有更新,每月都会有新的库诞生,老的库被弃用(deprecated),手工 API 分析以及建模和维护成本高昂,且并非一劳永逸.分析收效又没有保障,未能吸引足够的研究者关注.这导致执行可达性研究局限于命令行和文件读写应用程序以及相对简单的网络应用程序,对图形界面的程序、驱动程序、操作系统、库、中间件缺乏研究.

1.2.5 程序执行的不确定性

在并行、并发和分布式程序以及带有随机数计算的程序中,同样地输入 $\boldsymbol{X}$ 可以触发多个计算过程副本(p_i),但其中可能只有个别几个能够触发给定的目标状态 t.程序、进程和线程之间同步顺序受同步原语控制.共享内存结构下,内存访问的距离(局部性)也会影响程序的执行顺序.操作系统一般都含有随机事件收集机制,以收集到的随机事件作为随机数种子产生随机数,随机事件包括系统时间、CPU 调度、电容、电压等.可能的同步次序随同步操作呈指数级增长,随机数难以复现,这给执行可达性分析带来了额外的挑战.

并发程序的分析和测试技术大都是在监控器和代码插装技术的帮助下进行穷尽搜索.并发程序可达性测

试[45,87-90]基于前缀和同步序列枚举方式辅助生成测试用例,其能保证每种同步序列仅执行一次,并且不需要记录历史同步序列.但是,同步序列随同步操作呈指数级增长,难以穷尽.并发程序的模型检测也遇到了类似的状态爆炸问题[91,92].在CHESS[93]和CTrigger[94]以及基于随机测试[95]的并发错误检测系统中,常根据目标并发错误的特点设计相应的试探策略,例如插入抢占事件、检查事先定义的错误访问模式等,从而提高了实用性.

对于并发程序精准执行可达性分析问题,ESD[32]借助符号执行引导错误复现,其将同步顺序设置成符号变量枚举同步原语的交叠次序,通过以最短执行距离优先的策略来合成执行路径.Weeratunge 等人[96]通过比对在错误点处正常执行和错误执行"core dump"引导调度策略的还原,其在CHESS的插入抢占(inject preemptions)式搜索的基础上进行改进,能够在单核上复现多核执行时的错误.香港中文大学的Huang等人[97]指出,用于错误复现的执行计算存在冗余性,并给出了一种冗余评判准则来约减用于重现错误的执行记录.

对随机数问题,目前没有较好的解决办法,一般用符号变量或真实值来代替[2].

1.2.6 问题求解

为了有针对性地分析路径而不是随机测试,精准执行可达性分析问题最终仍需借助SMT求解器进行约束求解.但是,由于问题本身是NP问题,许多约束公式都无法在有限的时间内给出解答.路径归纳抽象出的数学问题常带有量词或与组合问题有关,而且一些常见运算,如哈希、加密等过程是不可逆的,这给求解带来了困难.所处应用领域不同以及所驱动的硬件不同,被分析的程序对象、抽象出来的领域数学问题也各不相同,需要专用定理证明器的支持.如何集成并高效地组织各类理论求解方法,仍然是多路径执行可达性搜索中的核心难题.

复杂的功能函数通过调用简单的功能函数来实现.因此,研究基本算法和数据结构的推理和求解问题有助于自底向上地组合分析.对此,中国科学技术大学张昱等人的工作[81,82]值得参考.其针对非循环和循环单/双向链表以及表示二叉树的二叉链表构造了一套形状图逻辑,并给出了形状图逻辑下循环不变式的迭代自动计算方法以及形状图理论和整数理论的组合判定方法.另一方面,程序有自身的规律,例如哈希函数常用于对象或者字符串比较,很多看似复杂的循环和递归就是在做批量的数据复制、修改和格式变换.识别这些可预测的重复计算模式,有利于辅助程序自动推理.机器学习和深度神经网络等基于统计的人工智能技术能否辅助解决代码自动识别和推理问题有待探索,南京大学的李鑫等人利用机器学习技术进行约束求解加速的研究[98],值得参考.

2 精准执行可达性分析应用

精准执行可达性分析的应用主要有定向测试、静态分析结果核验、错误复现、漏洞利用构造等.表5总结了近年来发表的该领域较为典型的应用研究及其发表时间,将这些应用研究中的被分析程序进行了总结,见表6,其中,第2列和第3列分别是应用研究在本文参考文献中的引用编号和应用研究所分析的程序类型.

Table 5 Publish time-table for application research of the precise reachability analysis

表5 精准执行可达性分析应用研究发表时间表

应用	2010	2011	2012	2013	2014	2015	2016
定向测试		[33,100]		[13,34]	[99]		[9]
静态分析结果核验		[101]	[25]				[102,103]
错误复现	[32,96]		[104]			[37]	
漏洞利用的构造		[5,59]	[105]			[8]	[58]

Table 6 Target programs in application research of the precise reachability analysis

表6 精准执行可达性分析应用研究中的被分析程序

应用	研究	程序类型	被分析程序
定向测试	[9]	C	cdaudio, diskperf, floppy, kbfiltr, ssh client, ssh server, space
	[33]	C	Coreutils (ptx, seq, paste, mknod, mkfifo, mkdir)
	[34]	C	findutils, diffutils, binutils
	[99]	C	人工构造(hard-loop, dart, unreach, slicing, narrow, easy-loop, trityp)
	[100]	Java	Wheel Brake System, Program that models NASA's On-board Abort Executive, Altitude Switch Application
	[13]	Java (Android)	TippyTipper, ConnectBot, Munchlife, OpenManager, DieDroid

Table 6 Target programs in application research of the precise reachability analysis (Contiuned)

表 6 精准执行可达性分析应用研究中的被分析程序(续)

应用	研究	程序类型	被分析程序
静态分析结果的核验	[25]	Java	apache-ant, batik, tomcat
	[101]	C	BIND, Sendmail, WU-ftpd
	[102]	C	Coreutils (date, mkdir, tac)
	[103]	C/C++	gzip, tftp-hpa, net-tool, inspired, udisks, tiff, freetype,firefox
错误复现	[32]	C	SQLite, HawkNL, coreutils (paste, mknod, mkfifo, mkdir, tac)
	[96]	C	apache, mysql
	[104]	C	sed, grep, gzip, ncompress, polymorph, aeon, glftpd, htget, socat, tipxd, aspell, exim, rsync, xmail
	[37]	Binary	Adobe Reader, Windows Media Player, Real Player, Orbital Viewer, Music Animation Machine Player
漏洞利用的构造	[5]	C	aeon, iwconfig, glftpd, ncompress, htget, expect, socat, tipxd, aspell, exim, xserver, rsync, xmail
	[59]	C	Free CD to MP3 Converter, FatPlayer, A-PDF Converter, MP3 CD Converter Pro, rsync, opendchub, gv, proftpd
	[105]	C	Linux: A2ps, Aeon, Aspell, Atphttpd, FreeRadius, GhostScript, Glftpd, Gnugol, Htget, Htpasswd, Iwconfig, Mbse-bbs, nCompress, OrzHttpd, PSUtils, Rsync, SharUtils, Socat, Squirrel Mail, Tipxd, xGalaga, Xtokkaetama, Windows: Coolplayer, Destiny, Dizzy, GAlan, GSPlayer, Muse, Soritong
	[8]	C	nginx, coreutils (sudo), sshd, WU-ftpd, orzhttpd, null httpd, ghttpd
	[58]	C	bitcoind, musl libc+ping, Wireshark, nginx, coreutils (sudo), mcrypt, ProFTPD, sshd, WU-ftpd

从表 6 中可以看出:除专门对 Android 事件处理的研究[13]外,其余的应用研究选用的大都是采用命令行、文件、网络协议这 3 类交互接口的应用程序.这些程序与外部软硬件的交互方式和依赖关系比较简单,大都是开源的中小型应用程序且程序功能较为单一.被分析程序中多是 C/Java 程序,对二进制及其他语言的研究较少.

2.1 定向测试

定向测试指的是有针对性测试特定的代码区或功能模块.美国马里兰大学的 Ma 在 2011 年最早提出了定向符号执行的概念[33],给出了一种基于函数内部符号执行和沿调用链拼接的定向路径探索方法.后续学者们不断地改进和发展了以引导符号执行做定向路径探测的思路.例如:为了发现补丁附近的错误,KATCH[34]使用一种最短距离优先的符号执行策略对补丁位置进行专门的测试,从目标语句逆向地计算最弱前置条件估计语句之间的可达路径和执行距离,以距离目标语句短路径优先的方式进行符号执行,将符号执行引导至打补丁的位置.为了在调试和回归测试中覆盖特定的程序分支,Dinges 提出了一种逆向的符号执行技术[99]以生成触发特定代码行的测试用例,同样借助最弱前置条件计算,其在遇到难解的循环和包含复杂运算的函数调用时,对路径片段加以包装,形成一个可测试的片段程序,通过启发式地试探不同的输入参数值对该片段进行可达性探测.这种方法虽然损失了一些可达路径,也没有理论保障,但其好处是将难解的长路径的可达性问题分段、拆分,分而治之.与补丁代码测试类似的一个应用是版本间的差异路径测试[100]以及补丁交叉验证[106],它们均使用了符号执行和约束求解来分析路径的执行可达性.最近,开始有人关注基于 Android 事件序列的定向测试生成技术,丹麦奥胡斯大学的 Jensen 等人[13]将基于模型的测试技术与定向符号执行中沿着调用链进行路径拼接[33]的思想相结合,构造了一种针对事件处理函数确定代码行构造触发事件序列的方法.他们通过在基于事件的状态迁移图上区分锚事件(anchor)、连通事件(connector),有效地减少了冗余计算.

2.2 静态分析结果核验

为了核验静态分析结果,一种最直接的做法就是沿着函数调用图,从粗到精地进行路径验证.IBM 实验室的 Sinha 借助这种方法验证了 Java 程序的缺陷[25],其沿着调用图逆向探索调用函数(caller),在函数内部正向搜索和展开被调函数(callee),在函数内部的正向分析帮助快速排除不可达路径,以目标为中心使用回溯法检查执行可达性,并从失败的路径探索中总结 caller/callee 不变式(如果调用序列满足该式,则缺陷一定是不可达的).通过沿着调用图搜索、失败、学习、回溯这 4 个主要步骤,以类似 DPLL 算法的方式去寻找可达缺陷位置的执行路

径.该方法专注于逆向推理中的数据依赖,因此可以减少不必要的函数展开.

加州大学伯克利分校的 Babic[101]借助下推自动机引导符号执行到静态分析发现的缺陷位置,进而核验缺陷的可触发性.该方法以循环次数的指数分布寻找循环模式,进而完成循环内部的启发式搜索,并按照数据流切片和最短路径引导对非循环路径启发式试探.通过识别控制流图上的强连通分量(SCC),能够很好地分离带有循环和非循环部分,从而分别进行分析.滑铁卢大学的 Parvez 使用定向路径搜索验证静态分析得到的缺陷路径,借助控制流上的关键边(后主导关系)进行不可达路径剪枝,并保证只在到达目标函数的强连通分量中进行探索[102].其基于 S2E[83,84]开发,同样面向二进制程序.

南京大学高凤娟等人设计了一套对 Fortify 报告的缓冲区溢出进行复查和修复的系统 BovInspector[103],该系统使用静态分析工具 Fortify 提供的路径信息指导 KLEE 进行符号执行路径探测.在遇到一条分支在 Fortify 报告的缓冲区溢出路径内、而另一条分支在 Fortify 报告的缓冲区溢出触发路径外时,对后者进行剪枝;对于其他情况则不进行剪枝.对于缓冲区溢出漏洞的执行可达性,该方法根据约束求解的结果给出真溢出(有解)、误报(无解)、未知(超时)这 3 种判定,其设计了 3 种自动修复策略对真溢出的情况进行修复.由于该方法在未抵达 Fortify 报告的分支状态时使用 KLEE 自带的启发式穷尽搜索,其性能与 KATCH[34]等使用的最短路径优先的探索策略的系统相比有待进一步研究.

2.3 错误复现

错误复现主要研究如何根据给定的错误报告复现错误,从而帮助调试和修复.为了复现错误报告中的多线程错误(给定线程栈),ESD[32]针对多线程程序的源码设计了符号执行和调度策略,合成错误报告中的路径.ESD 在分析路径的执行可达性时同样使用了符号执行和约束求解的方法,其通过识别控制流上的关键边(critical edge)定位必经的路径分支,并计算分支条件中变量的数据依赖(到达定值分析,reaching definition analysis),从而确定待触发的中间目标,用关键边和中间目标来引导符号执行,并采用最短距离优先的启发式策略.通过在访问临界区时 fork 线程的交替状态,进而探索所有的线程交替可能,以求发现错误报告中的死锁和数据竞争.Bugredux[104]系统通过类似的技术合成单线程程序崩溃路径,其研究单线程程序的错误可达路径搜索.

Hercules[37]系统面向二进制程序进行错误复现,其首先借助 IDA Pro 反汇编,识别出二进制程序的基本结构、绘制控制流图,并通过记录执行轨迹(trace)在测试组件中的输入集合中筛选出能够触发目标崩溃模块执行的文件.其次,通过污点分析识别出触发崩溃模块执行的关键输入,并借助记录下的执行轨迹修正控制流上的间接跳转,形成更加准确的控制流图.然后,对目标崩溃模块入口语句到崩溃点的执行路径进行符号分析,形成模块入口的执行摘要.最后,将关键输入的内容符号化,并保留其余不相关的文件部分的具体值,进行混合符号执行;通过约束求解,修正不满足入口摘要的变量值,进而寻找触发崩溃的输入.该方法与 ESD 非常相似,同样借助约束求解判定路径的执行可达性,通过构造中间目标减少搜索空间爆炸.与 ESD 面向源代码分析不同,其需要借助执行轨迹(trace)额外修正 IDA Pro 得到的静态控制流图上的间接跳转,而且 Hercules 针对非并发程序设计.

Weeratunge 等人[96]设计了一种根据并发错误的 core dump 和触发输入来还原调度策略的方法,通过比对在错误点处正常执行和错误执行 core dump 引导调度策略的还原,其在 CHESS 的插入抢占(inject preemptions)式搜索的基础上加入了执行索引来表示一个唯一的执行点,进而通过比较执行索引来引导调度策略的修正,从而逐渐逼近原错误,能够在单核上复现多核执行时的错误.

2.4 漏洞利用构造

软件漏洞的利用过程也是一个可达路径的构造过程.针对栈缓冲区溢出和格式化字符串漏洞,卡内基·梅隆大学的 Avgerinos 等人提出了一种自动构造控制流劫持攻击的方法(AEG[6,7]),其使用符号执行和约束求解来验证漏洞触发的可行性.与传统符号执行路径探测所不同,其根据栈溢出漏洞设计了关于输入的长度和内容的启发式构造策略,将该类约束作为预置条件进行符号执行(postconditioned symbolic execution),从而更有针对性地劫持控制流并实现攻击载荷的执行.Mayhem 系统[105]将 AEG 的约束建模方式挪用到了二进制程序的漏洞自动利用中,并增加了内存索引、混合在线和离线符号执行等加速方法.这两类方法没有考虑到内存防护机制,无法

绕过栈保护(DEP)、地址随机化(ASLR)以及控制流完整性检测(CFI)[52−57]等.针对栈缓冲区溢出的控制流劫持ROP攻击可以绕过DEP和ASLR,同样处于卡内基·梅隆大学的Schwartz也开发了相应的自动利用系统Q[59],其收集目标程序中的Gadget,并通过面向Gadget的编程语言QooL自动构建ROP攻击[47−51].

新加坡国立大学的胡宏等人构造了一种数据流劫持攻击方法,其能够在不改变控制流的前提下,通过修改指针和数据指向,进而触发权限提升和实现内存数据窃取.他们还设计了一种数据流自动缝合技术FlowStitch[8],FlowStitch通过构造2D-DFG来寻找活跃的可用数据和指针,通过预先识别安全敏感数据来缩小缝合数据的选择范围.其能够识别的敏感数据包括系统调用参数、被配置文件污染的变量以及栈保护随机数(canary).FlowStitch将路径约束、影响约束和控制流完整性约束三者的合取式交给Z3求解器求解,进而获得触发数据流攻击的输入数据.胡宏等人在后续的研究中[58]证明了这种攻击的图灵完备性,也就是说,数据流攻击可以完成图灵机能完成的任何事情.

3 存在的问题和挑战

精准执行可达性分析应用大都采用双向符号分析和约束求解技术,这主要是因为其能够准确模拟一条路径的执行行为,进而保证了执行可达性分析结论的准确性.但是,该领域仍存在许多待解决的问题.

(1) 复杂约束的模型构建和关联分析缺乏有效的分析框架和理论指导.已有的自动漏洞利用方案大都针对栈缓冲区溢出,存在一定的局限性,且相应的内存建模技术还不成体系,更多的案例研究有待开展;

(2) 路径探测策略缺乏理论指导.对于定向符号执行中常用的最短路径优先等启发式策略,如何进行理论上的定量评估和分析以及是否存在更高效的探测策略尚未明确.对固定代码位置目标可达路径模式研究还较少,对此,模糊测试中关于路径迁移关系的研究思路[31,107]值得借鉴;

(3) 数据结构的识别和推理能力不足.路径敏感的指针别名分析问题、数据结构高级性质的逆向推理和归纳问题以及对全局变量操作的逆向推理问题等,都是亟待解决的;

(4) 动态控制流迁移逆向分析存在挑战.源代码中的动态控制流迁移表现为函数指针的动态赋值、类的动态实例化、反射以及同步操作等;在二进制分析中的动态控制流迁移则表现为间接跳转和中断.该困难集中显现在基于事件和消息的系统以及含有回调机制的系统中,包括操作系统和GUI程序;

(5) 路径爆炸问题仍然是多路径执行可达性搜索的最大障碍,现有的路径归纳和组合分析方法还不完善.现有的循环路径摘要算法局限在处理简单的周期性字符处理和整数运算上,无法处理指针数据结构以及带有函数调用的循环.对面向逆向分析的按需路径摘要技术以及综合高层模式和执行细节的分析技术还有待探索;

(6) 面向程序分析的SMT求解技术还存在提升空间.由于SMT公式的可满足问题是将执行可达性分析问题按照路径片段拆解得到的,而现有的技术所使用的求解器大都是相对通用的,因此存在进一步优化的可能.程序路径的经验知识的重要性已经在用机器学习改进求解技术的研究中有所显现[98];

(7) 人工API建模与验证的成本高于执行可达性分析带来的回报,这阻碍了对不同类型程序的扩展研究.目前,精准执行可达性分析应用研究的自选测试集中大都是一些交互简单的文件处理和网络服务程序,缺乏对操作系统内核、设备驱动、中间件、带有图形界面的程序以及Android等新平台上的精准执行可达性分析的研究;

(8) 并发程序和含有随机数程序的精准执行可达性分析仍存在调度策略组合爆炸和执行路径重现方面的挑战.

4 总结与展望

精准执行可达性分析是软件测试、软件调试和漏洞挖掘实践中的核心技术.本文从理论和应用两方面对精准执行可达性分析相关研究进行了跟踪、归纳和总结,对精准执行可达性分析方法和应用进行了分析和分类.虽然精准执行可达性分析在中小应用程序的分析方面已经取得了一些成功,但是对到达目标状态的执行路径

的一般性分布特点还缺乏理论和经验研究的支持,引导不同程序执行到特定位置的方法体系还未形成,复杂场景约束的建模和关联求解也还未形成系统化的方法.除这些特有的问题外,精准执行可达性分析仍然面临诸如搜索空间爆炸、难解的 SMT 问题、接口建模和并发分析等程序分析的挑战.由于精准执行可达性分析以确定位置的目标状态为导向,灵活利用不同抽象层和逻辑切面约束信息来设计更加专注的试探策略,以及通过抽象目标状态特性,形成有针对性的聚焦和精准推理,仍然是未来研究的主题.

在复杂约束的关联分析方面,还缺乏对许多内存漏洞的利用自动构造问题的研究.诸如堆溢出、释放后重用、使用时未初始化以及与数据竞争和锁等并发控制相关的漏洞利用自动构造问题还有待尝试.除漏洞利用构造问题外,更多含有复杂约束的应用场景有待发现,可考虑在兼容、版本变更、缺陷自动修复等需要进行多因素考量的场景中挖掘类似的问题.另外,寻找触发同一缺陷的差异化执行路径[26]也是一个有价值的研究主题.在一些非合作场景中,为了降低试探失败的风险,对输入的成功率有了更高的要求,对非常态软件(代码混淆、加密)的精准执行可达性分析值得扩展.

在解决路径爆炸问题上,归纳和摘要对重复计算的预测和等价类划分是有效的.多元素数据结构常用操作的摘要和与他们相关的最弱前置条件快速推算方法值得研究.例如,线性数据结构的处理就有很多可归纳的操作,包含查找、排序、筛选、截取、拼接、复制、逆序等.探索高层逻辑和系统机制上以非公式形态出现的最弱前置条件表达[13],对执行可达性分析有重要的研究意义.除了从模型和机制上抽象和简化问题,程序分析技术也需要大数据和分布式计算来强化自身,毕竟单台计算机的处理能力是有限的.当然,分布式程序上的各类调试和测试问题也给精准执行可达性分析带来了新机遇.

对与外部软硬件交互复杂的程序,如操作系统内核、设备驱动、中间件、图形界面程序和分布式程序等的分析需要较强的领域知识作辅助以保障准确性.在对执行路径规律不了解的情况下,案例研究和经验统计研究是很好的突破口.此外,软件执行机制的高层抽象模型需要与执行路径细节分析有机地结合起来[13],通过分析 API 文档来抽取高层逻辑模型以及最弱前置条件相关信息[108]值得进一步探索.

对于运行时决定的间接控制流跳转问题,虽然在二进制程序中没有较好的追溯办法,但在源代码中,通过细致的分析还是可以找到若干可能的函数调用点的.由于目标状态单一且代码位置明确,指向分析不再是全局的而是按需进行的.以 C 语言为例,识别这些控制流除了要展开宏以及识别动态和静态加载的库函数,还需要对自动化编译脚本以及配置文件等进行分析.

最后,与其他学科和理论相结合可以开启更好的研究视角.例如,机器学习技术可能对基本数据结构和算法的识别以及自动程序摘要有帮助,特殊编程模式下的数据依赖定位和约束求解[98]中也可能蕴含着丰富的统计规律.使用了神经网络等“黑盒子”技术的软件系统也给执行可达性分析带来了新的挑战.

References:

[1] Rice HG. Classes of recursively enumerable sets and their decision problems. Trans. of the American Mathematical Society, 1953, 74(2):358–366. [doi: 10.2307/1990888]

[2] Shoshitaishvili Y, Wang R, Salls C, Stephens N, Polino M, Dutcher A, Grosen J, Feng S, Hauser C, Kruegel C, Vigna G. SOK: (state of) The art of war: Offensive techniques in binary analysis. In: Proc. of the 2016 IEEE Symp. on Security and Privacy (S&P). Piscataway: IEEE Press, 2016. 138–157. [doi: 10.1109/SP.2016.17]

[3] Christakis M, Müller P, Wüstholz V. Guiding dynamic symbolic execution toward unverified program executions. In: Proc. of the 38th Int'l Conf. on Software Engineering. New York: ACM Press, 2016. 144–155. [doi: 10.1109/SP.2012.31]

[4] Haller I, Slowinska A, Neugschwandtner M, Bos H. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In: King S, ed. Proc. of the 22nd USENIX Security Symp. Berkeley: USENIX Association, 2013. 49–64.

[5] Guo X, Wang P, Wang JY, Zhang HG. Program multiple execution paths verification based on *k* proximity weakest precondition. Chinese Journal of Computers, 2015,38(11):2203–2214 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2015.02203]

[6] Avgerinos T, Sang KC, Hao BLT, Brumley D. AEG: Automatic exploit generation. In: Proc. of the Network and Distributed System Security Symp. San Diego: Internet Society, 2011. [doi: 10.1145/2560217.2560219]

[7] Avgerinos T, Cha SK, Rebert A, Schwartz JE, Woo M, Brumley D. Automatic exploit generation. Communications of the ACM, 2014,57(2):74–84. [doi: 10.1145/2560217.2560219]

[8] Hu H, Chua ZL, Adrian S, Saxena P, Liang Z. Automatic generation of data-oriented exploits. In: Jung J, Holz T, eds. Proc. of the 24th USENIX Security Symp. Berkeley: USENIX Association, 2015. 177–192.

[9] Baluda M, Denaro G, Pezze M. Bidirectional symbolic analysis for effective branch testing. IEEE Trans. on Software Engineering, 2016,42(5):403–426. [doi: 10.1109/TSE.2015.2490067]

[10] Cadar C, Sen K. Symbolic execution for software testing: Three decades later. Communications of the ACM, 2013,56(2):82–90. [doi: 10.1145/2408776.2408795]

[11] Mei H, Wang QX, Zhang L, Wang J. Software analysis: A road map. Chinese Journal of Computers, 2009,32(9):1697–1710 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2009.01697]

[12] Nayrolles M, Hamoulhadj A, Tahar S, Larsson A. JCHARMING: A bug reproduction approach using crash traces and directed model checking. In: Proc. of the 22nd IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). Piscataway: IEEE Press, 2015. 101–110. [doi: 10.1109/SANER.2015.7081820]

[13] Jensen CS, Prasad MR. Automated testing with targeted event sequence generation. In: Proc. of the 2013 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2013. 67–77. [doi: 10.1145/2483760.2483777]

[14] Johnson B, Song Y, Murphyhill E, Bowdidge R. Why don't software developers use static analysis tools to find bugs? In: Proc. of the 2013 Int'l Conf. on Software Engineering. Piscataway: IEEE Press, 2013. 672–681. [doi: 10.1109/ICSE.2013.6606613]

[15] Parnin C, Orso A. Are automated debugging techniques actually helping programmers? In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2011. 199–209. [doi: 10.1145/2001420.2001445]

[16] Fang M, Hafiz M. Discovering buffer overflow vulnerabilities in the wild: An empirical study. In: Proc. of the 8th ACM/IEEE Int'l Symp. on Empirical Software Engineering and Measurement. New York: ACM Press, 2014. 23. [doi: 10.1145/2652524.2652533]

[17] King JC. Symbolic execution and program testing. Communications of the ACM, 1976,19(7):385–394. [doi: 10.1145/360248.360252]

[18] Dijkstra EW. A Discipline of Programming. New York: Prentice Hall, 1976.

[19] Jin JW, Ma FF, Zhang J. Brief introduction to SMT solving. Journal of Frontiers of Computer Science and Technology, 2015,9(7):769–780 (in Chinese with English abstract). [doi: 10.3778/j.issn.1673-9418.1405041]

[20] Li J, Liu WW. A survey on theoretical combination techniques of SMT solvers. Computer Engineering & Science, 2011,33(10):111–119 (in Chinese with English abstract).

[21] Beckman NE, Nori AV, Rajamani SK, Simmons RJ. Proofs from tests. In: Proc. of the 2008 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2008. 3–14. [doi: 10.1145/1390630.1390634]

[22] Tomb A. Program inconsistency detection: Universal reachability analysis and conditional slicing [Ph.D. Thesis]. Santa Cruz: University of California at Santa Cruz, 2011.

[23] Weiser M. Program slicing. In: Proc. of the 5th Int'l Conf. on Software Engineering. Piscataway: IEEE Press, 1981. 439–449. [doi: 10.1109/TSE.1984.5010248]

[24] Korel B, Laski J. Dynamic program slicing. Information Processing Letters, 1988,29(3):155–163. [doi: 10.1016/0020-0190(88)90054-3]

[25] Sinha N, Singhania N, Chandra S, Sridharan M. Alternate and learn: Finding witnesses without looking all over. In: Madhusudan P, Seshia SA, eds. Proc. of the 24th Int'l Conf. on Computer Aided Verification. Berlin, Heidelberg: Springer-Verlag, 2012. 599–615.

[26] Alipour MA, Groce A, Gopinath R, Christi A. Generating focused random tests using directed swarm testing. In: Proc. of the 25th Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2016. 70–81. [doi: 10.1145/2931037.2931056]

[27] Chen TY, Kuo FC, Merkel RG, Tse TH. Adaptive random testing: The ART of test case diversity. Journal of Systems & Software, 2010,83(1):60–66. [doi: 10.1016/j.jss.2009.02.022]

[28] Harman M, Mansouri SA, Zhang Y. Search-Based software engineering: Trends, techniques and applications. ACM Computing Surveys, 2012,45(1):1–61. [doi: 10.1145/2379776.2379787]

[29] Xuan J, Xie X, Monperrus M. Crash reproduction via test case mutation: Let existing test cases help. In: Proc. of the 10th Joint Meeting on Foundations of Software Engineering. New York: ACM Press, 2015. 910–913. [doi: 10.1145/2786805.2803206]

[30] Soltani M, Panichella A, Deursen AV. A guided genetic algorithm for automated crash reproduction. In: Proc. of the Int'l Conf. on Software Engineering. Piscataway: IEEE Press, 2017. 209–220. [doi: 10.1109/ICSE.2017.27]

[31] Böhme M, Pham VT, Roychoudhury A. Coverage-Based greybox fuzzing as Markov chain. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. New York: ACM Press, 2016. 1032–1043. [doi: 10.1145/2976749.2978428]

[32] Zamfir C, Candea G. Execution synthesis: A technique for automated software debugging. In: Proc. of the 5th European Conf. on Computer Systems. New York: ACM Press, 2010. 321–334. [doi: 10.1145/1755913.1755946]

[33] Ma KK, Phang KY, Foster JS, Hicks M. Directed symbolic execution. In: Yahav E, ed. Proc. of the 18th Int'l Conf. on Static Analysis. Berlin, Heidelberg: Springer-Verlag, 2011. 95–111. [doi: 10.1007/978-3-642-23702-7_11]

[34] Marinescu PD, Cadar C. KATCH: High-Coverage testing of software patches. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering. New York: ACM Press, 2013. 235–245. [doi: 10.1145/2491411.2491438]

[35] Isaev IK, Sidorov DV. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. Programming & Computer Software, 2010,36(4):225–236. [doi: 10.1134/S0361768810040055]

[36] Sidiroglou-Douskos S, Lahtinen E, Rittenhouse N, Piselli P, Long F, Kim D, Rinard M. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In: Proc. of the 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. New York: ACM Press, 2015. 473–486. [doi: 10.1145/2775054.2694389]

[37] Pham VT, Ng WB, Rubinov K, Roychoudhury A. Hercules: Reproducing crashes in real-world application binaries. In: Proc. of the 37th Int'l Conf. on Software Engineering—Vol.1. Piscataway: IEEE Press, 2015. 891–901. [doi: 10.1109/ICSE.2015.99]

[38] Chen N, Kim S. STAR: Stack trace based automatic crash reproduction via symbolic execution. IEEE Trans. on Software Engineering, 2015,41(2):198–220. [doi: 10.1109/TSE.2014.2363469]

[39] Holte RC, Felner A, Sharon G, Sturtevant NR. Bidirectional search that is guaranteed to meet in the middle. In: Schuurmans D, Wellman M P, eds. Proc. of the 30th AAAI Conf. on Artificial Intelligence. Palo Alto: AAAI Press, 2016. 3411–3417.

[40] Xie X, Chen B, Liu Y, Le W, Li X. Proteus: Computing disjunctive loop summary via path dependency analysis. In: Proc. of the 24th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. New York: ACM Press, 2016. 61–72. [doi: 10.1145/2950290.2950340]

[41] Xie X, Liu Y, Le W, Li XH, Chen HX. S-Looper: Automatic summarization for multipath string loops. In: Proc. of the 2015 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2015. 188–198. [doi: 10.1145/2771783.2771815]

[42] Godefroid P, Luchaup D. Automatic partial loop summarization in dynamic test generation. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2011. 23–33. [doi: 10.1145/2001420.2001424]

[43] Nie CJ, Liu HF, Su PR, Feng DG. A loop summarization method for dynamic binary program analysis. Chinese Journal of Electronics, 2014,42(6):1110–1117 (in Chinese with English abstract). [doi: 10.3969/j.issn.0372-2112.2014.06.012]

[44] Silver D, Huang A, Maddison CJ, Guez A, Sifre L, Driessche GVD, Schrittwieser J, Antonoglou L, Panneershelvam V, Lanctot M, Dieleman S, Grewe D, Nham J, Kalchbrenner N, Sutskever L, Lillcrap T, Leach M, Kavukcuoglu K, Graepel T, Hassabis D. Mastering the game of Go with deep neural networks and tree search. Nature, 2016,529(7587):484–489. [doi: 10.1038/nature16961]

[45] Hwang GH, Tai KC, Huang TL. Reachability testing: An approach to testing concurrent software. In: Proc. of the 1st Asia-Pacific Software Engineering Conf. IEEE, 1994. 246–255. [doi: 10.1109/APSEC.1994.465255]

[46] Lu S, Park S, Zhou Y. Finding atomicity-violation bugs through unserializable interleaving testing. IEEE Trans. on Software Engineering, 2012,38(4):844–860. [doi: 10.1109/TSE.2011.35]

[47] Shacham H. The geometry of innocent flesh on the bone: Return-Into-Libc without function calls (on the x86). In: Proc. of the 14th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2007. 552–561. [doi: 10.1145/1315245.1315313]

[48] Bletsch T, Jiang X, Freeh VW, Liang Z. Jump-Oriented programming: A new class of code-reuse attack. In: Proc. of the 6th ACM Symp. on Information, Computer and Communications Security. New York: ACM Press, 2011. 30–40. [doi: 10.1145/1966913.1966919]

[49] Checkoway S, Davi L, Dmitrienko A, Dmitrienko A, Sadegh AR. Return-Oriented programming without returns. In: Proc. of the 17th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2010. 559–572.

[50] Bosman E, Bos H. Framing signals—A return to portable shellcode. In: Proc. of the 2014 IEEE Symp. on Security and Privacy. Washington: IEEE Computer Society, 2014. 243–258. [doi: 10.1109/SP.2014.23]

[51] Bittau A, Belay A, Mashtizadeh A, Mazieres D, Boneh D. Hacking blind. In: Proc. of the 2014 IEEE Symp. on Security and Privacy. Washington: IEEE Computer Society, 2014. 227–242. [doi: 10.1109/SP.2014.22]

[52] Abadi M, Budiu M, Erlingsson U, Ligatti J. Control-Flow integrity. In: Proc. of the 12th ACM Conf. on Computer and Communications Security. New York: ACM Press, 2005. 340–353. [doi: 10.1145/1102120.1102165]

[53] Tice C, Roeder T, Collingbourne P, Checkoway S, Erlingsson Ú, Lozano L, Pike G. Enforcing forward-edge control-flow integrity in GCC & LLVM. In: Fu K, ed. Proc. of the 23rd USENIX Conf. on Security Symp. Berkeley: USENIX Association, 2014. 941–955. https://www.usenix.org/node/184460

[54] Zhang M, Sekar R. Control flow integrity for COTS binaries. In: King S T, ed. Proc. of the 22nd USENIX Conf. on Security. Berkeley: USENIX Association, 2013. 337–352. https://www.usenix.org/node/174767

[55] Zhang C, Wei T, Chen Z, Duan L, Szekeres L, McCamant S, Song D, Zou W. Practical control flow integrity and randomization for binary executables. In: Proc. of the IEEE Symp. on Security and Privacy. Washington: IEEE Computer Society, 2013. 559–573. [doi: 10.1109/SP.2013.44]

[56] Niu B, Tan G. Per-Input control-flow integrity. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. New York: ACM Press, 2015. 914–926. [doi: 10.1145/2810103.2813644]

[57] Veen VVD, Andriesse D, Göktaş E, Gras B, Sambuc L, Slowinska A, Bos H, Giuffrida C. Practical context-sensitive CFI. In: Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security. New York: ACM Press, 2015. 927–940. [doi: 10.1145/2810103.2813673]

[58] Hong H, Shweta S, Sendroiu A, Zhang LC, Saxena P, Liang Z. Data-Oriented programming: On the expressiveness of non-control data attacks. In: Proc. of the IEEE Symp. on Security and Privacy (Oakland 2016). Piscataway: IEEE Press, 2016. [doi: 10.1109/SP.2016.62]

[59] Schwartz EJ, Avgerinos T, Brumley D. Q: Exploit hardening made easy. In: Wagner D, ed. Proc. of the 20th USENIX Conf. on Security Symp. Berkeley: USENIX Association, 2011. 25. http://www.usenix.org/events/sec11/tech/full_papers/Schwartz.pdf

[60] Reps T. Undecidability of context-sensitive data-dependence analysis. ACM Trans. on Programming Languages and Systems (TOPLAS), 2000,22(1):162–186. [doi: 10.1145/345099.345137]

[61] Lhoták O, Hendren L. Context-Sensitive points-to analysis: Is it worth it? In: Proc. of the 15th Int'l Conf. on Compiler Construction. Berlin, Heidelberg: Springer-Verlag, 2006. 47–64. [doi: 10.1007/11688839_5]

[62] Andersen LO. Program analysis and specialization for the C programming language [Ph.D. Thesis]. Copenhagen: University of Copenhagen, 1994.

[63] Steensgaard B. Points-To analysis in almost linear time. In: Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM Press, 1996. 32–41. [doi: 10.1145/237721.237727]

[64] Heintze N, Tardieu O. Demand-Driven pointer analysis. ACM SIGPLAN Notices, 2001,36(5):24–34. [doi: 10.1145/381694.378802]

[65] Zheng X, Rugina R. Demand-Driven alias analysis for C. ACM SIGPLAN Notices, 2008,43(1):197–208. [doi: 10.1145/1328897.1328464]

[66] Yan D, Xu G, Rountev A. Demand-Driven context-sensitive alias analysis for Java. In: Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2011. 155–165. [doi: 10.1145/2001420.2001440]

[67] Shang L, Lu Y, Xue J. Fast and precise points-to analysis with incremental CFL-reachability summarisation: Preliminary experience. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. New York: ACM Press, 2012. 270–273. [doi: 10.1145/2351676.2351720]

[68] Shang L, Xie X, Xue J. On-Demand dynamic summary-based points-to analysis. In: Proc. of the 10th Int'l Symp. on Code Generation and Optimization. New York: ACM Press, 2012. 264–274. [doi: 10.1145/2259016.2259050]

[69] Feng Y, Wang X, Dillig I, Dillig T. Bottom-Up context-sensitive pointer analysis for Java. In: Proc. of the Asian Symp. on Programming Languages and Systems. Cham: Springer Int'l Publishing, 2015. 465–484. [doi: 10.1007/978-3-319-26529-2_25]

[70] Wilhelm R, Sagiv M, Reps T. Shape analysis. In: Proc. of the Int'l Conf. on Compiler Construction. Berlin, Heidelberg: Springer-Verlag, 2000. 1–17. [doi: 10.1007/3-540-46423-9_1]

[71] Sagiv M, Reps T, Wilhelm R. Solving shape-analysis problems in languages with destructive updating. ACM Trans. on Programming Languages and Systems (TOPLAS), 1998,20(1):1–50. [doi: 10.1145/271510.271517]

[72] Sagiv M, Reps T, Wilhelm R. Parametric shape analysis via 3-valued logic. ACM Trans. on Programming Languages and Systems (TOPLAS), 2002,24(3):217–298. [doi: 10.1145/514188.514190]

[73] Itzhaky S, Bjørner N, Reps T, Sagiv M, Thakur A. Property-Directed shape analysis. In: Armin B, Roderick B, eds. Proc. of the Int'l Conf. on Computer Aided Verification. Cham: Springer Int'l Publishing, 2014. 35–51. [doi: 10.1007/978-3-319-08867-9_3]

[74] Godefroid P. Compositional dynamic test generation. ACM Sigplan Notices, 2007,42(1):47–54. [doi: 10.1145/1190215.1190226]

[75] Anand S, Godefroid P, Tillmann N. Demand-Driven compositional symbolic execution. In: Ramakrishnan CR, Jakob R, eds. Proc. of the Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer-Verlag, 2008. 367–381. [doi: 10.1007/978-3-540-78800-3_28]

[76] Avgerinos T, Rebert A, Cha SK, Brumley D. Enhancing symbolic execution with veritesting. In: Proc. of the 36th Int'l Conf. on Software Engineering. New York: ACM Press, 2014. 1083–1094. [doi: 10.1145/2568225.2568293]

[77] Yi Q, Yang Z, Guo S, Wang C, Liu J, Zhao C. Postconditioned symbolic execution. In: Proc. of the IEEE Int'l Conf. on Software Testing, Verification and Validation. IEEE, 2015. 1–10. [doi: 10.1109/ICST.2015.7102601]

[78] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves R, Renesse R V, eds. Proc. of the USENIX Conf. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2008. 209–224. https://dl.acm.org/citation.cfm?id=1855756

[79] He YX, Wu W, Chen Y, Xu C. Path sensitive program verification based on SMT solvers. Ruan Jian Xue Bao/Journal of Software, 2012,23(10):2655–2664 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4196.htm [doi: 10.3724/SP.J.1001.2012.04196]

[80] Qin S, He G, Luo C, Chin WN, Chen X. Loop invariant synthesis in a combined abstract domain. Journal of Symbolic Computation, 2013,50(3):386–408. [doi: 10.1016/j.jsc.2012.08.007]

[81] Li ZP, Zhang Y, Chen YY. A shape graph logic and a shape system. Journal of Computer Science and Technology, 2013,28(6): 1063–1084. [doi: 10.1007/s11390-013-1398-1]

[82] Zhang Y, Chen YY, Li ZP. Theorem proving for a theory of shape graphs. Chinese Journal of Computers, 2016,39(12):2460–2480 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2016.02460]

[83] Chipounov V, Georgescu V, Zamfir C, Candea G. Selective symbolic execution. In: Fetzer C, Rodrigues R, eds. Proc. of the Workshop on Hot Topics in System Dependability. 2009. 1286–1299. https://infoscience.epfl.ch/record/139393/files/selsymbex.pdf

[84] Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, 2011,46(3):265–278. [doi: 10.1145/1961296.1950396]

[85] Henderson A, Prakash A, Yan LK, Hu X, Wang X, Zhou R, Yin H. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In: Proc. of the 2014 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2014. 248–258. [doi: 10.1145/2610384.2610407]

[86] Henderson A, Yan LK, Hu X, Prakash A, Yin H, McCamant S. DECAF: A platform-neutral whole-system dynamic binary analysis platform. IEEE Trans. on Software Engineering, 2017,43(2):164–184. [doi: 10.1109/TSE.2016.2589242]

[87] Lei Y, Carver RH. Reachability testing of concurrent programs. IEEE Trans. on Software Engineering, 2006,32(6):382–403. [doi: 10.1109/TSE.2006.56]

[88] Sen K, Agha G. Automated systematic testing of open distributed programs. In: Luciano B, Reiko H, eds. Proc. of the 9th Int'l Conf. on Fundamental Approaches to Software Engineering. Berlin, Heidelberg: Springer-Verlag, 2006. 339–356. [doi: 10.1007/11693017_25]

[89] Taylor RN, Levine DL, Kelly CD. Structural testing of concurrent programs. IEEE Trans. on Software Engineering, 1992,18(3): 206–215. [doi: 10.1109/32.126769]

[90] Koppol PV, Tai KC. An incremental approach to structural testing of concurrent software. ACM SIGSOFT Software Engineering Notes, 1996,21(3):14–23. [doi: 10.1145/226295.226298]

[91] Long ZY. Automatic analysis and verification of concurrent programs [Ph.D. Thesis]. Beijing: University of Chinese Academy of Sciences, 2013 (in Chinese with English abstract).

[92] Zeng YQ. Research on context-bounded reachability of concurrent programs [MS. Thesis]. Guilin: Guilin University of Electronic Technology, 2013 (in Chinese with English abstract).

[93] Musuvathi M, Qadeer S, Ball T, Basler G, Nainar PA, Neamtiu I. Finding and reproducing Heisenbugs in concurrent programs. In: Draves R, Renesse RV, eds. Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2008. 267–280. https://dl.acm.org/citation.cfm?id=1855760

[94] Park S, Lu S, Zhou Y. CTrigger: Exposing atomicity violation bugs from their hiding places. In: Proc. of the Architectural Support for Programming Languages and Operating Systems. 2009. 25–36. [doi: 10.1145/1508284.1508249]

[95] Sen K. Race directed random testing of concurrent programs. ACM SIGPLAN Notices, 2008,43(6):11–21. [doi: 10.1145/1379022.1375584]

[96] Weeratunge D, Zhang X, Jagannathan S. Analyzing multicore dumps to facilitate concurrency bug reproduction. In: Proc. of the Int'l Conf. on Architectural Support for Programming Languages & Operating Systems. New York: ACM Press, 2010. 155–166. [doi: 10.1145/1736020.1736039]

[97] Huang J, Zhang C. LEAN: Simplifying concurrency bug reproduction via replay-supported execution reduction. In: Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications. New York: ACM Press, 2012. 451–466. [doi: 10.1145/2398857.2384649]

[98] Li X, Liang Y, Qian H, Hu YQ, Bu L, Yu Y, Chen X, Li X. Symbolic execution of complex program driven by machine learning based constraint solving. In: Proc. of the 31st IEEE/ACM Int'l Conf. on Automated Software Engineering. New York: ACM Press, 2016. 554–559. [doi: 10.1145/2970276.2970364]

[99] Dinges P, Agha G. Targeted test input generation using symbolic-concrete backward execution. In: Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. New York: ACM Press, 2014. 31–36. [doi: 10.1145/2642937.2642951]

[100] Person S, Yang G, Rungta N, Khurshid S. Directed incremental symbolic execution. ACM SIGPLAN Notices, 2011,46(6):504–515.

[101] Babić D, Martignoni L, McCamant S, Song D. Statically-Directed dynamic automated test generation. In: Dwyer M, ed. Proc. of the 2011 Int'l Symp. on Software Testing and Analysis. New York: ACM Press, 2011. 12–22. [doi: 10.1145/2001420.2001423]

[102] Parvez MR. Combining static analysis and targeted symbolic execution for scalable bug-finding in application binaries [MS. Thesis]. Waterloo: University of Waterloo, 2016.

[103] Gao F, Wang L, Li X. BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities. In: Lo D, Apel S, Khurshid S, eds. Proc. of the IEEE/ACM Int'l Conf. on Automated Software Engineering. New York: ACM Press, 2016. 786–791.

[104] Jin W, Orso A. BugRedux: Reproducing field failures for in-house debugging. In: Proc. of the 34th Int'l Conf. on Software Engineering. Piscataway: IEEE Press, 2012. 474–484. [doi: 10.1109/ICSE.2012.6227168]

[105] Sang KC, Avgerinos T, Rebert A, Brumley D. Unleashing mayhem on binary code. In: Proc. of the IEEE Symp. on Security and Privacy. Washington: IEEE Computer Society, 2012. 380–394. [doi: 10.1109/SP.2012.31]

[106] Ramos DA, Engler D. Under-Constrained symbolic execution: Correctness checking for real code. In: Jung J, ed. Proc. of the 24th USENIX Security Symp. Berkeley: USENIX Association, 2015. 49–64. https://www.usenix.org/node/196226

[107] Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H. VUzzer: Application-Aware evolutionary fuzzing. In: Proc. of the Network and Distributed System Security Symp. San Diego: Internet Society, 2017. [doi: 10.14722/ndss.2017.23404]

[108] Jana S, Kang YJ, Roth S, Ray B. Automatically detecting error handling bugs using error specifications. In: THolz T, Savage S, eds. Proc. of the USENIX Security Symp. Berkeley: USENIX Association, 2016. 345–362.

附中文参考文献:

[5] 郭曦,王盼,王建勇,张焕国.基于 k 近邻最弱前置条件的程序多路径验证方法.计算机学报,2015,38(11):2203–2214. [doi: 10.11897/SP.J.1016.2015.02203]

[11] 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. [doi: 10.3724/SP.J.1016.2009.01697]

[19] 金继伟,马菲菲,张健.SMT 求解技术简述.计算机科学与探索,2015,9(7):769–780. [doi: 10.3778/j.issn.1673-9418.1405041]

[20] 李婧,刘万伟.SMT 求解器理论组合技术研究.计算机工程与科学,2011,33(10):111–119.

[43] 聂楚江,刘海峰,苏璞睿,冯登国.一种面向程序动态分析的循环摘要生成方法.电子学报,2014,42(6):1110–1117. [doi: 10.3969/j.issn.0372-2112.2014.06.012]

[79] 何炎祥,吴伟,陈勇,徐超.基于 SMT 求解器的路径敏感程序验证.软件学报,2012,23(10):2655–2664. http://www.jos.org.cn/1000-9825/4196.htm [doi: 10.3724/SP.J.1001.2012.04196]

[82] 张昱,陈意云,李兆鹏.形状图理论的定理证明.计算机学报,2016,39(12):2460–2480. [doi: 10.11897/SP.J.1016.2016.02460]

[91] 龙震岳.并发程序的自动分析与验证[博士学位论文].北京:中国科学院大学,2013.

[92] 曾宇清.基于上下文限界的并发程序可达性研究[硕士学位论文].桂林:桂林电子科技大学,2013.

杨克(1989－),男,河北辛集人,硕士,CCF 学生会员,主要研究领域为软件安全分析,操作系统安全.

贺也平(1962－),男,博士,研究员,博士生导师,主要研究领域为系统安全,隐私保护.

马恒太(1970－),男,博士,副研究员,主要研究领域为软件安全分析,操作系统安全.

王雪飞(1989－),女,助理工程师,主要研究领域为软件安全分析,操作系统安全.