软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(1):80−109 [doi: 10.13328/j.cnki.jos.005651]

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

程序分析研究进展*

张 健[1,2], 张 超[3], 玄跻峰[4], 熊英飞[5], 王千祥[6], 梁 彬[7], 李 炼[2,8], 窦文生[1,2], 陈振邦[9], 陈立前[9], 蔡 彦[1]

[1](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)

[2](中国科学院大学,北京 100049)

[3](清华大学 网络科学与网络空间研究院,北京 100084)

[4](武汉大学 计算机学院,湖北 武汉 430072)

[5](高可信软件技术教育部重点实验室(北京大学),北京 100871)

[6](华为技术有限公司,北京 100095)

[7](中国人民大学 信息学院,北京 100872)

[8](中国科学院 计算技术研究所,北京 100190)

[9](国防科技大学 计算机学院,湖南 长沙 410073)

通讯作者: 张健, E-mail: zj@ios.ac.cn

摘 要: 在信息化时代,人们对软件的质量要求越来越高.程序分析是保障软件质量的重要手段之一,日益受到学术界和产业界的重视.介绍了若干基本程序分析技术(抽象解释、数据流分析、基于摘要的分析、符号执行、动态分析、基于机器学习的程序分析等),特别是最近 10 余年的研究进展.进而介绍了针对不同类型软件(移动应用、并发软件、分布式系统、二进制代码等)的分析方法.最后展望了程序分析未来的研究方向和所面临的挑战.

关键词: 程序分析;软件质量保障;静态分析;动态分析

中图法分类号: TP311

中文引用格式: 张健,张超,玄跻峰,熊英飞,王千祥,梁彬,李炼,窦文生,陈振邦,陈立前,蔡彦.程序分析研究进展.软件学报,2019,30(1):80−109. http://www.jos.org.cn/1000-9825/5651.htm

英文引用格式: Zhang J, Zhang C, Xuan JF, Xiong YF, Wang QX, Liang B, Li L, Dou WS, Chen ZB, Chen LQ, Cai Y. Recent progress in program analysis. Ruan Jian Xue Bao/Journal of Software, 2019,30(1):80−109 (in Chinese). http://www.jos.org.cn/1000-9825/5651.htm

Recent Progress in Program Analysis

ZHANG Jian[1,2], ZHANG Chao[3], XUAN Ji-Feng[4], XIONG Ying-Fei[5], WANG Qian-Xiang[6], LIANG Bin[7], LI Lian[2,8], DOU Wen-Sheng[1,2], CHEN Zhen-Bang[9], CHEN Li-Qian[9], CAI Yan[1]

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

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

[3](Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China)

* 基金项目: 国家重点基础研究发展计划(973)(2014CB340701); 中国科学院前沿科学重点项目(QYZDJ-SSW-JSC036); 国家自然科学基金(61772308, U1736209, 61872273, 61672045, 61472440, 61632015, 61872445, 61502465)

Foundation item: National Key Basic Research Program of China (973) (2014CB340701); Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036); National Natural Science Foundation of China (61772308, U1736209, 61872273, 61672045, 61472440, 61632015, 61872445, 61502465)

本文由"软件学科发展回顾特刊"特约编辑梅宏教授、金芝教授、郝丹副教授推荐.

收稿时间: 2018-08-08; 修改时间: 2018-08-30; 采用时间: 2018-09-25; jos 在线出版时间: 2018-11-22

CNKI 网络优先出版: 2018-11-23 07:18:06, http://kns.cnki.net/kcms/detail/11.2560.TP.20181123.0717.006.html

[4](School of Computer Science, Wuhan University, Wuhan 430072, China)
[5](Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100871, China)
[6](Huawei Technologies Co. Ltd., Beijing 100095, China)
[7](School of Information, Renmin University of China, Beijing 100872, China)
[8](Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China)
[9](School of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract: In the information age, people are increasingly demanding high quality of software systems. Program analysis is one of the important approaches to guarantee the quality of software, and has been receiving attentions from academia and industry. This article mainly focuses on the research progress in program analysis in the last decade. First, the article introduces the basic program analysis techniques, including abstract interpretation, data flow analysis, summary-based analysis, symbolic execution, dynamic analysis, machine learning-based program analysis, etc. Then, it summarizes program analysis approaches for different types of software systems, including mobile applications, concurrent software, distributed systems, binary code, etc. Finally, the article discusses potential research directions and challenges of program analysis in the future.

Key words: program analysis; software quality assurance; static analysis; dynamic analysis

随着信息化的不断发展,软件对人们生活的影响越来越大,在国民经济和国防建设中具有越来越重要的地位.如何提高软件质量,保证其行为的可信性,是学术界和工业界共同关注的重要问题.要解决该问题,应加强软件开发过程管理,在全生命周期采取各种方法和技术提升软件质量.由于软件系统的复杂性,在编码实现完成之后,甚至在软件产品发布、被广泛使用之后,往往还有各种各样的缺陷和漏洞.各种软件测试和分析技术,是发现这些缺陷、漏洞的有效手段.

不同于很多黑盒测试方法,软件分析技术可深入软件系统内部,细致地考察其结构及各个组成部分,进而发现其各种特性,如性能、正确性、安全性等.软件分析已逐渐发展成为程序语言和软件工程领域的一个重要研究方向,并已影响到信息安全等相关领域.进入 21 世纪以来,该方向进展显著,研究成果不断涌现.软件分析不仅可用来发现软件中的缺陷、漏洞,还可用于软件理解、修复以及测试用例生成等方面[1].

软件包含程序和文档.由于篇幅所限,本文主要介绍程序分析方面的研究,特别是最近 10 余年该方向的一些重要工作.本文将介绍程序分析的基本概念(分析对象、难度、评价等)、主要的基础性分析技术、针对不同类型分析对象的分析方法等.最后,简要提及一些挑战性问题以及新兴的研究方向.

1 程序分析简介

本节介绍程序分析的基本概念、程序及其性质多样性等,进而解释程序分析的难度及其评价.

1.1 基本概念

程序分析指的是:对计算机程序进行自动化的处理,以确认或发现其特性,比如性能、正确性、安全性等[2].程序分析的结果可用于编译优化、提供警告信息等,比如被分析程序在某处可能出现指针为空、数组下标越界的情形等.

传统上,程序分析包括各种静态分析技术(类型检查、数据流分析、指向分析等)与动态分析技术:所谓的静态分析,是指对程序代码进行自动化的扫描、分析,而不必运行程序;与静态分析相对应的是动态分析技术,其利用程序运行过程中的动态信息,分析其行为和特性.

与程序分析密切相关的两类方法是形式验证及测试:前者试图通过形式化方法,严格证明程序具有某种性质,目前,其自动化程度尚有不足,难于实用;测试方法多种多样,在实际工程中广泛使用,这些方法也是以发现程序中的缺陷为目的,它们一般都需要人们提供输入数据,以便运行程序,观察其输出结果.

1.2 程序及其性质的多样性

程序分析内涵比较丰富,具有多样性.

1) 被分析对象的多样性

程序分析的对象既可以是源代码,也可以是二进制可执行代码.对于源程序,根据其实现语言不同,可能需要不同的分析技术.比如,C 程序和 Java 程序的分析,其技术可能就不一样:后者需要处理一些面向对象的结构.即使是同一种高级语言编写的程序,也可能具有不同的特性.比如,程序中是否有比较多的指针?是否使用并发控制结构、通信机制?是否有大量的数值计算.从规模看,被分析的程序可以是几百行的代码,也可以是几十万行乃至于上千万行代码.

2) 程序性质的多样性

对不同类型的程序,在不同的应用环境下,人们关注的性质也不一样.以 C 程序为例,很多 C 程序会用到指针,用于动态分配、释放内存.对这样的程序,人们会关注空指针、内存泄漏等问题.但对某些嵌入式系统来说,程序中很少动态分配内存空间,却可能要处理中断.另外一些 C 程序可能有大量的数值(浮点数)计算,程序员可能更关注其中是否存在溢出问题.

1.3 程序分析的难度及评价

由于程序语言的复杂性、程序性质的多样性,自动化的程序分析往往具有相当大的难度.根据 Rice 定理:对于程序行为的任何非平凡属性,都不存在可以检查该属性的通用算法[3].也就是说,大量的程序分析问题都是不可判定的.比如,一般情况下,程序的终止性就是不可判定的.也就是说,不存在一种算法,它能判断任何给定的程序是否总能终止.程序路径的可行性(path feasibility)也是不可判定的[4].如果存在输入数据使得程序沿着一条路径执行,程序中该路径被称为是可行的(feasible).没有一种算法,它能判断程序中某条给定路径是否可行.最近,Cousot 等人[5]从抽象解释的角度形式化地证明了:从可计算性角度看,相比程序验证(只需要检查一个给定的程序不变式是否正确),程序分析(需要综合出正确的程序不变式)是一个更难的问题.具体而言:对于证明有穷域上的程序断言,程序分析与程序验证是等价的问题;但是对于无穷域上的断言,程序分析比程序验证更难.

鉴于程序分析的理论难度以及被分析程序的复杂度,我们往往不要求对程序进行完全精确的分析.这就可能带来误报(false positive)、漏报(false negative)等问题:前者指的是报警信息指出的缺陷实际上不存在、不可能发生;后者指的是程序中存在的某个缺陷,没有被程序分析工具所发现.

对程序分析技术或工具进行评价,不仅要看其分析对象的规模、复杂度,分析过程的效率,还要看其对用户的要求,发现缺陷的严重程度,以及误报率、漏报率等.

2 程序分析方法和技术

本节主要介绍一些近期有重要进展的程序分析方法和技术,包括抽象解释、数据流分析、基于摘要的过程间分析、符号执行、动态分析、基于机器学习的程序分析等.这 6 项方法和技术的关系如图 1 所示.

Fig.1 Main program analysis techniques

图 1 主要的程序分析技术

程序分析总体可以分为静态分析和动态分析,涉及的基础理论包括抽象解释、约束求解、自动推理等.而静态分析的关键技术包括数据流分析、过程间分析、符号执行等.最后,近期机器学习技术被用于提升各种不同的程序分析技术.

除了这 6 种基本程序分析技术之外,还有一些其他广泛使用的程序分析技术也在近期有显著进展,如污点分析、模型检验、编程规则检查[6]等,由于受篇幅所限,本文将不再对这些技术进行详述.

2.1 抽象解释

抽象解释是一种对程序语义进行可靠抽象(或近似)的通用理论[7].该理论为程序分析的设计和构建提供了

一个通用的框架[8],并从理论上保证了所构建的程序分析的终止性和可靠性(即考虑了所有的程序行为).基于抽象解释来设计程序分析,本质上是通过对程序语义进行不同程度的抽象,以在分析精度和计算效率之间取得权衡.这种由某种语义抽象及其上的操作所构成的数学结构称为抽象域.抽象解释采用 Galois 连接来刻画具体域与抽象域之间的关系.设$\langle D,\sqsubseteq\rangle$和$\langle D^{\#},\sqsubseteq^{\#}\rangle$是两个给定的偏序集,函数$\alpha:D\to D^{\#}$及$\gamma:D^{\#}\to D$构成的函数对$(\alpha,\gamma)$称为$D$与$D^{\#}$之间的 Galois 连接,当且仅当$\forall x\in D,x^{\#}\in D^{\#},\alpha(x)\sqsubseteq^{\#}x^{\#}\Leftrightarrow x\sqsubseteq\gamma(x^{\#})$,其中,$\langle D,\sqsubseteq\rangle$称为具体域,$\langle D^{\#},\sqsubseteq^{\#}\rangle$称为抽象域,$\alpha$称为抽象化函数,$\gamma$称为具体化函数.由定义中的性质$\alpha(x)\sqsubseteq^{\#}x^{\#}$(亦即 $x\sqsubseteq\gamma(x^{\#})$)可知,$x^{\#}$是 x 的可靠抽象(又称上近似).给定具体域$\langle D,\sqsubseteq\rangle$和抽象域$\langle D^{\#},\sqsubseteq^{\#}\rangle$之间的 Galois 连接$(\alpha,\gamma)$,对于具体域 D 上的函数 f 和抽象域 $D^{\#}$上的函数 $f^{\#}$,当$\forall x^{\#}\in D^{\#},(f\circ\gamma)(x^{\#})\sqsubseteq(\gamma\circ f^{\#})(x^{\#})$时,我们称$f^{\#}$是$f$的可靠抽象.抽象解释理论的一个重要思想是,通过在抽象域上计算程序的抽象不动点来表达程序的抽象语义.在程序分析中,程序状态集合通过抽象域中的域元素来近似,而程序语义动作(如赋值、条件测试等)通过抽象域中的域操作(如迁移函数)来可靠建模.此外,为了加速抽象域上不动点迭代的收敛速度并保证不动点迭代的终止性,抽象解释框架提供了加宽算子(widening).抽象解释框架能够保证抽象域上迭代求得的抽象不动点是程序最小不动点(对应程序的具体聚集语义)的上近似.换言之,抽象解释提供了严格的理论来保证基于上近似抽象的推理的可靠性,即:所有基于上近似抽象推理得出的性质,在原程序中也必然成立.但是,由于抽象带来的精度损失,不保证所有在原程序中成立的性质都能基于上近似抽象推理得到.

抽象域是抽象解释框架下的核心要素,一般是面向某类特定性质设计的.到目前为止,已出现了数十种面向不同性质的抽象域.其中,具有代表性的抽象域包括区间抽象域、八边形抽象域、多面体抽象域等数值抽象域.此外,还出现了若干开源的抽象域库,如 APRON[9]、ELINA[10]、PPL[11]等.基于抽象解释的程序分析工具也不断涌现,出现了 PolySpace[12]、Astrée[13]等商业化工具和 Frama-C Value Analysis[14]、CCCheck(code contract static checker)[15]、Interproc[16]等学术界工具.随着这些工具的日益完善,抽象解释在工业界大规模软件,尤其是嵌入式软件的分析与验证中得到了成功应用.典型的应用案例是,Astrée 成功应用于空客 A340(约 13.2 万行 C 代码)、A380(约 35 万行 C 代码)等系列飞机飞行控制软件的自动分析并实现了分析的零误报[17].最近,Astrée 的扩展版本 AstréeA 支持多线程 C 程序中运行时错误、数据竞争、死锁等的检测,并成功应用于 ARINC 653 航空电子应用软件(约 220 万行代码)的分析[18].总体而言,基于抽象解释的程序分析主要面临提高分析精度、可扩展性两方面的挑战.

在提高分析精度方面,基于加宽的不动点迭代过程所导致的分析精度损失问题和抽象域本身表达能力的局限性是当前面临的主要问题.

- 在缓解加宽所导致的精度损失方面,近年来的研究进展大致可以分为两种思路:(1) 使用基于策略迭代[19]或抽象加速[20]的不动点迭代过程来取代传统的基于加宽的不动点迭代过程,以获得更精确的分析结果,但是,该方法只适用于一些特殊类别的程序和抽象域;(2) 改进加宽/变窄算子及其迭代序列,如结合加宽变窄算子的交叠迭代策略[21].
- 在弥补抽象域本身表达能力的局限性方面,最近的研究进展可分为 3 类:(1) 将符号化方法与抽象方法结合起来,利用 SMT 求解器[22,23]、插值[24]等技术来计算程序中语句迁移函数的最佳抽象,以改进抽象域在语句迁移函数上的精度损失;(2) 提高抽象域的析取表达能力,如基于区间线性代数、绝对值约束、集合差、非格结构、决策树等方法构造的非凸抽象域[25,26];(3) 提高抽象域的非线性表达能力,如基于组合递推分析[27,28]将符号化分析与抽象解释结合起来以生成多项式、指数、对数等形式非线性不变式,基于椭圆幂集来生成二次不变式[29]等.

在提高可扩展性方面,如何有效降低分析过程中抽象状态表示与计算的时空开销是目前考虑的主要问题.在这方面,最近的研究进展包括:

- 利用变量访问的局部性原理,降低当前抽象环境中所涉及的变量维数,并根据数据流依赖的稀疏性,降低抽象状态的存储开销和传播开销.基于该思想,最近,Oh 等人提出了一种通用的全局稀疏分析框架,在不损失分析精度的前提下能够显著降低时空开销,并在静态分析工具 Sparrow 上进行了应用,取

得了显著的可扩展性提升效果[30,31];

- 利用矩阵分解等在线分解优化策略来对抽象域操作的算法进行优化.基于该思想,最近,Singh 等人[32,33]对常用的八边形抽象域、多面体抽象域的实现进行了优化,优化后,分析的性能取得了显著的提升,性能提升最大的达 140 多倍;在此基础上,Singh 等人还提出了一种通用的基于分解的优化策略[34],能够在不改变基抽象域的基础上自动实现基于分解的优化,从而无需人工重新实现基抽象域,并基于这一思想实现了开源抽象域库 ELINA.这种基于在线分解的方法不会造成精度损失.

最近,Singh 等人[35]还提出了一种基于强化学习来加速静态程序分析的方法,在每次迭代过程中,利用强化学习来决策选择哪个转换子,以在精度和不动点迭代收敛速度之间进行权衡.在抽象域编码实现上,Becchi 等人[36]最近改进了未必封闭多面体域(支持严格不等式约束)的双重描述法,在表示中避免了松弛变量的引入,极大地提高了分析效率,并开发了多面体域的开源实现 PPLite.

此外,将抽象解释应用到特定类型程序或特定性质的分析验证方面的研究也取得了不少进展,主要的关注点包括复杂数据结构自动分析的支持、不同谱系目标程序的支持、活性性质分析的支持.在复杂数据结构的自动分析方面,最近的研究重点关注针对数组内容的精确分析[37]、混杂数据结构的建模[38]、数值与形态混合的程序分析[39,40]、关系型形态分析[41].在支持不同谱系目标程序方面,最近的研究重点关注多线程程序的自动分析[18]、中断驱动型程序的自动分析[42,43]、概率程序的分析[44,45]、操作系统代码的安全和功能性分析[38,46]、JavaScript 等动态语言的分析[47]、二进制代码的分析[48]、Web 应用程序的安全性分析[49].在目标性质支持方面,近年来,在抽象解释领域出现了一些新的用来分析时序性质和终止性的方法[50−52].

未来,抽象解释技术将进一步在新的架构、语言、应用等实际需求驱动下不断发展.值得关注的方向包括对弱内存模型的分析验证[53,54]、神经网络的分析与验证[55,56]、大数据处理相关错误的分析[57]、Python 程序的自动分析[58]等.与约束求解、自动推理、人工智能等基础支撑技术的紧密结合,将是抽象解释后续的研究趋势之一[59−61].同时,降低误报率将依然是基于抽象解释的程序分析技术拓展实际应用的研究挑战和重点.

2.2 数据流分析

数据流分析通过分析程序状态信息在控制流图中的传播来计算每个静态程序点(语句)在运行时可能出现的状态.经典的数据流分析理论[62]用有限高度的格$\langle L,\sqcap\rangle$来抽象表示所有可能状态的集合,并对每个程序语句定义一个单调的转移函数(transfer function),以计算其对程序状态的更新.数据流分析可以是前向或后向,对应程序状态信息在控制流图中的前向或后向传播.在程序控制流图中,多个分支交汇的程序点状态为其所有前驱(或后继,如果是后向传播)程序点状态的$\sqcap$,表示不同执行路径下可能出现的所有程序状态.

数据流分析为抽象解释的一个特例,其计算的状态信息(抽象域)局限于有限高度的格$\langle L,\sqcap\rangle$.数据流分析已经在编译器实现中得到广泛应用,常见的应用包括常数传播分析、部分冗余分析等.相比于通用的抽象解释理论,经典数据流分析的实现可以通过一个迭代计算框架来计算所有语句的输出直至不动点.单调性和格的有限高度保证了数据流分析迭代计算框架的收敛性,而无需引入加宽算子.编译器中的数据流分析多为过程内数据流分析,全局过程间的分析可以使用基于摘要的方法,通过对函数自动分析摘要得以实现.近年来,对数据流分析方向的应用已不仅仅局限于编译优化,研究者们也提出了多种方法来高效实现过程间的上下文敏感的数据流分析,主要包括如下两种方法.

1) IFDS/IDE 数据流分析框架

IFDS 分析框架由 Reps 等人[63]于 1995 年提出.IFDS 将抽象域(即数据流分析计算的状态信息)为满足分配性的有限集合的一大类数据流分析问题转换为一个图可达问题,从而能够有效地进行上下文敏感的过程间分析. IFDS 框架基于程序过程间控制流图定义了一个超级流图(supergraph),其中,每个节点对应在一个程序点的抽象域中的一个元素;而节点间的边表示该元素在过程间控制流图的传播,对应着数据流分析中的转移函数.通过求解是否存在从程序入口到每个程序点的一个可达路径,我们可以得到该程序点的状态信息.基于该分析方法的框架已经实现于开源分析系统,如 Soot[64]和 Wala[65]中,并广泛用于包括 Android 污点分析[66]在内的多种应

用中.

Reps 等人[67]后续进一步扩展了该框架,通过已有抽象域为环境而计算得到新的属性,用于过程间常数传播等应用中,并形式化定义了上述图可达问题为上下文无关文法图可达问题.上下文无关文法图可达问题对图中的边进行标号,图中任意两点可达需要两点间的一条路径上的标号串满足事先定义的上下文无关文法.多种不同程序分析问题均可表示为对上下文无关文法图可达问题的求解.近年来,包括指针分析[68,69]、并行程序分析[70,71]等多种分析技术均通过求解上下文无关文法图可达问题来有效地加以实现.

2) 基于值流图的稀疏数据流分析方法

传统的数据流分析在程序控制流图上将所需计算的状态信息在每个程序点传播得到最终分析结果.此过程通常存在较多冗余操作,对效率,特别是过程间数据流分析效率会有很大影响.为了进一步提高数据流分析的效率,近年来,研究者们提出了多种稀疏的分析方法,从而无需计算状态信息在每个程序点的传播即可得到与数据流分析相同的结果.该类分析技术[72−76]通过一个稀疏的值流图(value flow graph)直接表示程序变量的依赖关系,从而使得状态信息可以有效地在该稀疏的值流图上传播.该值流图保证了状态信息可以有效地传播到其需要使用该信息的程序点,并避免了在无效程序点的冗余传播,可大幅度提高效率.

2.3 基于摘要的过程间分析

摘要(summary)是可复用的程序模块分析结果,能够简要地刻画模块的外部行为.创建摘要和利用摘要开展分析的过程称为摘要分析.在编写程序的过程中,一个程序模块往往需要调用其他程序模块.对主调(caller)模块的分析必然涉及到对被调模块(callee)的分析.与复用被调模块代码可提高软件生产效率类似,摘要分析期望复用被调模块的分析结果也能提高分析主调模块的效率.通过基于摘要的程序分析技术,我们将被复用模块的分析结果被构造成为摘要,并在分析主调模块时对其实例化,以加快分析速度.

传统摘要分析的研究主要关注模块化(modular)分析,即:在分析过程中将软件划分为多个模块,对各个模块分别分析和创建摘要,然后合并摘要获得整体的分析结果.这样,在分析过程中创建摘要的技术也称为在线摘要技术.摘要分析在最近 10 年的主要进展是离线摘要技术,即:在程序分析之前对常用代码库生成摘要,从而加快使用这些代码库的客户端的分析速度.离线摘要根据其自动化程度的不同,可以分成人工编写摘要和自动生成摘要,分别在接下来的两小节中加以介绍.

1) 人工编写摘要

对于程序中一些难以分析的代码,例如第三方的库和系统代码等,可采用人工编写摘要的方式近似代码的行为.这种技术原先应用于程序验证中,比如 ESC/Java 通过用户提供的前置条件和后置条件来对程序进行模块化验证.FlowDroid[66]在处理调用系统代码和调用其他组件时,采用由人工来编写的近似摘要.这样的摘要能够模拟被调模块中的常见数据流路径.这种方式没有考虑到用户编写的代码对被调模块的影响.Ali 和 Lhoták[77,78]提出了一种对库建立轻量级的上近似摘要.这种摘要的建立,需要代码库满足分离编译假设(separate compilation assumption),无需应用代码即可编译代码库.Java 代码库普遍满足这项假设.在该假设约束下的 Java 标准库被简化为一个人工编写的占位库,而这个占位库的文件仅有 80KB 大小,远小于 Java 标准库的原本大小.Ali 和 Lhoták[77,78]的方法改造了现有的全程序调用图构造算法,以占位库作为所有库函数的替代品,所有应用代码与库函数代码间的调用关系被简化为应用代码与占位库间的调用关系.该方法既提升了效率(10 倍以上),又保证了正确性,并且有足够的精确度.

2) 自动生成摘要

在程序分析过程前创建摘要,通常是对程序常用的模块(如 Java 代码库)进行摘要分析,随后加速客户代码的分析.诸多的未知信息[79]和代码库太大造成算法难以扩展.Cousot[80]从理论上总结了创建模块摘要的各种方式.该工作首先分析了创建模块摘要的主要问题是处理模块间的环形依赖.在没有环形依赖时,我们可以按照依赖关系给模块排一个顺序,每个模块只依赖于之前的模块,然后按顺序依次分析.但是,由于环形依赖的存在,所有环形依赖的模块就只能被当作一个模块分析,使得创建模块的摘要无法进行.文献[80]针对环形依赖给出了 3 种解决方案.

- 基于简化的分析:这个方案的思路是针对环形依赖导致的大模块上的分析通过标准的编译技术(如partial evaluation)进行化简,提高分析速度.每个模块上的分析程序可以被当成是一个函数,然后,把这些函数看成是标准程序之后就可以应用普通的程序分析进行化简了;
- 最坏情况分析:每个模块分别分析,对于该模块所依赖的其他模块作最坏情况假设.这样的分析会导致不精确,并且在多数情况下严重不精确;
- 符号化关系型分析:把未知的信息做成符号化信息,然后把程序分析转变成符号化分析.所有与符号化相关的部分不进行计算.

Smaragdakis等人[81]提出了针对所有流不敏感指针分析的代码库预处理技术,属于基于简化的分析.该工作发现:当代码中存在符合图代码模式时,将模式中的冗余语句删除,进而避免冗余运算,加速代码分析.Rountev和Ryder[82]提出了另一种代码库的简化方式,对库的摘要就是把库里面所有的赋值语句提取出来.这个摘要本身只是省掉了语法分析的时间.然后在摘要上进行了两个优化.

a) 对于连续赋值,比如 $a=b;c=a$;去掉中间变量,替换成 $c=b$;

b) 去掉与客户端无关的赋值语句.

这两个优化能省掉4%~69%的时间.该方法仅支持上下文不敏感且流不敏感的分析.

Rountev等人[83,84]提出的构件级别分析(component-level analysis)对代码库的IFDS/IDE框架进行了摘要分析.该方法采用Sharir和Pnueli[85]提出的函数型分析方法,为过程内的数据流和过程间的部分数据流建立了摘要.对于过程间边相关的未知量,即虚函数调用(virtual function)和回调点(callback site),该方法在计算过程中不予考虑.该方法只考虑了函数中的局部变量.StubDroid[86]在FlowDroid的基础上采用了构件级别分析对库进行了自动分析,支持多层字段敏感性,无需手动编写摘要.

Tang等人[87]提出了条件可达性,解决了传统摘要无法描述的回调函数问题.具体而言,该工作考虑了过程间的数据依赖分析.当代码库存在回调函数时,部分信息缺失导致已有方法难以为代码库建立有效的摘要.为了处理回调函数带来的影响,该工作提出了基于树-邻接语言的可达性的摘要分析技术.注意到包含回调函数信息的可达性关系(称为“条件可达性关系”)可用于加速用户代码的分析.为描述条件可达性关系,该工作引入了原本用于描述自然语言语法的形式语言——树-邻接语言,提出了树-邻接语言的可达性分析技术.树-邻接语言可达性扩展了表达两点间可达关系的传统上下文无关语言可达性,具有表达4个点之间的可达关系的能力,从而更自然地描述了上述条件可达性关系.实验结果表明:建立基于树-邻接语言的可达性摘要可以在合理的时间内完成,该技术所建立的摘要可以使得对用户代码分析的效率平均提高约8倍.在后续工作中[88],Tang等人进一步将该方法扩展到了Dyck-CFL可达性分析上.该方法通过直接在较为通用的CFL可达性分析过程[67]中引入带条件边,使得大量基于CFL可达性的分析就可以直接支持条件摘要.同时,该方法引入桥接边来避免条件组合爆炸的问题.通过这些处理,该方法避免了原有的Dyck-CFL分析中摘要不完整和分析效率不高的问题.实验结果表明,其分析速度相比树-邻接语言进一步提升了3倍多.

由于模块含有大量的未知量,建立考虑所有情况的摘要在许多分析任务中非常困难.所以,一些工作通过动态程序分析或训练的方式建立了部分摘要.Palepu等人[89]提出了加速程序分析速度的动态依赖摘要技术(dynamic dependence summary),提前创建软件库的摘要,用来快速创建程序切片.尽管得到的是不充分的摘要,导致生成的程序切片并不保证完全正确,然而在程序测试的实践中能够找到大部分的错误.Kulkarni等人[90]采用跨程序的训练办法为代码库建立不完整的摘要,以提高客户代码的分析速度.该方法基于Datalog语言,有效地对中间计算过程进行剪枝,并将剪枝结果记录于摘要中.同时,该方法还对Datalog语言进行改造,使得利用摘要进行分析时可以跳过中间计算过程.该方法保证正确性的方式是人工编写时间复杂度较低的规则,判断训练得到的摘要是否能够保证正确:如果不能保证正确性,则不采用摘要来进行计算.

上述工作都是针对整个代码库进行摘要分析.当精度要求较高时,分析完整的代码库比较困难.部分工作只对单个函数进行分析.Yorsh等人[91]提出的框架可以为函数建立精确而简练的函数摘要.在不同上下文环境中,采用函数摘要与重新分析函数所得到的最终结果相同,从而保证精确性.框架中采用有限显式输入输出表来表

示摘要,并通过摘要组合操作,使得不同上下文之下分析过程的相似之处得以合并为更加简练的形式,从而利用有限的数据结构表达无限可能的函数行为.该框架采用了微转换子,微转换子可以编码的问题包括IFDS/IDE问题.该框架还可以解决模块的线性常量传播问题和模块的类型状态[92]检验问题.

Dillig 等人[93]的工作是针对C的上下文敏感、流敏感、不考虑字段的函数指向分析.核心问题是分析一个函数时,如何考虑所有可能的调用上下文.解决方案是将函数参数所有可能的别名关系编码到一个二部图上,一边是参数,另一边是符号化的位置,表示彼此不交的内存地址集合,中间连上带约束的边,编码了不同的别名关系.当遇到语句的转移函数时则采用图上再加一层的方法来保证流敏感性,这样可以做到强更新.函数调用处理方面,需要计算出被调函数的别名情况,实例化主调函数的摘要,再把调用函数的图和被调函数实例化的图拼起来.通过不动点算法计算出一个上近似的结果.

2.4 符号执行

符号执行[94−97]是一种相对精确的程序分析技术.传统的符号执行技术使用符号化输入代替实际输入以模拟执行(不实际执行)被分析程序,程序中的操作被转化为相应的符号表达式操作.在遇到条件语句时,程序的执行也相应地分叉以探索每个分支,分支条件则被加入到当前路径的路径条件(path condition)中.通过调用SAT/SMT求解器,对路径条件的可满足性进行求解来加以判断:如果判定结果为可满足,则说明路径实际可行(存在具体输入能够让程序产生此路径);如果判定结果为不可满足,则表明此路径不可行,终止对该路径的分析.

在约束条件可被判定的情况下,符号执行提供了一种系统遍历程序路径空间的手段.符号执行中的程序路径精确刻画了程序路径上的信息,可基于路径信息开展多种软件验证确认阶段的活动,包括自动测试、缺陷查找以及部分的程序验证等[98].理论上,相比于需要固定程序输入的分析方法,符号执行通过符号分析,可覆盖更多的程序行为.另一方面,符号执行技术依赖于SAT/SMT技术,求解器的能力是决定符号执行效果的关键因素.符号执行中,程序路径空间大小随着程序规模的扩大而呈指数级增长.例如:单就串行程序而言,一个具有n个条件语句的程序段就有可能包含2^n条路径,这也是制约符号执行能力的关键因素.

最初,符号执行主要用于程序自动测试,但受制于当时的计算能力和求解技术与工具的能力,符号执行技术并没有得到实际的应用.近年来,随着计算能力的提高、SAT/SMT技术和工具的蓬勃发展[99−101],符号执行技术得到了长足的进步,出现了以动态符号执行(concolic testing)[102,103]为代表的一批新的符号执行方法或技术,以及以SAGE[104]、KLEE[105]、SPF[106]、Pex[107]、S2E[108]为代表的符号执行工具.

不同于传统的符号执行,动态符号执行[102,103]实际运行被分析的程序,在实际运行的同时收集运行路径上的路径条件,然后翻转路径条件得到新的路径条件,通过对新的路径条件进行求解得到新的程序输入以再一次地运行被分析程序,从而探索与之前运行不同的程序路径.由于在实际运行程序过程中,动态符号执行在碰到复杂或不可解的路径条件时,可以使用实际值来简化路径条件.在碰到外部调用时,也可以通过实际执行来缓解外部调用的问题.因此,在方法层面,动态符号执行通过牺牲部分分析的可靠性来提高方法的可扩展性和可行性.

目前,符号执行技术仍然面临提高可扩展性(scalability)与可行性(feasibility)这两方面的挑战:可扩展性挑战是指如何在有限资源(比如时间、内存)的前提下提高符号执行的效率,更快地达成分析目标;可行性挑战是指如何支持不同类型分析目标的符号执行,权衡分析的可靠性与精确性.已有的研究工作基本都是围绕这两个方面展开.

在可扩展性方面,路径空间爆炸和约束求解是主要的两大难题.在缓解路径空间爆炸方面,目前已有的工作基本分为两个思路:(1) 在具体目标下提供高效的搜索策略,使符号执行分析更快地达到目标,包括提高程序的覆盖率[105,109−111]、判断某个程序点是否可达[112]、产生满足正规性质的程序路径[113]、探索程序不同版本的差异部分[114,115]等;(2) 通过约束输入范围、削减或合并路径来减小程序的路径空间,包括基于输入模版[116,117]、程序切片[118−121]、程序抽象(包括摘要技术)[122−129]、偏序约减[130,131]、条件合并[132,133]以及等价路径约减[121,134−136]等一些方法.在约束求解方面,已有的工作也可分为两个方面:(1) 在调用求解器之前对路径条件的查询进行优化,以减少求解器的调用次数或缩短求解时间,包括查询缓存和重用[105,137−139]、基于约束独立性的优化[103,105]、增量式求解[105,140]等;(2) 支持复杂程序特征的高效编码,包括对机器数[104,105]、数组[100,105]、浮点[141−145]、字符

串[146]、动态数据结构[147−149]等方面的支持.

在可行性方面,环境建模和多形态分析目标的支持是目前的主要问题.在环境建模方面,已有的工作基本都是在分析的精确性、可靠性、建模工作量以及可扩展性之间进行权衡和折中,包括手工建模[105,106]、自动合成[150]、动态执行[102]、全栈执行[108]等.在多形态分析目标方面,主要是应对多语言和多应用领域的复杂性,包括二进制程序[108,151,152]、脚本语言程序[153]、分布式程序[154]、数据库操作程序[155]、无线传感器网络程序[156]、并发或并行程序[135,157,158]、嵌入式程序[159]、PLC 程序[160]等符号执行方法.

同时,面向新的分析需求,也有一些新的符号执行技术出现,其中比较有代表性的是概率符号执行技术[161−163],其基本思想是:通过符号执行来得到程序的路径,然后使用 SMT 解空间体积计算技术来计算每条路径的路径条件的解的个数,通过每条路径对应的解的个数,可以计算每条路径的概率.基于此,可以开展包括低概率缺陷查找、低概率路径的测试用例生成、生成程序的性能分布[164]、刻画代码变迁[165]等活动.

随着符号执行技术近些年的发展,符号执行技术在工业界也得到了实际的采纳和应用,其中有代表性的工作有:微软公司把自己开发的二进制动态符号执行工具 SAGE 用于 Win 7 的测试,发现了文件模糊测试中 1/3 的缺陷[166].由于 SAGE 通常是微软公司最后使用的缺陷检测工具,因此,这些由 SAGE 发现的缺陷,很多都没有被之前在开发过程中使用的静态分析工具、黑盒测试工具所发现.微软公司多个开发小组已把 SAGE 作为日常工具在使用,SAGE 目前也被商业化为微软公司的安全风险检测(microsoft security risk detection)服务.微软公司在 Visual Studio 2015 中正式发布了基于动态符号执行的 C#自动单元测试工具 IntelliTest[167],可大幅度提高 C#程序单元的测试效率.C 程序符号执行工具 KLEE[105]对 GNU Coreutils 程序集进行了自动测试,可自动达到 94%的语句覆盖率,并发现 3 个程序崩溃问题;Canalyze[168]在开源软件中发现数百个真实缺陷,并在工业界嵌入式系统中发现两个缺陷.2016 年 8 月,在美国 DARPA 举办的网络空间安全竞赛(CGC)中,最终排名前三的参赛队伍全部使用了符号执行技术,用于自动发现并利用二进制程序中的漏洞.美国 GrammaTech 公司开发的商业程序分析工具 CodeSonar 中也使用了符号执行技术,用以发现程序中的深层缺陷.

未来,符号执行技术将进一步在软件工程、安全、系统、网络等相关领域的实际需求驱动下不断发展.面向大规模软件的高效符号执行方法、技术和工具将是下一步研究所面临的挑战和重点.同时,符号执行搜索策略的更加智能化[169]也将是下一步的研究重点.此外,与其他技术在不同层面的密切结合[170−173],以进一步提高软件分析效果,也将成为符号执行后续的研究趋势之一.

2.5 动态分析

动态分析是指通过在指定测试用例下运行给定的程序,并分析程序运行过程或结果,用于缺陷检测等.与静态分析相比,动态分析能够更好地处理编程语言中的动态属性,例如指针、动态绑定、面向对象语言中的多态与继承、线程交替等.动态分析在一定程度上弥补了静态分析的不足之处.

基本的程序动态分析可以简单地分为在线(online)动态分析与离线(offline)动态分析[1].在线动态分析是指在程序的运行过程中分析当前程序行为;而离线动态分析需要记录程序的运行行为,在程序运行结束后再进行分析.两者的基本思想都是通过对程序运行过程或者结果的分析,查找定位缺陷.具体来说,一方面可以将程序运行结果和预知结果对比,确定程序中是否含有缺陷;另一方面,可以通过插桩或其他监控技术分析程序的运行行为,查找错误的行为.前者很直观,但是对于被触发了却没有反映在输出中的缺陷无法检测;后者则可以直接观察到程序中缺陷的触发,即使该次触发并没有导致错误的输出.但是后者需要提前定义错误的行为,其对于非常见的缺陷无法检测.已有文献中有大量这方面的介绍[174].

程序动态分析,从报告缺陷的准确性出发,也可以分为:(1) 缺陷动态检测;(2) 缺陷动态预测.一般所指的缺陷动态查找方法是缺陷动态检测,是指在程序的运行过程中某个缺陷已经发生了之后的检测,即,缺陷已经反映在程序行为中或者运行结果中.其关注点是:如何设计检测算法等,保证将所有已经发生的缺陷检测出来.然而,程序中的不同缺陷不会都在有限的若干测试用例中被触发.因此,为了提高缺陷动态检测的有效性,需要从程序若干次运行中预测出该程序潜在的某些行为,并判断这些潜在行为是否会触发缺陷.这种方法称为缺陷的动态预测.例如:某个程序中存在一个缓冲区溢出缺陷,该缺陷只有在输入 input 大于 128 时发生.那么,如果我们可以

从某个 input 不大于 128 的输入下预测对应行为是一个潜在缺陷,其可能会在 input 大于 128 时发生.进一步地,我们可以构造一个满足预测条件的输入,再次运行程序去触发(验证)缺陷.缺陷的动态预测在并发程序的动态分析中经常用到,主要原因之一是:并发程序的运行除了与输入有关外,还与程序中线程之间的调度有关.程序缺陷的动态预测还会涉及到一些其他技术,例如,测试用例生成、约束求解、缺陷重现、线程调度等,其分析过程与缺陷动态检测相比更为复杂.

2.6 基于机器学习的程序分析

经典的程序分析技术提供相对精确的分析结果,但同时也带来了包括路径组合爆炸、误报率较高等一系列在实践中不可避免却难以应对的问题.随着近年来通用计算设备能力的提高,海量的程序执行数据被存储和管理;研究者采用机器学习、统计分析等系列技术提升现有的程序分析能力.

现有的程序分析技术依赖于一定的启发式策略,如符号执行技术中的深度(或广度)优先搜索.在动态分析技术中,为了达到分析精度,这些策略可能带来较高的计算成本.基于现有的标记数据(即已知分析结果的程序)建立学习模型,机器学习技术可以学习新的可自适应的策略,减少启发式策略带来的成本消耗.Li 等人[144]为了解决符号执行中的路径可达性问题,以最小化不满足性为目标,建立机器学习模型 MLB,以减少经典的约束求解方案在求解非线性约束及函数调用时的不足.Kong 等人[175]面向自动机验证,研究了多个随机测试与符号执行技术的整合策略,通过调整状态转移概率优化动态符号执行.Wang 等人[172]提出了基于马尔可夫过程的动态符号执行方法,以达到应用符号执行获得精确分析结果和应用随机测试覆盖搜索空间的平衡.该方法采用贪心算法获得优化模型的解,以期找到对应于近似最优性能的策略.Chen 等人[176]针对信息物理系统(cyber-physical system)的攻防模型,设计了基于程序变异的方案,获得具有植入错误的模型训练数据以避免人工建立模型的巨大成本.Xiong 等人[177]提出了基于概率程序合成框架 L2S.该框架整合了包括程序合成的搜索空间、路径和潜在解的概率估计方案,能够按需地构建程序合成和修复技术,研究者可基于该框架深度定制和设计相关方法.

在静态分析技术中,分析工具在获得较高的分析精度的同时,往往带来过高的误报率.研究者建立了机器学习模型,以剔除潜在的误报并减少静态分析的成本.Heo 等人[178]提出了基于异常点检测技术来滤除静态分析结果中的误报.该方法提取代码循环和函数调用中的特征,基于训练数据建立学习模型,识别潜在的误报并提升静态分析的实用性.Chae 等人[59]针对上下文敏感的指针分析,提出了基于分类器的自动特征抽取方案.该方案可有效提升现有的静态分析技术.Oh 等人[179,180]针对静态分析的精度和计算成本,提出了基于贝叶斯优化的自适应学习方案,并以此建立面向数据流和上下文的部分静态分析工具.Jeong 等人[181]设计了数据驱动的上下文敏感的指针分析方案.该方案通过非线性的上下文选择,建立了识别上下文敏感函数的学习方法.

3 面向特定软件的程序分析技术

本节介绍程序分析技术在一些重点领域软件的应用,包括移动应用软件、并发软件、分布式系统、二进制代码等方面的重要应用.

3.1 移动应用软件

近 10 年来,以智能手机为代表的智能终端以惊人的速度得以普及.目前,以 Android 和苹果 iOS 系统为代表的智能手机数量和使用频度已远超个人计算机,成为最流行的个人电子设备.通过移动支付、购物和社交等各式各样的移动应用,智能手机已经深度渗透进了人们的日常活动中.相应地,这些应用的运行状态和广大用户的日常生活和工作有着直接的关系,也在很大程度上影响到了整个互联网的运转.为此,研究人员通过对移动应用软件的程序分析,来检测其是否具有所期望的以安全性为核心的各种特性.

1) 污点分析技术

由于智能手机的使用特点,移动应用的安全性分析常常可以归结为应用代码上跟踪敏感数据流的动态/静态污点分析(taint analysis)问题.

- 动态污点分析:最具代表性的 Android 应用动态污点分析技术是 TaintDroid[182],其通过修改的 Dalvik

Java 虚拟机,在应用的 Java 字节码的解释执行过程中进行动态插桩,以实施对敏感数据的跟踪分析.在 TaintDroid 的基础上,研究人员还研发出了其他一些应用安全性分析和防护系统,如 AppFence[183]和 DroidBox[184]等.Yan 等人开发了一个基于全系统虚拟化的动态污点分析平台 DroidScope[185],构建于 CPU 模拟器 QEMU[186]之上,发展自面向桌面平台的污点分析系统 TEMU[187].DroidScope 可在 CPU 指令模拟执行层面对运行于模拟器上的整个系统(包括 Android 应用和操作系统)中的信息流进行跟踪,但在较低层面实施污点分析不可避免地带来一定的语义鸿沟(semantic gap),从而影响到污点分析工作的精度;

- 静态污点分析:FlowDroid[66]是影响较大的基于Android的静态污点分析工具.FlowDroid基于过程间控制流图进行静态的Jimple代码模拟执行,根据Jimple指令的语义跟踪敏感数据在潜在执行路径上的传播,从而检测分析目标应用中可能存在的隐私泄露等危险操作.类似地,在 Android 应用 Java 字节码层面进行静态污点分析的系统还有 AndroidLeaks[188]和 Apposcopy[189]等.为了更精确地分析 Android 应用中的敏感信息流,DroidSafe[190]对 Android 底层系统进行了建模,将其表示为与应用开发语言相匹配的 Java 程序,从而跟踪分析涉及到系统底层的信息流.Jin 等人[191]针对采用 JavaScript 来实现应用逻辑的 HTML5 混合型移动应用,设计实现了一种 JavaScript 代码注入漏洞的静态检测方法,所采用的核心技术也是静态污点分析.针对苹果手机应用,PiOS[192]通过对 iOS 应用的 Mach-O 二进制可执行文件进行静态数据流分析来检测应用是否有泄露隐私的行为.

2) 面向移动应用特性的程序分析技术

与桌面应用的分析相比,移动应用的分析还会涉及到一些与移动平台特性密切相关的分析任务,如组件间通信(inter-component communication)的分析[193]、电量过度消耗等资源泄露(resource leak)的检测[194]、权限泄露(capability leak)[195]以及权限重代理(permission re-delegation)[196]的检测防御等.

3) 移动应用分析辅助技术

为了达到较好的分析效果,仅仅关注于应用软件分析技术本身是不够的,研究人员还发展出一些分析辅助技术来进一步提升分析效果.为了在动态分析工作中获得较高的覆盖率,学术界研究出一些系统化的应用运行驱动技术,如 AndroidRipper[197]、Dynodroid[198]、AppDoctor[199]、EvoDroid[200]和 TrimDroid[201]等.为了较为方便地在真实 Android 手机环境中部署分析机制,You 等人提出一种称为引用劫持(reference hijacking)的技术[202],可在不刷机、不 ROOT 设备的情况下,将动态污点分析机制植入到底层系统库中.与桌面平台相比,移动平台具有更为复杂的权限管理机制.PScout[203]使用静态分析从 Android 系统源代码中抽取出应用编程 API 所对应的权限规范,从而为 Android 应用的安全分析提供了重要的支持.Android 应用中存在有大量的隐式调用,导致静态构建函数调用图是一个非常具有挑战性的任务.EdgeMiner[204]通过静态分析 Android 的 Framework 层代码来生成各个 API 所导致的隐式控制流转移的函数摘要,能够被集成在已有的静态分析工具中以提高 Android 应用分析的精度.对应用进行污点分析需要明确知道引入敏感数据的 Source 点和会导致危险操作的 Sink 点,但现实中缺乏一个完全的 Source/Sink 点规范.Rasthofer 等人[205]利用机器学习技术设计了一个支持向量机分类器以自动识别 Android 系统 API 中未知的 Source 点和 Sink 点.为了获得可供分析的代码,研究者还研发了一些从加壳后的 Android 应用中抽取 Dex 代码的工具,代表性的有 DexHunter[206]和 PackerGrind[207].这些分析辅助技术对提高移动应用分析工作的效能具有重要的意义.

3.2 并发软件

自从计算机进入并发处理时代,系统效率得到显著提升.然而,并发程序(如多线程程序)的使用,导致并发问题(或者并发缺陷)的存在且难以解决.其难点是:并发程序在运行时,多个任务之间的交替运行空间巨大,而搜索该空间是 NP 难的.2007 年图灵奖、2013 年图灵奖、2016 年哥德尔奖等都授予了为解决并发问题而做出突出贡献的学者.他们提出的方法,例如模型检验[208]工具,虽然已成功用于很多并发程序(算法)的验证,但是仍然无法应用到大规模真实程序中,从而失去了实际意义.2000 年以来,多核处理器的快速发展使得大规模多线程程序被广泛使用,尤其是近几年来,大数据、云计算、高性能计算等产品的应用中,都使用了大量的基于线程的高并

发处理,这进一步加剧了并发缺陷的严重性.

由于多线程程序运行时各个线程之间的交替执行(interleaving)[209−212]使其具有不确定性,传统的单线程程序的测试方法无法用于测试多线程程序以找出并发缺陷.即使某个并发缺陷被检测到一次,也很难被再一次检测到或重现这个缺陷[213].因此,多线程程序中并发缺陷的检测变得比较困难.也因此,近年来并发缺陷的相关研究也非常热门.

并发缺陷的检测包括静态分析[214−216]、动态分析[217−221]以及二者结合的混合分析[222].静态分析主要是检测程序代码(源代码、中间代码和二进制代码等)中是否存在特定模式(即那些可能导致并发缺陷发生)的同步以及资源的访问.虽然静态分析能够通过分析给定程序中所有的代码来达到很大的覆盖面,但是由于缺乏程序运行时的信息,尤其是多线程程序特有的交替运行信息,其检测结果是非常不准确的[223].动态分析则是依赖于程序的具体运行去检测并发缺陷,其检测结果相对静态分析会准确很多.但是一个多线程程序的运行依赖于其中所有线程的交替运行,而这种交替运行的空间是非常大的.因此,单次或有限次多线程程序的运行很难覆盖该程序的所有其他运行情况[224].从而动态分析只能找到给定程序运行中的缺陷,包括潜在的缺陷.另外,动态分析会引起程序运行时的时间开销,例如,在 C++程序上动态分析数据竞争时,其时间开销很容易达到原有程序运行的 100 多倍[225].混合分析则是结合了静态分析和动态分析各自的优点,其首先通过静态分析找出所有潜在的缺陷,然后通过动态分析去验证这些潜在的缺陷是否为真实的缺陷[222].但是,这种分析方法又受制于测试用例的生成.亦即:即使静态分析中检测到的一个潜在的缺陷是真实缺陷,但是怎样通过动态分析去验证它.因为一个缺陷的发生不仅取决于程序运行过程中线程之间的交替,还取决于运行该程序时相应的输入是否可以导致该缺陷的发生[226].另外,如果潜在缺陷的数量非常庞大,则其需要大量地运行给定的程序才能确认每个潜在的缺陷.

1) 全面调度技术

动态分析依赖于程序的具体运行来检测并发缺陷,因此其无法检测那些没有运行或者被隐藏的并发缺陷.引入随机性则可以增加动态分析的检测能力.ConTest[227]在程序运行时随机加入噪音(例如随机休眠某个线程一小段时间),使得一个程序在每次运行时,各个线程之间的交替产生差异.然而,这种差异只能导致很少的一部分并发缺陷被检测到,无法让那些很难检测到的并发缺陷被检测到.

PCT[228]和 RPro[229]是最新的两种有概率保证的调度技术.PCT 可使一个程序按照事先产生的随机调度运行,而这些随机调度在概率上可以保证动态分析检测到每一个并发缺陷.但是 PCT 需要非常多的运行次数,因为其概率的保证是非常小的.因此,在规模稍大的程序上,PCT 效果很差.RPro 提出了缺陷半径的概念,并结合实际,使其每次产生的调度只会在特定的程序运行范围内,从而极大地提高了缺陷的检测概率.从另一个角度来说,PCT 是 RPro 以程序运行中所有事件数量为半径时的一个特例.

2) 限定调度

完全探索一个并发程序所有可能的运行空间是 NP 难的.近年来,研究人员提出了限定调度技术来缓解这一难题.CHESS[213]只完全遍历程序中引发不确定运行的事件,包括线程同步以及对 volatile 变量的访问等,且其只限定在给定的有限个调度点之内的事件.SKI[230]也采用了类似的思路,在系统内核中通过遍历未知运行来找到并发缺陷.Maple[231]通过定义一些已知的模式来预测程序运行中的潜在并发错误.但这类工作在实际中的效果也不是很好.例如,一项最新的研究表明[232],这些工作(包括部分全面调度技术)还是会漏掉很多并发缺陷.

3) 启发式调度

并发缺陷主要包括死锁(deadlock)、数据竞争(data race)和原子性违反(atomicity violation)[233].不同的缺陷通常会有一些特别的检查方法.

针对数据竞争检测,FastTrack[234]舍弃了全面内存访问追踪,在保证可以在每个内存上检测到第 1 个竞争的条件下,改进了检测效率.DrFinder[235]通过预测并反转同步的方法,提高数据竞争检测的覆盖面.CP[236]通过重新定义数据竞争的方法在一次运行中检测到更多的数据竞争.但是这种方法只能保证在一个给定程序中只有一个数据竞争时是正确的.RaceMob[222]通过在程序的用户端来确认给定的潜在数据竞争哪些是真实的,但其仍然需要静态分析来首先找到潜在的数据竞争缺陷.为了解决动态分析的时间开销,各种采样的方法被用于数据竞

争的检测中.LiteRace[237]首先通过降低一个程序中经常被调用的方法的监控来减少时间开销.Pacer[238]采用定期采样的方法,使得采样率和数据竞争的检测率保持线性关系.Carisma[239]进一步对 Java 程序中同一代码产生的变量进行采样,其在采样率很低时,数据竞争的检测效果较好.采样分析虽然减少了时间开销,但也降低了数据竞争的检测数量,并且无法克服动态分析漏报的缺点.DataCollider[240]通过静态采样(代码级别的采样)来检测数据竞争.其不但保证采样率和数据竞争检测率的线性关系(与 Pacer 类似),而且时间开销比 Pacer 更少.但是 DataCollider 只能检测到特定类型的数据竞争,即:两次访问中,写操作数和另外一个读或者写操作数不一致的数据竞争.RaceFuzzer[241]在每一个潜在数据竞争发生的位置去调度(例如主动暂停)相关的线程来判断这个潜在的数据竞争是否会被触发.只有那些被触发的数据竞争才会被报告出来.这种方法在一定程度上会检测到更多的数据竞争,但其检测能力仍然受制于所使用的调度算法.简单的调度并不会增强数据竞争的检测能力,而复杂的调度很难开发.这一点不仅仅针对于数据竞争的检测,在死锁等的检测上也是如此[217,242].Huang 等人[243]最近提出了干扰程序运行时的控制流来检测到更多的数据竞争的方法.这种方法需要针对每一个潜在的数据竞争来解决相应的约束条件,使其在潜在数据竞争很多的时候,无法快速地解决相应的约束条件.CRSampler[244]采用了一种新型的针对硬件采样的数据竞争定义方法,使其可以在时间开销很低的情况下有效地检测已部署程序中的数据竞争.AtexRace[245]则是一个针对跨运行和跨线程的数据竞争采样技术,在大量测试用例条件下,其可以减少整个测试的时间开销.

原子性违反检测的难点,首先是如何确定程序中的内存访问应该是原子执行的.一旦原子性区域被识别,其违反的检测就会比较容易.AVIO[246]通过从正确运行的程序中检测单个变量上两条访问指令上的原子性区域.MUVI[247]进一步考虑多变量的内存访问中的原子性区域.AtomFuzzer[248]直接假设每种方法中的内存访问是原子性的.AtomTracker[249]首次提出了不需要原子性标记且针对任意原子性区域、任意变量数下的原子性区域检测方法.但是,AtomTracker 需要运行一个程序多次,且不能含有错误运行.

死锁不同于数据竞争和原子性违反,其涉及到线程之间的同步.同样,死锁一旦发生,则其很容易被检测到.但是如何从一个没有死锁发生的程序中预测潜在死锁,同样是并发程序分析中的一个难题.其中的主要难题是如何高效地检测死锁.iGoodLock[242,250]提出了锁依赖关系来表示程序中每个线程对锁的获取行为,并基于这些关系的集合来检测死锁.然而,在检测过程中,由于需要生成大量的中间表示关系,iGoodLock 会消耗很多内存,导致其效率低下.MagicLock[250,251]被认为是死锁检测方面截止目前最快的检测方法.其将锁依赖关系针对线程来分析存放,且提出了锁依赖的等价关系与非等价关系以及一系列优化措施,使得死锁的检测效率大为提高.

4) 消除误报与漏报

并发缺陷的检测通常会有误报[217,241].近年来,主动调度技术(active scheduling)被广泛提及,并用于误报消除.RaceFuzzer[241]和 AtomFuzzer[248]通过在有潜在缺陷的点插入调度点,并通过随机调度来触发预测缺陷.DeadlockFuzzer[242]采用了类似的想法来触发死锁.这些技术都受制于 Thrashing,使其触发概率不高.Conlock[217,252]和 ASN[253]通过产生约束以及并发缺陷触发的必要条件来触发死锁,在一定条件下,这两种技术可以保证触发真实死锁.

另一方面,动态分析必然产生漏报,其中一个原因就是缺乏并发测试用例.ConTeGe[254]针对 Java 库程序,提出了一种简单的包含两个线程且针对单个共享变量的并发测试用例生成技术.CovCon[255]进一步考虑了并发覆盖度,从而针对那些很少被覆盖的代码来生成(选择)更多的测试用例.

3.3 分布式系统

随着越来越多的数据与计算从本地向云端迁移,大规模分布式系统逐步得到广泛使用,比如分布式存储系统、分布式计算框架、同步服务、集群管理服务等.典型的大规模分布式系统包括 HDFS、Hbase、Hadoop、Spark、ZooKeeper、Mesos、YARN 等.与传统的单机系统相比,大规模分布式系统具有较好的可扩展性和容错能力,获得同样计算能力的成本较低.因此,分布式系统在应对复杂业务场景以及单机无法完成的大规模计算任务时具有较大的优势,已成为支撑大规模网络应用不可或缺的一部分.大规模分布式系统已在大型互联网公司,如阿里巴巴、谷歌、百度等得到广泛应用.

大规模分布式系统必须管理大量分布式软件组件、硬件及其配置,使得该类系统异常复杂.因而,大规模分布式系统不可避免地会发生故障,并影响到大量终端用户,降低了它的可靠性和可用性[256].例如:2013 年 1 月 10 日,由于一个客户端与服务器的同步错误,Dropbox 故障超过 15 个小时;2014 年 6 月 23 与 24 日,Microsoft Lync 与 Exchange 相继故障,导致其部分用户 9 小时无法访问邮件系统;2017 年 2 月,亚马逊 AWS 服务宕机,造成许多基于亚马逊云服务的互联网应用无法正常工作.因此,分布式系统中缺陷理解、检测及验证技术成为当前研究的一个重点.

1) 分布式系统中缺陷的实证研究

近年来,为了提高复杂分布式系统的可靠性,研究人员对分布式系统中的各种缺陷进行实证研究以加深对这些缺陷的理解.CBS[257]针对 6 个开源分布式系统(Hadoop MapReduce、HDFS、Hbase、Cassandra、ZooKeeper 和 Flume)中 3 655 个致命缺陷进行综合性分析研究,详细总结了分布式系统中出现的各种缺陷类型,包括可靠性、性能、可用性、安全性、可扩展性等方面的缺陷,并形成一个开放的分布式系统缺陷数据集.但是,CBS 没有对分布式系统中不同类型的缺陷进行深入分析,比如缺陷模式等.因此,后续研究对分布式系统中特定类型的缺陷进行深入分析.TaxDC[258]深入分析了 4 个开源分布式系统(Cassandra、Hadoop MapReduce、HBase 和 ZooKeeper)中的 104 个并发相关缺陷.Dai 等人[259]对 11 个常用的云服务系统中 156 个超时(timeout)相关缺陷进行了分析,总结了若干超时相关缺陷模式.CREB[260]针对 4 个开源分布式系统(Cassandra、Hadoop MapReduce、HBase 和 ZooKeeper)中的 103 个与节点失效恢复相关的缺陷进行了深入分析,总结了分布式系统中若干节点失效恢复相关的缺陷模式.Guo 等人[261]对分布式系统中的恶性失效恢复行为进行了归类说明,认为失败恢复应该遵守无害准则.Yuan 等人[262]对发生在 Cassandra、Hbase、HDFS、Hadoop MapReduce 和 Redis 上的 198 个系统失效进行了深入分析,理解分布式系统中的故障最终会演变为用户可见失败的模式.Wang 等人[263]详细分析了几十万台服务器上的 290 000 个硬件失败报告,发现了若干硬件失效模式.上述实证研究使得研究人员对分布式系统中不同类型的缺陷得到深入理解,为进一步分析、检测相关缺陷提供支撑.

2) 基于动态/静态分析的分布式系统缺陷检测

由于分布式系统的复杂性,很少有动态/静态分析工具直接检测分布式系统特有的缺陷,比如由于消息引起的并发缺陷、节点失效引起的缺陷、超时相关缺陷等.得益于近年来对分布式系统中缺陷的深入分析,开发人员构建了一系列新的开发工具.DCatch[264]将分布式系统中并发缺陷归结为节点本地的内存访问冲突问题.DCatch 通过记录一次正确的执行中的关键事件(节点交互消息),建立起分布式系统中事件之间的偏序关系,静态地对这些事件进行分析,识别并发的内存访问冲突,发现可能的分布式系统并发缺陷.FCatch[265]识别在发生节点失效情况下可能的冲突操作,自动地预测与节点故障事件相关的缺陷.Aspirator[262]基于发现异常处理错误的缺陷模式,开发了一个基于规则的静态检查器,发现不恰当的异常处理机制导致的系统失效.D3S[266]通过在运行时检查系统是否违反开发者指定的分布式属性断言来发现可能的问题,并提供导致问题的状态序列来帮助开发者更快地解决问题.Xu 等人[267]结合源代码分析和信息检索来解析控制日志,构建复合特征,然后使用机器学习的方法来分析这些特征,自动地检测系统运行中的问题.Dinv[268]利用静态和动态的程序分析方法来自动地推断分布式系统中不同节点上变量之间的关系,从而帮助开发者揭露系统在运行时应当满足的属性.为了简化失效重现场景,DEMi[269]通过动态偏序推理[270]和 Delta debugging[271]来自动地削减分布式系统的错误执行序列,从而定位错误.

3) 分布式系统验证与模型检验

大规模分布式系统由数量众多的计算节点组成,运行不同的复杂协议.建立可验证的分布式系统是避免错误的一个重要手段.Verdi[272]、IronFleet[273]和 Chapar[274]通过利用证明框架 Coq 和 TLA 来建立可验证的分布式系统协议.IronFleet 结合 TLA 风格的状态机精炼和霍尔逻辑验证来构建实际的且能被证明正确的分布式系统.Verdi 通过对各种网络语义和不同的故障进行规范化,使得开发者在验证分布式系统实现时可以选择最合适的故障模型.比如,首先在一个理想化的故障模型上进行验证,再将得到的正确性保证转移到一个更加现实的故障模型上.Chapar 利用 Coq 和 Ocaml,模块化地验证 Key-value 存储实现及客户端程序的因果一致性.但是,通过验

证方法,以较小的性能开销来构建可验证的、真实的大规模系统还有一定困难.当前的研究也发现,形式化验证过的分布式系统实现依然不是完全可靠的[275].

模型检验是分析现存分布式系统可靠性的重要手段.MODIST[276]系统化分析分布式系统在所有可能事件下的响应.利用模型检验的方法检测分布式系统往往需要应对巨大的状态空间.DEMETER[277]利用分布式系统的模块具有良好定义接口的特性,对接口行为进行抽象,从而大大缩减了状态空间.SAMC[278]拦截分布式系统中的不确定性事件并交换它们的顺序.SAMC 采用灰盒测试技术,在传统黑盒模型检验的基础上加入分布式系统的语义信息,从而缩减了状态空间,尽可能地避免模型检验中的状态爆炸问题.SDE[279]开发了一种新的算法,能够显著地消减测试中的冗余状态,使得对分布式系统进行可扩展的符号执行成为可能.

4) 基于失效注入的分布式系统分析技术

在分布式系统中,计算节点可能由于硬件、断电、操作系统错误等发生失效.分布式系统需要从节点失效中正确恢复,以保证系统的正确性.但是由于节点失效时机、失效恢复过程难以测试,研究人员开发了一系列工具分析分布式系统的失效恢复行为.SETSUD[280]利用对系统内部状态的感知来精确地控制扰动的时序,暴露分布式系统中系统层面的缺陷.PreFail[281]是一个可编程的失败注入工具,允许用户自定义失败注入策略.FATE and DESTINI[282]通过避免测试相同的恢复行为来尽可能地测试多种多样的失败场景.用户可以通过 Datalog 来描述故障测试方法以及分布式系统恢复规范,从而系统化测试分布式系统中故障恢复逻辑.PACE[283]通过系统地生成和探索分布式系统执行中可能产生的文件信息,来检测与所有副本同时失效相关的分布式系统漏洞.CORDS[284]通过注入文件系统错误检测分布式系统的失效恢复能力.MOLLY[285]采用谱系驱动的故障注入方法来发现故障容忍的分布式数据管理系统中的缺陷.

3.4 二进制代码

二进制分析是经久不衰的研究话题.尽管越来越多的程序是用解释型语言(Python、JavaScript 等)编写,但是它们仍然需要二进制程序来解释执行,或者通过即时编译(JIT)技术转换为二进制代码执行.另一方面,传统的操作系统等对性能要求较高的应用仍然是以 C/C++等编译型语言编写.此外,物联网中大量设备的计算资源有限,运行的都是 C 等语言编写的二进制程序.再者,从安全的角度来讲,二进制程序中可能引入源码中不存在的安全问题,例如 Thompson 提出的编译器后门问题[286]、Xcode Ghost 污染事件[287]等.

二进制分析的首要任务是反汇编,即识别二进制程序中的代码和数据,解析函数间调用关系,及函数内的控制流图.最直接的反汇编方法是线性扫描[288],通过逐条指令解码的方式恢复代码.更精确的方式是递归遍历[289],根据指令的语义寻找下一条指令的位置并解码.但是递归遍历面临着一个巨大的挑战:静态分析无法准确识别间接跳转指令的跳转目标.Cifuentes 等人[290]基于程序切片技术将间接跳转表进行规范化表示,根据一些启发式特征识别间接跳转语句的目标.Kinder 通过在(不完整)控制流图上进行数据流分析,进而完善控制流图,再迭代式地进行数据流分析,逐步恢复程序的控制流图[289,291].Xu 等人[292]通过动态分析识别间接跳转的目标,并采用强制执行的方式驱动程序探索所有路径,从而构建相对完整的控制流图.随着人工智能技术的发展,研究人员也将深度学习技术应用到反汇编中,例如,Shin 等人[293]通过 RNN 识别二进制程序中的函数边界.然而,反汇编仍然是一个开放的难题,现有的方案仍然存在很多局限[294].

二进制分析面临的另一个难题是高级语义恢复.恢复二进制程序语义可以用于多种安全应用,例如漏洞挖掘和安全防护.与源码程序不同,二进制程序中大量信息缺失,例如函数和变量名、函数类型、数据结构定义、虚函数调用与类信息等.Chua 等人[295]采用自然语言处理类似的技术识别二进制程序中的函数特征(参数类型及个数).Cifuentes 等人[296]通过切片技术提取函数调用指令的操作数的规范化表示,根据启发式特征识别虚函数调用点.Reps 等人[297]通过识别程序中静态已知的全局地址、栈偏移等识别全局变量和栈变量,通过数据流分析识别间接内存读操作的返回结果等,实现对二进制程序中的内存访问操作语义的识别.Jin 等人[298]通过过程间数据流分析,跟踪 this 指针的流向,识别候选的类成员函数及变量,从而恢复二进制程序中的 C++对象.Lin 等人[299]提出了一种基于动态分析的数据结构识别方法,对执行过程中的变量赋予时间戳标签,进而进行前向和后向传播分析,根据标签传播到已知类型的使用位置,反推原始变量的类型和成员布局.Lin 将同样的思想应用到

内核二进制代码中,可以识别内核中的对象[300].

二进制程序分析通常需要对二进制程序进行代码插装或改写.主流有 3 类二进制插装方案:在原始二进制程序中直接静态修改、将二进制程序提升到中间表示中进行修改或者在代码执行过程中动态修改.第 1 类方案,静态二进制插装,面临的最大挑战是反汇编的准确率,基于不准确的反汇编结果进行代码插装可能导致程序执行异常.Smithson 等人[301]提出了一种兼容的方案 SecondWrite,通过保留未知的代码段以及对间接跳转指令进行保守的指针转换,确保二进制程序改写的正确性.UROBOROS[302,303]和 Ramblr[304]方案则通过将二进制程序反汇编,进而在汇编码上进行插装,最后再组装生成新的二进制程序的方式进行二进制改写.第 2 类方案是通过将二进制程序提升到中间表示 IR 上进行修改,这类方案的优点是插装在 IR 上完成,从而与二进制的指令集无关.Song 等人提出的 BitBlaze 平台[305]的 VINE 模块通过将二进制程序转换为 VEX 中间表示完成分析和插装.Brumley 等人提出的 BAP 平台[306]基于 VINE 提出了一个新的中间语言.第 3 类方案是在程序运行时插装,通过受控的执行环境,在目标基本块、函数等执行之前进行插装,经典的方案包括 DynamoRIO[307]、Dyinst[308]、Valgrind[309]以及 Intel PIN[310].

二进制分析是许多安全分析的基础,具体的安全应用场景包括漏洞挖掘、恶意代码识别、安全防护等.在漏洞挖掘方面,二进制分析方案通常通过匹配漏洞模式的方式来挖掘漏洞.Machiry 等人[311]通过静态分析扫描驱动代码,根据漏洞模式识别未知驱动漏洞.Wang 等人[312]提出了双取漏洞的更精确的模型,进而采用静态分析在内核中发现了多个双取漏洞.Dewey 等人[313]基于静态分析识别虚函数调用点,进而基于可达性分析识别虚函数表溢出漏洞.在恶意代码识别方面,大部分方案也是通过匹配恶意行为特征实现检测.Feng 等人[189]通过静态分析组件间的调用关系,与恶意代码的特征进行匹配,从而识别 Android 恶意代码.Kruegel 等人[314]通过静态分析内核接口,识别正常调用所访问的内核内存,与目标模块所访问的内核内存进行对比,从而发现内核 rootkit.Bergeron 等人[315,316]通过采用摘要技术和符号替换技术,比对目标代码的特征与已知恶意代码的抽象表示,从而识别混淆的恶意代码.在安全防护方面,主流的二进制方案是通过二进制改写部署新的安全策略.Padraig 等人[317]通过二进制改写工具 SecondWrite 对二进制程序植入众多经典源码层的防御方案,可以有效地保护历史遗留的二进制代码.Wartell 等人[318]通过静态改写二进制程序,在程序加载时对代码位置进行随机化,能够有效缓解部分攻击.他们提出了另外一个方案[319],通过对所有 API 调用点植入安全检查,自动引入安全监控器来增强二进制程序安全性.Zhang 等人[320]通过反汇编二进制程序,并植入控制流完整性安全策略,可以有效地缓解控制流劫持攻击.Batyuk 等人[321]利用静态分析评估了 Android 应用中的恶意行为比例,并通过二进制改写缓解应用中的恶意行为.Zhang 等人[322]通过静态分析识别虚函数调用点,进而通过二进制改写部署安全策略,极大地缓解了虚函数劫持攻击.

4 讨论与展望

程序分析是一项重要的基础性技术,它不仅能够直接用于发现各类软件中的缺陷、改进软件质量,还可以在软件测试、调试、维护以及缺陷预测等方面发挥作用,包括测试数据生成、软件维护与程序理解、程序修复等.例如,软件测试的一个重要问题是:如何自动构造比较合适的测试数据,从而达到一定的测试覆盖率.Xu 和 Zhang[323]针对 C 程序的单元测试,采用符号执行、约束求解与线性规划等技术,自动构造较小的测试数据集.在软件调试方面,Wu 等人提出了 ChangeLocator[324],在控制流分析和程序切片的基础上,根据软件崩溃报告自动识别出引发崩溃的代码变更(crash-inducing changes),从而帮助开发者较快地理解软件崩溃的原因并找到解决方案.在自动化程序修复方面,Gao 等人[325]基于前述数据流分析技术,通过建立合适的抽象域,自动修复程序中的内存泄漏并保证安全性;Xuan 等人[326]基于动态分析提取实时运行值,将程序合成转换为约束求解问题,并应用 SMT 获得可修复程序的代码补丁.在缺陷预测方面,Briand 等人[327,328]利用程序分析技术提取面向对象程序中的内聚性和耦合性等特征,构建定量模型,预测可能包含缺陷的类.

随着新型软件形式的出现,程序分析技术也面临一些新的问题.下面针对新兴的智能合约及机器学习软件上的程序分析进行讨论.

4.1 面向智能合约的程序分析

最近数年,源自比特币(bitcoin)的区块链(blockchain)技术已成为各行各业关注的焦点.区块链可被认为是一种分布式、去中心化的计算与存储架构,除了支持数字资产外,还能被用于商品溯源、信用管理乃至游戏等各种各样的去中心化应用(decentralized application,简称 DApp).为了方便 DApp 的开发,一些基于区块链技术的区块链平台应运而生,其中最为引人注目的是以太坊(Ethereum)[329].以太坊被认为是一种可编程的区块链,用户可以在其上编写和部署由智能合约(SmartContract)组成的 DApp.目前,以太坊智能合约是最流行的 DApp 模式.

智能合约由 Solidity 语言开发,被编译为字节码后在以太坊虚拟机 EVM 上运行.由于智能合约可能承载着数额巨大的数字资产,自然也成为了黑客的攻击目标.针对智能合约的攻击已经造成了惊人的财产损失[330,331],研究人员开始探索通过软件分析技术来检测、验证智能合约的安全性.如在文献[332,333]中,符号执行技术被用于在字节码层面检测智能合约中的已知类型的潜在漏洞.此外,相对于传统软件,智能合约的体量较小(代码一般为数十至数百行),使得对其使用形式化技术成为可能,涌现出多个智能合约验证系统[334−339].这些系统根据用户给出的待验证属性或断言,采用模型检验等技术来验证目标合约的实现是否具有相关特性.与传统软件的分析相比,智能合约的分析在技术角度上并无太大不同,有的工作[334,338]甚至直接将智能合约代码转换为已有验证系统所能支持的形式,借助已有验证系统快速形成了对智能合约的分析能力.如在文献[338]中,智能合约的 Solidity 代码被转换为 LLVM Bitcode,从而可以利用已有的分析和验证工具来验证智能合约.但是智能合约的分析工作也具有一些特殊性,突出表现为部署后的智能合约升级维护较为困难,难以像传统软件那样随意打补丁,使得人们特别期望能够在部署前就发现所有的潜在问题,这对智能合约分析系统的分析效能提出了很高的要求.目前,智能合约分析工作面临的主要挑战是如何有效地提供分析所需的安全模式或属性,这往往会同时涉及特定领域知识和程序验证知识,仅仅依赖用户提供并不现实.此外,智能合约相关技术还处在进化中,可能会涌现出新的应用模式和新的安全问题,需要持续不断地跟踪归纳.如何高效地提取分析所需的先验知识,是一个非常有意义的研究问题.

4.2 面向机器学习软件的程序分析

近年来,机器学习及其他智能技术在行业软件中得到了广泛的应用,尤其是深度学习技术与工具在语音识别、图像检索、自动驾驶等领域取得了较大突破.然而,由于广泛存在的概率模型、多层传播的复杂网络结构、黑盒形式的用户接口等特性,深度学习工具的质量难以度量.现有的软件分析技术难以直接应用.研究者提出了一系列程序分析方案用以发现机器学习系统的潜在风险和程序错误,提升机器学习系统的质量[340].Qin 等人[341]提出了基于程序合成的机器学习系统的功能评估方法 SynEva,该方法通过学习场景的程序合成,用于评估机器学习系统.Sun 等人[342]提出了面向深度学习的动态符号执行方法,该符号执行方法将测试需求表示为量化线性运算(quantified linear arithmetic),以神经元覆盖为目标测试深度神经网络的鲁棒性.

在形式化验证方面,Gehr 等人[55]提出将卷积神经网络转化为 CAT 函数,再使用抽象解释技术对 CAT 函数的行为进行上近似,从而验证对应神经网络的局部鲁棒性.Wang 等人[56]设计了基于符号化区间的形式化验证方案,用来验证不适宜使用 SMT 的属性.Ruan 等人[343]针对深度神经网络,提出了基于可证明保障(provable guarantee)的可达性分析方法.Huang 等人[344]提供了基于 SMT 的深度神经网络的安全性验证方法.机器学习软件往往依赖于复杂的数据处理.针对数据量过大难以调试的问题,学术界也提出了若干方法.Ma 等人[345]提出了 LAMP,该方法用较低的开销在基于图的机器学习算法中记录追踪关系,以方便程序员定位缺陷.Gulzar 等人[346]设计了一系列的调试算子,用于在基于 Spark 的大数据分析程序中进行低成本的交互式调试,而不必反复重启整个计算过程.然而,由于机器学习技术的复杂特性,相对于传统软件的分析,机器学习软件的静态分析、动态分析等方面都鲜见成果.针对深度学习工具的程序分析仍在起步阶段.

5 结束语

程序分析是剖析复杂软件系统、提高软件质量的重要手段,是软件工程、程序设计语言、操作系统、信息

安全等领域日益受到关注的研究方向.经过多年的发展,程序分析已在多个方面取得了长足的进步.研究人员将程序语言理论与编译、人工智能、数据处理等技术相结合,针对不同的软件形态发展出众多程序分析技术,同时开发了高效率的自动化分析工具,其中有些工具帮助人们在开源软件中找到了很多缺陷、漏洞,有些工具在大公司里得到应用,并发挥了重要作用.由于篇幅所限,本文只是简述了该方向近期的一部分重要成果.

当前,程序分析技术还面临一系列挑战,比如准确性、可扩展性等.另外,新型的软件形态,如智能合约、深度学习等,也需要全新的程序分析技术.随着软件应用范围的发展以及对软件可靠性要求的提高,程序分析在软件开发、维护过程中起的作用将越来越大.

致谢 在本文写作过程中,日本九州大学的赵建军和国防科技大学的王戟提供了很多建设性意见,中国科学院软件研究所的苏静和刘晴帮助整理了部分资料,在此一并感谢.

References:

[1] 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).

[2] Nielson F, Nielson HR, Hankin C. Principles of Program Analysis. Springer-Verlag, 1999.

[3] Rice HG. Classes of recursively enumerable sets and their decision problems. Trans. of the American Mathematical Society, 1952, 74(2):358−366.

[4] Zhang J, Wang X. A constraint solver and its application to path feasibility analysis. Int'l Journal of Software Engineering and Knowledge Engineering, 2001,11(2):139−156.

[5] Cousot P, Giacobazzi R, Ranzato F. Program analysis is harder than verification: A computability perspective. In: Proc. of the CAV. 2018. 75−95.

[6] Li Z, Lu S, Myagmar S, Zhou Y. CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEEE Trans. on Software Engineering, 2006,32(3):176−192.

[7] Cousot P, Cousot R. Abstract interpretation: A unified lattice mode for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the POPL. 1977. 238−252.

[8] Cousot P, Cousot R. Systematic design of program analysis frameworks. In: Proc. of the POPL. 1979. 269−282.

[9] Jeannet B, Min A. Apron: A library of numerical abstract domains for static analysis. In: Proc. of the CAV. 2009. 661−667.

[10] Singh G, Schel MP, Vechev MT. A practical construction for decomposing numerical abstract domains. In: Proc. of the POPL. 2018. 1−28.

[11] Bagnara R, Hill PM, Zaffanella E. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming, 2008,72:3−21.

[12] Polyspace code prover. 2018. https://www.mathworks.com/products/polyspace-code-prover.html

[13] Astrée runtime error analyzer. 2018. https://www.absint.com/astree/index.htm

[14] Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B. Frama-C: A software analysis perspective. Formal Aspects of Computing, 2015,27(3):573−609.

[15] Logozzo F. Practical verification for the working programmer with code contracts and abstract interpretation. In: Proc. of the VMCAI. 2011. 19−22.

[16] Interproc. 2018. http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/

[17] Blanchet B, Cousot P, Cousot R, Feret J, Mauborgne L, Mine A, Monniaux D, Rival X. A static analyzer for large safety-critical software. In: Proc. of the PLDI. 2003. 196−207.

[18] Miné A, Delmas D. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In: Proc. of the EMSOFT. 2015. 65−74.

[19] Roux P, Garoche PL. Practical policy iterations-a practical use of policy iterations for static analysis: The quadratic case. Formal Methods in System Design, 2015,46(2):163−196.

[20] Jeannet B, Schrammel P, Sankaranarayanan S. Abstract acceleration of general linear loops. In: Proc. of the POPL. 2013. 529−540.

[21] Amato G, Scozzari F, Seidl H, Apinis K, Vojdani V. Efficiently intertwining widening and narrowing. Science of Computer Programming, 2016,120:1−24.

[22] Li Y, Albarghouthi A, Kincaid Z, Gurfinkel A, Chechik M. Symbolic optimization with SMT solvers. In: Proc. of the POPL. 2014. 607–618.

[23] Jiang J, Chen L, Wu X, Wang J. Block-Wise abstract interpretation by combining abstract domains with SMT. In: Proc. of the VMCAI. 2017. 310–329.

[24] Cousot P. Abstracting induction by extrapolation and interpolation. In: Proc. of the VMCAI. 2015. 19–42.

[25] Chen L, Liu J, Miné A, Kapur D, Wang J. An abstract domain to infer octagonal constraints with absolute value. In: Proc. of the SAS. 2014. 101–117.

[26] Chen J, Cousot P. A binary decision tree abstract domain functor. In: Proc. of the SAS. 2015. 36–53.

[27] Kincaid Z, Breck J, Reps T. Compositional recurrence analysis revisited. In: Proc. of the PLDI. 2017. 248–262.

[28] Kincaid Z, Cyphert J, Breck J, Reps T. Non-Linear reasoning for invariant synthesis. In: Proc. of the POPL. 2017. 1–33.

[29] Allamigeon X, Gaubert S, Goubault E, Putot S, Stott N. A fast method to compute disjunctive quadratic invariants of numerical programs. Trans. on Embedded Computing Systems, 2017,16(5):1–19.

[30] Oh H, Heo K, Lee W, Lee W, Park D, Kang J, Yi K. Global sparse analysis framework. Trans. on Programming Languages and Systems, 2014,36(3):1–44.

[31] Oh H, Lee W, Heo K, Yang H, Yi K. Selective *x*-sensitive analysis guided by impact pre-analysis. Trans. on Programming Languages and Systems, 2015,38(2):1–45.

[32] Singh G, Püschel M, Vechev M. Making numerical program analysis fast. In: Proc. of the PLDI. 2015. 303–313.

[33] Singh G, Püschel M, Vechev M. Fast polyhedra abstract domain. In: Proc. of the POPL. 2017. 46–59.

[34] Singh G, Püschel M, Vechev M. A practical construction for decomposing numerical abstract domains. In: Proc. of the POPL. 2018. 46–59.

[35] Singh G, Püschel M, Vechev M. Fast numerical program analysis with reinforcement learning. In: Proc. of the CAV. 2018. 211–229.

[36] Becchi A, Zaffanella E. A direct encoding for NNC polyhedra. In: Proc. of the CAV. 2018. 230–248.

[37] Liu J, Rival X. Abstraction of arrays based on non contiguous partitions. In: Proc. of the VMCAI. 2015. 282–299.

[38] Liu J, Rival X, Chen L. Automatic verification of embedded manipulating dynamic structures stored in system code contiguous regions. In: Proc. of the EMSOFT. 2018.

[39] Chen L, Miné A, Wang J, Cousot P. An abstract domain to discover interval linear equalities. In: Proc. of the VMCAI. 2010. 112–128.

[40] Fu Z. Modularly combining numeric abstract domains with points-to analysis, and a scalable static numeric analyzer for Java. In: Proc. of the VMCAI. 2014. 282–301.

[41] Illous H, Lemerre M, Rival X. A relational shape abstract domain. In: Proc. of the NFM. 2013. 212–229.

[42] Wu X, Chen L, Miné A, Dong W, Wang J. Static analysis of runtime errors in interrupt-driven programs via sequentialization. ACM Trans. on Embedded Computing Systems, 2016,15(4):1–26.

[43] Sung C, Kusano M, Wang C. Modular verification of interrupt-driven software. In: Proc. of the ASE. 2017. 206–216.

[44] Chakarov A, Sankaranarayanan S. Expectation invariants for probabilistic program loops as fixed points. In: Proc. of the SAS. 2014. 85–100.

[45] Wang D, Hoffmann J, Reps T. PMAF: An algebraic framework for static analysis of probabilistic programs. In: Proc. of the PLDI. 2018. 513–528.

[46] Ouadjaout A, Miné A, Lasla N, Badache N. Static analysis by abstract interpretation of functional properties of device drivers in TinyOS. Journal of Systems and Software, 2016,120:114–132.

[47] Cox A, Chang BE, Rival X. Desynchronized multi-state abstractions for open programs in dynamic languages. In: Proc. of the ESOP. 2015. 483–509.

[48] Lim J, Reps T. TSL: A system for generating abstract interpreters and its application to machine-code analysis. Trans. on Programming Languages and Systems, 2013,35(1):1–59.

[49] Tripp O, Pistoia M, Cousot P, Cousot R, Guarnieri S. Andromeda: Accurate and scalable security analysis of Web applications. In: Proc. of the FASE. 2013. 210–225.

[50] Urban C, Miné A. Proving guarantee and recurrence temporal properties by abstract interpretation. In: Proc. of the VMCAI. 2015. 190–208.

[51] Urban C. FuncTion: An abstract domain functor for termination. In: Proc. of the TACAS. 2015. 464–466.

[52] Gonnord L, Monniaux D, Radanne G. Synthesis of ranking functions using extremal counterexamples. In: Proc. of the PLDI. 2015. 608–618.

[53] Dan A, Meshman Y, Vechev M, Yahav E. Effective abstractions for verification under relaxed memory models. In: Proc. of the VMCAI. 2017. 62–76.

[54] Alglave J, Cousot P. Ogre and Pythia: An invariance proof method for weak consistency models. In: Proc. of the PLDI. 2017. 3–18.

[55] Gehr T, Mirman M, Drachsler-Cohen D, Tsankov P, Chaudhuri S, Vechev M. AI2: Safety and robustness certification of neural networks with abstract interpretation. In: Proc. of the IEEE S&P. 2018. 3–18.

[56] Wang S, Pei K, Whitehouse J, Yang J, Jana S. Formal security analysis of neural networks using symbolic intervals. Technical Report, 1804.10829, 2018.

[57] Urban C, Müller P. An abstract interpretation framework for input data usage. In: Proc. of the ESOP. 2018. 683–710.

[58] Fromherz A, Ouadjaout A, Miné A. Static value analysis of python programs by abstract interpretation. In: Proc. of the NFM. 2018. 185–202.

[59] Chae K, Oh H, Heo K, Yang H. Automatically generating features for learning program analysis heuristics for C-like languages. In: Proc. of the OOPSLA. 2017. 1–25.

[60] Seladji Y. Finding relevant templates via the principal component analysis. In: Proc. of the VMCAI. 2017. 483–499.

[61] Cousot P, Cousot R. Abstract interpretation: Past, present and future. In: Proc. of the CSL-LICS. 2014. 1–10.

[62] Aho AV, Sethi R, Ullman JD, *et al*. Compilers: Principles, Techniques, and Tools. Pearson Education Singapore, 1986.

[63] Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In: Proc. of the POPL. 1995. 49–61.

[64] Bodden E. Inter-Procedural data-flow analysis with IFDS/IDE and Soot. In: Proc. of the SOAP. 2012. 3–8.

[65] Dolby J. Watson libraries for analysis (WALA). 2018. https://github.com/wala/WALA

[66] Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Traon YL, Octeau D, McDaniel P. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Proc. of the PLDI. 2014. 259–269.

[67] Reps T. Program analysis via graph reachability. Information and Software Technology, 1998,40(11-12):701–726.

[68] Sridharan M, Bodik R. Refinement-Based context-sensitive points-to analysis for Java. In: Proc. of the PLDI. 2006. 387–400.

[69] Sui Y, Li Y, Xue J. Query-Directed adaptive heap cloning for optimizing compilers. In: Proc. of the CGO. 2013. 1–11.

[70] Pratikakis P, Foster JS, Hicks M. LOCKSMITH: Context-Sensitive correlation analysis for race detection. In: Proc. of the PLDI. 2006. 320–331.

[71] Zhou Q, Li L, Wang L, Xue J, Feng X. May-Happen-in-Parallel analysis with static vector clocks. In: Proc. of the CGO. 2018. 228–240.

[72] Li L, Cifuentes C, Keynes N. Boosting the performance of flow-sensitive points-to analysis using value flow. In: Proc. of the FSE. 2011. 343–353.

[73] Sui Y, Xue J. On-Demand strong update analysis via value-flow refinement. In: Proc. of the FSE. 2016. 460–473.

[74] Yu H, Xue J, Huo W, Feng X, Zhang Z. Level by level: Making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In: Proc. of the CGO. 2010. 218–229.

[75] Hardekopf B, Lin C. Flow-Sensitive pointer analysis for millions of lines of code. In: Proc. of the CGO. 2011. 289–298.

[76] Li L, Cifuentes C, Keynes N. Precise and scalable context-sensitive pointer analysis via value flow graph. In: Proc. of the ISMM. 2013. 85–96.

[77] Ali K, Lhoták O. Application-Only call graph construction. In: Proc. of the ECOOP. 2012. 688–712.

[78] Ali K, Lhoták O. Averroes: Whole-Program analysis without the whole program. In: Proc. of the ECOOP. 2013. 378–400.

[79] Dillig I, Dillig T, Aiken A. Reasoning about the unknown in static analysis. Communications of the ACM, 2010,53(8):115–123.

[80] Cousot P, Cousot R. Modular static program analysis. In: Proc. of the CC. 2002. 159–179.

[81] Smaragdakis Y, Balatsouras G, Kastrinis G. Set-Based pre-processing for points-to analysis. In: Proc. of the OOPSLA. 2013. 253–270.

[82] Rountev A, Ryder BG. Points-To and side-effect analyses for programs built with precompiled libraries. In: Proc. of the CC. 2001. 20–36.

[83] Rountev A, Kagan S, Marlowe T. Interprocedural dataflow analysis in the presence of large libraries. In: Proc. of the CC. 2006. 2–16.

[84] Rountev A, Sharp M, Xu G. IDE dataflow analysis in the presence of large object-oriented libraries. In: Proc. of the CC. 2008. 53–68.

[85] Sharir M, Pnueli A. Two approaches to interprocedural data flow analysis. Computer Science Department, Courant Institute of Mathematical Sciences, New York University, 1978. http://www.rw.cdl.uni-saarland.de/teaching/spa10/slides/kboesche.pdf

[86] Arzt S, Bodden E. StubDroid: Automatic inference of precise data-flow summaries for the Android framework. In: Proc. of the ICSE. 2016. 725–735.

[87] Tang H, Wang X, Zhang L, Xie B, Zhang L, Mei H. Summary-Based context-sensitive data-dependence analysis in presence of callbacks. In: Proc. of the POPL. 2015. 83–95.

[88] Tang H, Wang D, Xiong Y, Zhang L, Wang X, Zhang L. Conditional DYCK-CFL reachability analysis for complete and efficient library summarization. In: Proc. of the ESOP. 2017. 880–908.

[89] Palepu VK, Xu G, Jones JA. Improving efficiency of dynamic analysis with dynamic dependence summaries. In: Proc. of the ASE. 2013. 59–69.

[90] Kulkarni S, Mangal R, Zhang X, Naik M. Accelerating program analyses by cross-program training. In: Proc. of the OOPSLA. 2016. 359–377.

[91] Yorsh G, Yahav E, Chandra S. Generating precise and concise procedure summaries. In: Proc. of the POPL. 2008. 221–234.

[92] Strom RE, Yemini S. Typestate: A programming language concept for enchancing software reliability. IEEE Trans. on Software Engineering, 1986,12(1):157–171.

[93] Dillig I, Dillig T, Aiken A, Sagiv M. Precise and compact modular procedure summaries for heap manipulating programs. In: Proc. of the PLDI. 2011. 567–577.

[94] Boyer RS, Elspas B, Levitt KN. SELECT—A formal system for testing and debugging programs by symbolic execution. ACM SIGPLAN Notices, 1975,10(6):234–245.

[95] Clarke LA. A program testing system. In: Proc. of the Annual Conf. 1976. 488–491.

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

[97] Zhang J. Sharp static analysis of programs. Chinese Journal of Computers, 2008,31(9):1549–1553 (in Chinese with English abstract).

[98] Zhang J. Constraint solving and symbolic execution. In: Proc. of the VSTTE. 2008. 539–544.

[99] De Moura L, Bjørner N. Z3: An efficient SMT solver. In: Proc. of the TACAS. 2008. 337–340.

[100] Ganesh V, Dill DL. A decision procedure for bit-vectors and arrays. In: Proc. of the CAV. 2007. 519–531.

[101] Dutertre B, de Moura L. A fast linear-arithmetic solver for DPLL(T). In: Proc. of the CAV. 2016. 81–94.

[102] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing. In: Proc. of the PLDI. 2005. 213–223.

[103] Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C. In: Proc. of the FSE. 2005. 263–272.

[104] Godefroid P, Levin MY, Molnar D. Automated whitebox fuzz testing. In: Proc. of the NDSS. 2008. 151–166.

[105] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the OSDI. 2008. 209–224.

[106] Păsăreanu CS, Mehlitz PC, Bushnell DH, Gundy-Burlet K, Lowry M, Person S, Pape M. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: Proc. of the ISSTA. 2008. 15–26.

[107] Tillmann N, de Halleux J. Pex-White box test generation for .NET. In: Proc. of the TAP. 2008. 134–153.

[108] Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. In: Proc. of the ASPLOS. 2011. 265–278.

[109] Xie T, Tillmann N, de Halleux J, Schulte W. Fitness-Guided path exploration in dynamic symbolic execution. In: Proc. of the DSN. 2009. 359–368.

[110] Li Y, Su Z, Wang L, Li X. Steering symbolic execution to less traveled paths. In: Proc. of the OOPSLA. 2013. 19–32.

[111] Seo H, Kim S. How we get there: A context-guided search strategy in concolic testing. In: Proc. of the FSE. 2014. 413–424.

[112] Ma KK, Khoo YP, Foster JS, Hicks M. Directed symbolic execution. In: Proc. of the SAS. 2011. 95–111.

[113] Zhang Y, Chen Z, Wang J, Dong W, Liu Z. Regular property guided dynamic symbolic execution. In: Proc. of the ICSE. 2015. 643–653.

[114] Person S, Yang G, Rungta N, Khurshid S. Directed incremental symbolic execution. In: Proc. of the PLDI. 2011. 504–515.

[115] Dan Marinescu P, Cadar C. Make test-zesti: A symbolic execution solution for improving regression testing. In: Proc. of the ICSE. 2012. 716–726.

[116] Godefroid P, Kiezun A, Levin MY. Grammar-Based whitebox fuzzing. In: Proc. of the PLDI. 2008. 206–215.

[117] Siddiqui JH, Khurshid S. Scaling symbolic execution using ranged analysis. In: Proc. of the OOPSLA. 2012. 523–536.

[118] Slabý J, Strejček J, Trtík M. Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution. In: Proc. of the FMICS. 2012. 207–221.

[119] Cui H, Hu G, Wu J, Yang J. Verifying systems rules using rule-directed symbolic execution. In: Proc. of the ASPLOS. 2013. 329–342.

[120] Trabish D, Mattavelli A, Rinetzky N, Cadar C. Chopped symbolic execution. In: Proc. of the ICSE. 2018. 350–360.

[121] Yu H, Chen Z, Wang J, Su Z, Dong W. Symbolic verification of regular properties. In: Proc. of the ICSE. 2018. 871–881.

[122] Boonstoppel P, Cadar C, Engler D. RWset: Attacking path explosion in constraint-based test generation. In: Proc. of the TACAS. 2008. 351–366.

[123] Jaffar J, Murali V, Navas JA. Boosting concolic testing via interpolation. In: Proc. of the FSE. 2013. 48–58.

[124] Godefroid P. Compositional dynamic test generation. In: Proc. of the POPL. 2007. 47–54.

[125] Saxena P, Poosankam P, McCamant S, Song D. Loop-Extended symbolic execution on binary programs. In: Proc. of the ISSTA. 2009. 225–236.

[126] Godefroid P, Luchaup D. Automatic partial loop summarization in dynamic test generation. In: Proc. of the ISSTA. 2011. 23–33.

[127] Strejček J, Trtík M. Abstracting path conditions. In: Proc. of the ISSTA. 2012. 155–165.

[128] Qiu R, Yang G, Pâsâreanu CS, Khurshid S. Compositional symbolic execution with memoized replay. In: Proc. of the ICSE. 2015. 632–642.

[129] Yi Q, Yang Z, Guo S, Wang C, Liu J, Zhao C. Eliminating path redundancy via postconditioned symbolic execution. IEEE Trans. on Software Engineering, 2018,44(1):25–43.

[130] Sen K. Scalable automated methods for dynamic program analysis [Ph.D. Thesis]. University of Illinois at Urbana-Champaign, 2006.

[131] Wang C, Yang Z, Kahlon V, Gupta A. Peephole partial order reduction. In: Proc. of the TACAS. 2008. 382–396.

[132] Kuznetsov V, Kinder J, Bucur S, Candea G. Efficient state merging in symbolic execution. ACM SIGPLAN Notices, 2012,47(6): 193–204.

[133] Avgerinos T, Rebert A, Sang KC, Brumley D. Enhancing symbolic execution with veritesting. In: Proc. of the ICSE. 2014. 1083–1094.

[134] Qi D, Nguyen HDT, Roychoudhury A. Path exploration based on symbolic output. ACM Trans. on Software Engineering and Methodology, 2013,22(4):32.

[135] Guo S, Kusano M, Wang C, Yang Z, Gupta A. Assertion guided symbolic execution of multithreaded programs. In: Proc. of the FSE. 2015. 854–865.

[136] Wang H, Liu T, Guan X, Shen C, Zheng Q, Yang Z. Dependence guided symbolic execution. IEEE Trans. on Software Engineering, 2017,43(3):252–271.

[137] Visser W, Geldenhuys J, Dwyer MB. Green: Reducing, reusing and recycling constraints in program analysis. In: Proc. of the FSE. 2012. 58.

[138] Aquino A, Bianchi FA, Chen M, Denaro G, Pezzè M. Reusing constraint proofs in program analysis. In: Proc. of the ISSTA. 2015. 305–315.

[139] Jia X, Ghezzi C, Ying S. Enhancing reuse of constraint solutions to improve symbolic execution. In: Proc. of the ISSTA. 2015. 177–187.

[140] Zhang Y, Chen Z, Wang J. Speculative symbolic execution. In: Proc. of the ISSRE. 2012. 101–110.

[141] Barr ET, Vo T, Le V, Su Z. Automatic detection of floating-point exceptions. In: Proc. of the POPL. 2013. 549–560.

[142] Lakhotia K, Tillmann N, Harman M, de Halleux J. FloPSy-Search-Based floating point constraint solving for symbolic execution. In: Proc. of the ICTSS. 2010. 142–157.

[143] Romano A. Practical floating-point tests with integer code. In: Proc. of the VMCAI. 2014. 337–356.

[144] 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 ASE. 2016. 554–559.

[145] Fu Z, Su Z. XSat: A fast floating-point satisfiability solver. In: Proc. of the CAV. 2016. 187–209.

[146] Bjørner N, Tillmann N, Voronkov A. Path feasibility analysis for string-manipulating programs. In: Proc. of the TACAS. 2009. 307–321.

[147] Khurshid S, Pasareanu CS, Visser W. Generalized symbolic execution for model checking and testing. In: Proc. of the TACAS. 2003. 553–568.

[148] Ramos DA, Engler D. Under-Constrained symbolic execution: Correctness checking for real code. In: Proc. of the USENIX Security. 2015. 49–64.

[149] Rosner N, Geldenhuys J, Aguirre NM, Visser W, Frias MF. BLISS: Improved symbolic execution by bounded lazy initialization with SAT support. IEEE Trans. on Software Engineering, 2015,41(7):639–660.

[150] Jeon J, Qiu X, Fetter-Degges J, Foster JS, Solar-Lezama A. Synthesizing framework models for symbolic execution. In: Proc. of the ICSE. 2016. 156–167.

[151] Cha SK, Avgerinos T, Rebert A, Brumley D. Unleashing mayhem on binary code. In: Proc. of the IEEE S&P. 2012. 380–394.

[152] 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 IEEE S&P. 2016. 138–157.

[153] Banabic R, Candea G, Guerraoui R. Finding trojan message vulnerabilities in distributed systems. In: Proc. of the ASPLOS. 2014. 113–126.

[154] Bucur S, Kinder J, Candea G. Prototyping symbolic execution engines for interpreted languages. In: Proc. of the ASPLOS. 2014. 239–253.

[155] Emmi M, Majumdar R, Sen K. Dynamic test input generation for database applications. In: Proc. of the ISSTA. 2007. 151–162.

[156] Sasnauskas R, Landsiedel O, Alizai MH, Weise C, Kowalewski S, Wehrle K. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In: Proc. of the IPSN. 2010. 186–196.

[157] Fu X, Chen Z, Yu H, Huang C, Dong W, Wang J. Poster: Symbolic execution of MPI programs. In: Proc. of the ICSE. 2015. 809–810.

[158] Yu H. Combining symbolic execution and model checking to verify MPI programs. In: Proc. of the ICSE. 2018. 527–529.

[159] Davidson D, Moench B, Jha S, Ristenpart T. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In: Proc. of the USENIX Security. 2013. 463–478.

[160] Guo S, Wu M, Wang C. Symbolic execution of programmable logic controller code. In: Proc. of the FSE. 2017. 326–336.

[161] Liu S, Zhang J. Program analysis: From qualitative analysis to quantitative analysis. In: Proc. of the ICSE. 2011. 956–959.

[162] Geldenhuys J, Dwyer MB, Visser W. Probabilistic symbolic execution. In: Proc. of the ISSTA. 2012. 166–176.

[163] Filieri A, Pasareanu CS, Visser W. Reliability analysis in symbolic pathFinder. In: Proc. of the ICSE. 2013. 622–631.

[164] Chen B, Liu Y, Le W. Generating performance distributions via probabilistic symbolic execution. In: Proc. of the ICSE. 2016. 49–60.

[165] Filieri A, Pasareanu CS, Yang G. Quantification of software changes through probabilistic symbolic execution. In: Proc. of the ASE. 2015. 703–708.

[166] Godefroid P, Levin MY, Molnar D. SAGE: Whitebox fuzzing for security testing. Communications of the ACM, 2012,10(3):40–44.

[167] Miscrosfot. Visual studio 2015 RTM. 2018. https://www.visualstudio.com/news/vs2015-vs#Testing

[168] Xu Z, Zhang J, Wang J. Canalyze: A static bug-finding tool for C programs. In: Proc. of the ISSTA. 2014. 425–428.

[169] Cha S, Hong S, Lee J, Oh H. Automatically generating search heuristics for concolic testing. In: Proc. of the ICSE. 2018. 1244–1254.

[170] Su T, Fu Z, Pu G, He J, Su Z. Combining symbolic execution and model checking for data flow testing. In: Proc. of the ICSE. 2015. 654–665.

[171] Christakis M, Müller P, Wüstholz V. Guiding dynamic symbolic execution toward unverified program executions. In: Proc. of the ICSE. 2016. 144–155.

[172] Wang X, Sun J, Chen Z, Zhang P, Wang J, Lin Y. Towards optimal concolic testing. In: Proc. of the ICSE. 2018. 291–302.

[173] Stephens N, Grosen J, Salls C, Dutcher A, Wang R, Corbetta J, Shoshitaishvili Y, Kruegel C, Vigna G. Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. of the NDSS. 2016. 21–24.

[174] Gosain A, Sharma G. A survey of dynamic program analysis techniques and tools. In: Proc. of the FICTA. 2014. 113–122.

[175] Kong P, Li Y, Chen X, Sun J, Sun M, Wang J. Towards concolic testing for hybrid systems. In: Proc. of the FM. 2016. 460–478.

[176] Chen Y, Poskitt CM, Sun J. Learning from mutants: Using code mutation to learn and monitor invariants of a cyber-physical system. In: Proc. of the SP. 2018. 648–660.

[177] Xiong Y, Wang B, Fu G, Zang L. Learning to synthesize. In: Proc. of the GI. 2018. 37–44.

[178] Heo K, Oh H, Yi K. Machine-Learning-Guided selectively unsound static analysis. In: Proc. of the ICSE. 2017. 519–529.

[179] Oh H, Yang H, Yi K. Learning a strategy for adapting a program analysis via Bayesian optimisation. In: Proc. of the OOPSLA. 2015. 572–588.

[180] Heo K, Oh H, Yang H, Yi K. Adapting static analysis via learning with bayesian optimization. ACM Trans. on Programming Languages and Systems, 2018.

[181] Jeong S, Jeon M, Cha S, Oh H. Data-Driven context-sensitivity for points-to analysis. In: Proc. of the OOPSLA. 2017. 1–28.

[182] Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, Sheth AN. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proc. of the USENIX Security. 2014. 393–407.

[183] Hornyack P, Han S, Jung J, Schechter S, Wetherall D. These aren't the Droids you're looking for: Retrofitting Android to protect data from imperious applications. In: Proc. of the CCS. 2011. 639–652.

[184] DroidBox: Android application sandbox. 2018. http://code.google.com/p/droidbox/

[185] Yan LK, Yin H. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In: Proc. of the USENIX Security. 2014. 569–584.

[186] Bellard F. QEMU, a fast and portable dynamic translator. In: Proc. of the ATEC. 2005. 41–46.

[187] TEMU: The BitBlaze dynamic analysis component. 2018. http://bitblaze.cs.berkeley.edu/temu.html

[188] Gibler C, Crussell J, Erickson J, Chen H. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In: Proc. of the TRUST. 2012. 291–307.

[189] Feng Y, Anand S, Dillig I, Aiken A. Apposcopy: Semantics-Based detection of Android malware through static analysis. In: Proc. of the FSE. 2014. 576–587.

[190] Gordon MI, Kim D, Perkins J, Gilham L, Nguyen N, Rinard M. Information-Flow analysis of Android applications in DroidSafe. In: Proc. of the NDSS. 2015. 8–11.

[191] Jin X, Hu X, Ying K, Du W, Yin H, Peri GN. Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation. In: Proc. of the CCS. 2014. 66–77.

[192] Egele M, Kruegel C, Kirda E, Vigna G. PiOS: Detecting privacy leaks in iOS applications. In: Proc. of the NDSS. 2011. 11.

[193] Octeau D, McDaniel P, Jha S, Bartel A, Bodden E, Klein J, Traon YL. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In: Proc. of the USENIX Security. 2013. 543–558.

[194] Wu T, Liu J, Xu Z, Guo C, Zhang Y, Yan J, Zhang J. Light-Weight, inter-procedural and callback-aware resource leak detection for Android apps. IEEE Trans. on Software Engineering, 2016,42(11):1054–1076.

[195] Chan PPF, Hui LCK, Yiu SM. DroidChecker: Analyzing Android applications for capability leak categories. In: Proc. of the WISEC. 2012. 125–136.

[196] Felt AP, Wang HJ, Moshchuk A, Hanna S, Chin E. Permission re-delegation: Attacks and defenses. In: Proc. of the USENIX Security. 2011. 22–22.

[197] Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM. Using GUI ripping for automated testing of Android applications. In: Proc. of the ASE. 2012. 258–261.

[198] Machiry A, Tahiliani R, Naik M. Dynodroid: An input generation system for Android apps. In: Proc. of the FSE. 2013. 224–234.

[199] Hu G, Yuan X, Tang Y, Yang J. Efficiently, effectively detecting mobile app bugs with AppDoctor. In: Proc. of the EuroSys. 2014. 1–15.

[200] Mahmood R, Mirzaei N, Malek S. EvoDroid: Segmented evolutionary testing of Android apps. In: Proc. of the FSE. 2014. 599–609.

[201] Mirzaei N, Garcia J, Bagheri H, Sadeghi A, Malek S. Reducing combinatorics in GUI testing of Android applications. In: Proc. of the ICSE. 2016. 559–570.

[202] You W, Liang B, Shi W, Zhu S, Wang P, Xie S, Zhang X. Reference hijacking: Patching, protecting and analyzing on unmodified and non-rooted Android devices. In: Proc. of the ICSE. 2016. 959–970.

[203] Au KWY, Zhou YF, Huang Z, Lie D. PScout: Analyzing the Android permission specification. In: Proc. of the CCS. 2012. 217–228.

[204] Cao Y, Fratantonio Y, Bianchi A, Egele M, Kruegel C, Vigna G, Chen Y. EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework. In: Proc. of the NDSS. 2015. 8–11.

[205] Rasthofer S, Arzt S, Bodden E. A machine-learning approach for classifying and categorizing Android sources and sinks. In: Proc. of the NDSS. 2014. 23–26.

[206] Zhang Y, Luo X, Yin H. DexHunter: Toward extracting hidden code from packed Android applications. In: Proc. of the ESORICS. 2015. 293–311.

[207] Xue L, Luo X, Yu L, Wang S, Wu D. Adaptive unpacking of Android apps. In: Proc. of the ICSE. 2017. 358–369.

[208] Clarke EM, Emerson EA, Sistla AP. Automatic verification of finite-state concurrent systems using temporal logic specifications. Trans. on Programming Languages and Systems, 1986,8(2):244–263.

[209] Bron A, Farchi E, Magid Y, Nir Y, Ur S. Applications of synchronization coverage. In: Proc. of the PPoPP. 2005. 206–212.

[210] Křena B, Letko Z, Vojnar T. Coverage metrics for saturation-based and search-based testing of concurrent software. In: Proc. of the RV. 2011. 177–192.

[211] Sorrentino F, Farzan A, Madhusudan P. PENELOPE: Weaving threads to expose atomicity violations. In: Proc. of the FSE. 2010. 37–46.

[212] Wang C, Said M, Gupta A. Coverage guided systematic concurrency testing. In: Proc. of the ICSE. 2011. 221–230.

[213] Musuvathi M, Qadeer S, Ball T, Basler G, Nainar PA, Neamtiu I. Finding and reproducing heisenbugs in concurrent programs. In: Proc. of the OSDI. 2008. 267–280.

[214] Kahlon V, Sinha N, Kruus E, Zhang Y. Static data race detection for concurrent programs with asynchronous calls. In: Proc. of the FSE. 2009. 13–22.

[215] Naik M, Aiken A, Whaley J. Effective static race detection for Java. In: Proc. of the PLDI. 2006. 308–319.

[216] Voung JW, Ranjit J, Lerner S. RELAY: Static race detection on millions of lines of code. In: Proc. of the FSE. 2007. 205–214.

[217] Cai Y, Wu S, Chan WK. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In: Proc. of the ICSE. 2014. 491–502.

[218] Pozniansky E, Schuster A. Efficient on-the-fly data race detection in multihreaded C++ programs. In: Proc. of the PPoPP. 2003. 179–190.

[219] Savage S, Burrows M, Nelson G, Sobalvarro P, Anderson T. Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. on Computer Systems, 1997,15(4):391–411.

[220] Yu Y, Rodeheffer T, Chen W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In: Proc. of the SOSP. 2005. 221–234.

[221] Xie X, Xue J. Acculock: Accurate and efficient detection of data races. In: Proc. of the CGO. 2011. 201–212.

[222] Kasikci B, Zamfir C, Candea G. RaceMob: Crowdsourced data race detection. In: Proc. of the SOSP. 2013. 406–422.

[223] Williams A, Thies W, Ernst M. Static deadlock detection for Java libraries. In: Proc. of the ECOOP. 2005. 602–629.

[224] Clarke EM, Emerson EA. Design and synthesis of synchronization skeletons using branching time temporal logic. In: Proc. of the Logic of Programs. 1981. 52–71.

[225] Bach M, Charney M, Cohn R, Demikhovsky E, Devor T, Hazelwood K, Jaleel A, Luk C, Lyons G, Patil H, Tal A. Analyzing parallel programs with pin. IEEE Computer, 2010,43(3):34–41.

[226] Eslamimehr M, Palsberg J. Race directed scheduling of concurrent programs. In: Proc. of the PPoPP. 2014. 301–314.

[227] Edelstein O, Farchi E, Nir Y, Ratsaby G, Ur S. Multithreaded Java program test generation. IBM Systems Journal, 2002,41(1): 111–125.

[228] Burckhardt S, Kothari P, Musuvathi M, Nagarakatte S. A randomized scheduler with probabilistic guarantees of finding bugs. In: Proc. of the ASPLOS. 2010. 167–178.

[229] Cai Y, Yang Z. Radius aware probabilistic testing of deadlocks with guarantees. In: Proc. of the ASE. 2016. 356–367.

[230] Fonseca P, Rodrigues R, B.Brandenburg B. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In: Proc. of the OSDI. 2014. 415–431.

[231] Yu J, Narayanasamy S, Pereira C, Pokam G. Maple: A coverage-driven testing tool for multithreaded programs. In: Proc. of the OOPSLA. 2012.

[232] Abdelrasoul M. Promoting secondary orders of event pairs in randomized scheduling using a randomized stride. In: Proc. of the ASE. 2017. 741–752.

[233] Lu S, Park S, Seo E, Zhou Y. Learning from mistakes—A comprehensive study on real world concurrency bug characteristics. In: Proc. of the ASPLOS. 2008. 329–339.

[234] Flanagan C, Freund SN. FastTrack: Efficient and precise dynamic race detection. Communications of the ACM, 2010,53(11): 93–101.

[235] Cai Y, Cao L. Effective and precise dynamic detection of hidden races for Java programs. In: Proc. of the FSE. 2015. 450–461.

[236] Smaragdakis Y, Evans JM, Sadowski C, Yi J, Flanagan C. Sound predictive race detection in polynomial time. In: Proc. of the POPL. 2012. 387–400.

[237] Marino D, Musuvathi M, Narayanasamy S. LiteRace: Effective sampling for lightweight data-race detection. In: Proc. of the PLDI. 2009. 134–143.

[238] Bond MD, Coons KE, Mckinley KS. PACER: Proportional detection of data races. In: Proc. of the PLDI. 2010. 255–268.

[239] Zhai K, Xu B, Chan WK, Tse TH. CARISMA: A context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In: Proc. of the ISSTA. 2012. 221–231.

[240] Erickson J, Musuvathi M, Burckhardt S, Olynyk K. Effective data-race detection for the kernel. In: Proc. of the OSDI. 2010. 1–16.

[241] Sen K. Race directed random testing of concurrent programs. In: Proc. of the PLDI. 2008. 11–21.

[242] Joshi P, Park CS, Sen K, Naik M. A randomized dynamic program analysis technique for detecting real deadlocks. In: Proc. of the PLDI. 2009. 110–120.

[243] Huang J, Meredith PO, Rosu G. Maximal sound predictive race detection with control flow abstraction. In: Proc. of the PLDI. 2013. 337–348.

[244] Cai Y, Zhang J, Cao L, Liu J. A deployable sampling strategy for data race detection. In: Proc. of the PLDI. 2016. 810–821.

[245] Guo Y, Cai Y, Yang Z. AtexRace: Across thread and execution sampling for in-house race detection. In: Proc. of the FSE. 2017. 315–325.

[246] Lu S, Tucek J, Qin F, Zhou Y. AVIO: Detecting atomicity violations via access-interleaving invariants. In: Proc. of the ASPLOS. 2006. 37–48.

[247] Lu S, Park S, Hu C, Ma X, Jiang W, Li Z, Popa RA, Zhou Y. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In: Proc. of the SOSP. 2007. 103–116.

[248] Park CS, Sen K. Randomized active atomicity violation detection in concurrent programs. In: Proc. of the FSE. 2008. 135–145.

[249] Muzahid A, Otsuki N, Torrellas J. AtomTracker: A comprehensive approach to atomic region inference and violation detection. In: Proc. of the MICRO. 2010. 287–297.

[250] Cai Y, Chan WK. MagicFuzzer: Scalable deadlock detection for large-scale applications. In: Proc. of the ICSE. 2012. 606–616.

[251] Cai Y, Chan WK. Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs. IEEE Trans. on Software Engineering, 2014,40(3):266–281.

[252] Cai Y, Lu Q. Dynamic testing for deadlocks via constraints. IEEE Trans. on Software Engineering, 2016,42(9):825–842.

[253] Cai Y, Jia C, Wu S, Zhai K, Chan WK. ASN: A dynamic barrier-based approach to confirmation of deadlocks from warnings for large-scale multithreaded programs. IEEE Trans. on Parallel and Distributed Systems, 2015,26(1):13–23.

[254] Pradel M, Gross TR. Fully automatic and precise detection of thread safety violations. In: Proc. of the PLDI. 2012. 521–530.

[255] Choudhary A, Lu S, Pradel M. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In: Proc. of the ICSE. 2017. 266–277.

[256] The 10 biggest cloud outages of 2014. 2018. http://www.crn.com/slide-shows/cloud/300075204/the-10-biggest-cloud-outages-of-2014.html

[257] Gunawi HS, Hao M, Leesatapornwongsa T, Patana-anake T, Do T, Adityatama J, Eliazar KJ, Laksono A, Lukman JF, Martin V, Satria AD. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In: Proc. of the SOCC. 2014. 1–14.

[258] Leesatapornwongsa T, Lukman JF, Lu S, Gunawi HS. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In: Proc. of the ASPLOS. 2016. 517–530.

[259] Dai T, He J, Gu X, Lu S. Understanding real-world timeout problems in cloud server systems. In: Proc. of the IC2E. 2018. 1–11.

[260] Gao Y, Dou W, Qin F, Gao C, Wang D, Wei J, Huang R, Zhou L, Wu Y. An empirical study on crash recovery bugs in large-scale distributed systems. In: Proc. of the FSE. 2018. 539–550.

[261] Guo Z, McDirmid S, Yang M, Zhuang L, Zhang P, Luo Y, Bergan T, Bodik P, Musuvathi M, Zhang Z, Zhou L. Failure recovery: When the cure is worse than the disease. In: Proc. of the HotOS. 2013. 1–6.

[262] Yuan D, Luo Y, Zhuang X, Rodrigues GR, Zhao X, Zhang Y, Jain PU, Stumm M. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In: Proc. of the OSDI. 2014. 249–265.

[263] Wang G, Zhang L, Xu W. What can we learn from four years of data center hardware failures? In: Proc. of the DSN. 2017. 25–36.

[264] Liu H, Li G, Lukman JF, Li J, Lu S, Gunawi HS, Tian C. DCatch: Automatically detecting distributed concurrency bugs in cloud systems cloud systems. In: Proc. of the ASPLOS. 2017. 677–691.

[265] Liu H, Wang X, Li G, Lu S, Ye F, Tian C. FCatch: Automatically detecting time-of-fault bugs in cloud systems. In: Proc. of the ASPLOS. 2018. 1–11.

[266] Liu X, Guo Z, Wang X, Chen F, Lian X, Tang J, Wu M, Kaashoek MF, Zhang Z. D3S: Debugging deployed distributed systems. In: Proc. of the NSDI. 2008. 423–437.

[267] Xu W, Huang L, Fox A, Patterson D, Jordan MI. Detecting large-scale system problems by mining console logs. In: Proc. of the SOSP. 2009. 117–132.

[268] Grant S, Cech H, Beschastnikh I. Inferring and asserting distributed system invariants. In: Proc. of the ICSE. 2018. 1149–1159.

[269] Scott C, Panda A, Brajkovic V, Necula G, Krishnamurthy A, Shenker S. Minimizing faulty executions of distributed systems. In: Proc. of the NSDI. 2016. 291–309.

[270] Flanagan C, Godefroid P. Dynamic partial-order reduction for model checking software. In: Proc. of the POPL. 2005. 110–121.

[271] Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. IEEE Trans. on Software Engineering, 2002,28(2): 183–200.

[272] Wilcox JR, Woos D, Panchekha P, Tatlock Z, Wang X, Ernst MD, Anderson T. Verdi: A framework for implementing and formally verifying distributed systems. In: Proc. of the PLDI. 2015. 357–368.

[273] Hawblitzel C, Howell J, Kapritsos M, Lorch JR, Parno B, Roberts ML, Setty S, Zill B. IronFleet: Proving practical distributed systems correct. In: Proc. of the SOSP. 2015. 1–17.

[274] Lesani M, Bell CJ, Chlipala A. Chapar: Certified causally consistent distributed key-value stores. In: Proc. of the POPL. 2016. 357–370.

[275] Fonseca P, Zhang K, Wang X, Krishnamurthy A. An empirical study on the correctness of formally verified distributed systems. In: Proc. of the EuroSys. 2017. 328–343.

[276] Yang J, Chen T, Wu M, Xu Z, Liu X, Lin H, Yang M, Long F, Zhang L, Zhou L. MODIST: Transparent model checking of unmodified distributed systems. In: Proc. of the NSDI. 2009. 213–228.

[277] Guo H, Wu M, Zhou L, Hu G, Yang J, Zhang L. Practical software model checking via dynamic interface reduction. In: Proc. of the SOSP. 2011. 265–278.

[278] Leesatapornwongsa T, Hao M, Joshi P, Lukman JF, Gunawi HS. SAMC: Semantic-Aware model checking for fast discovery of deep bugs in cloud systems. In: Proc. of the OSDI. 2014. 399–414.

[279] Sasnauskas R, Dustmann OS, Kaminski BL, Wehrle K, Weise C, Kowalewski S. Scalable symbolic execution of distributed systems. In: Proc. of the ICDCS. 2011. 333–342.

[280] Joahi P, Ganai M, Balakrishnan G, Gupta A, Papakonstantinou N. SETSUDO: Perturbation-Based testing framework for scalable distributed systems pallavi. In: Proc. of the TRIOS. 2013. 1–14.

[281] Joshi P, Gunawi HS, Sen K. PREFAIL: A programmable tool for multiple-failure injection. In: Proc. of the OOPSLA. 2011. 171–188.

[282] Gunawi HS, Do T, Joshi P, Alvaro P, Hellerstein JM, Arpaci-Dusseau AC, Arpaci-Dusseau RH, Sen K, Borthakur D. FATE and DESTINI: A framework for cloud recovery testing. In: Proc. of the NSDI. 2011. 1–18.

[283] Alagappan R, Ganesan A, Patel Y, Pillai TS, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Correlated crash vulnerabilities. In: Proc. of the OSDI. 2016. 151–167.

[284] Ganesan A, Alagappan R, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to file-system faults. In: Proc. of the FAST. 2017. 149–166.

[285] Alvaro P, Rosen J, Hellerstein JM. Lineage-Driven fault injection. In: Proc. of the SIGMOD. 2015. 331–346.

[286] Thompson K. Reflections on trusting trust. Communications of the ACM, 1984,27(8):761–763.

[287] Xiao C. Novel malware xcodeghost modifies xcode, infects apple iOS apps and hits app store. Technical Report, 2018. https://researchcenter.paloaltonetworks.com/2015/09/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/

[288] De Sutter B, de Bus B, de Bosschere K, Keyngnaert P, Demoen B. On the static analysis of indirect control transfers in binaries. In: Proc. of the PDPTA. 2000.

[289] Kinder J. Static analysis of x86 executables (statische analyse von programmen in x86-Maschinensprache) [Ph.D. Thesis]. Fachbereich Informatik, Technische Universitat Darmstadt, 2010.

[290] Cifuentes C, van Emmerik M. Recovery of jump table case statements from binary code. Science of Computer Programming, 2001, 40(2–3):171–188.

[291] Kinder J, Veith H. Jakstab: A static analysis platform for binaries. In: Proc. of the CAV. 2008. 423–427.

[292] Xu L, Sun F, Su Z. Constructing precise control flow graphs from binaries. Technical Report, 2009.

[293] Shin ECR, Song D, Moazzezi R. Recognizing functions in binaries with neural networks. In: Proc. of the USENIX Security. 2015. 611–626.

[294] Andriesse D, Chen X, Van der Veen V, Slowinska A, Bos H. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: Proc. of the USENIX Security. 2016. 583–600.

[295] Chua ZL, Shen S, Saxena P, Liang Z. Neural nets can learn function type signatures from binaries. In: Proc. of the USENIX Security. 2017. 99–116.

[296] Tröger J, Cifuentes C. Analysis of virtual method invocation for binary translation. In: Proc. of the WCRE. 2002. 65–74.

[297] Balakrishnan G, Reps T. Analyzing memory accesses in x86 executables. In: Proc. of the CC. 2004. 5–23.

[298] Jin W, Cohen C, Gennari J, Hines C, Chaki S, Gurfinkel A, Havrilla J, Narasimhan P. Recovering C++ objects from binaries using inter-procedural data-flow analysis. In: Proc. of the POPL. 2014. 1–11.

[299] Lin Z, Zhang X, Xu D. Automatic reverse engineering of data structures from binary execution. In: Proc. of the NDSS. 2010.

[300] Zeng J, Lin Z. Towards automatic inference of kernel object semantics from binary code. In: Proc. of the RAID. 2015. 538–561.

[301] Smithson M, El Wazeer K, Anand K, Kotha A, Barua R. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In: Proc. of the WCRE. 2013. 52–61.

[302] Wang S, Wang P, Wu D. UROBOROS: Instrumenting stripped binaries with static reassembling. In: Proc. of the SANER. 2016. 236–247.

[303] Wang S, Wang P, Wu D. Reassembleable disassembling. In: Proc. of the USENIX Security. 2015. 627–642.

[304] Wang R, Shoshitaishvili Y, Bianchi A, Machiry A, Grosen J, Grosen P, Kruegel C, Vigna G. RAMBLR: Making reassembly great again. In: Proc. of the NDSS. 2017.

[305] Song D, Brumley D, Yin H, Caballero J, Jager I, Kang MG, Liang Z, Newsome J, Poosankam P, Saxena P. BitBlaze: A new approach to computer security via binary analysis. In: Proc. of the ICISS. 2008. 1–25.

[306] Brumley D, Jager I, Avgerinos T, Schwartz EJ. BAP: A binary analysis platform. In: Proc. of the CAV. 2011. 463–469.

[307] Bruening D. Efficient, transparent, and comprehensive runtime code manipulation [Ph.D. Thesis]. Massachusetts Institute of Technology, 2004.

[308] Bernat AR, Miller BP. Anywhere, any-time binary instrumentation. In: Proc. of the PASTE. 2011. 9–16.

[309] Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proc. of the PLDI. 2007. 89–100.

[310] Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. In: Proc. of the PLDI. 2005. 190–200.

[311] Machiry A, Spensky C, Corina J, Stephens N, Kruegel C, Vigna G, Barbara S. DR.CHECKER: A soundy analysis for Linux kernel drivers. In: Proc. of the USENIX Security. 2017. 1007–1024.

[312] Wang P, Krinke J, Lu K, Li G, Dodier-Lazaro S. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the Linux kernel. In: Proc. of the USENIX Security. 2017. 1–16.

[313] Dewey D, Giffin J. Static detection of C++ vtable escape vulnerabilities in binary code. In: Proc. of the NDSS. 2012.

[314] Kruegel C, Robertson W, Vigna G. Detecting kernel-level rootkits through binary analysis. In: Proc. of the ACSAC. 2004. 91–100.

[315] Bergeron J, Debbabi M, Erhioui MM, Ktari B. Static analysis of binary code to isolate malicious behaviors. In: Proc. of the WETICE. 1999. 184–189.

[316] Bergeron J, Debbabi M, Desharnais J, M.Erhioui M, Lavoie Y, Tawbi N. Static detection of malicious code in executable programs. In: Proc. of the Symp. on Requirements Engineering for Information Security. 2001.

[317] O'Sullivan P, Anand K, Kotha A, Smithson M, Barua R, Keromytis AD. Retrofitting security in COTS software with binary rewriting. In: Proc. of the SEC. 2011. 154–172.

[318] Wartell R, Mohan V, Hamlen KW, Lin Z. Binary stirring: Self-Randomizing instruction addresses of legacy x86 binary code. In: Proc. of the Computer and Communications Security. New York: ACM Press, 2012. 157–168.

[319] Wartell R, Mohan V, Hamlen KW, Lin Z. Securing untrusted code via compiler-agnostic binary rewriting. In: Proc. of the ACSAC. ACM Press, 2012. 299–308.

[320] Zhang M, Sekar R. Contol flow integrity for COTS binaries. In: Proc. of the USENIX Security. 2013. 337–352.

[321] Batyuk L, Herpich M, Camtepe SA, Raddatz K, Schmidt AD, Albayrak S. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within Android applications. In: Proc. of the MALWARE. 2011. 66–72.

[322] Zhang C, Song C, Chen KZ, Chen Z, Song D. VTint: Protecting virtual function tables' integrity. In: Proc. of the NDSS. 2015.

[323] Xu Z, Zhang J. A test data generation tool for unit testing of C programs. In: Proc. of the QSIC. 2006. 107–116.

[324] Wu R, Wen M, Cheung SC, Zhang H. ChangeLocator: Locate crash-inducing changes based on crash reports. Empirical Software Engineering, 2017,23(5):2866–2900.

[325] Gao Q, Xiong Y, Mi Y, Zhang L, Yang W, Zhou Z, Xie B, Mei H. Safe memory-leak fixing for C programs. In: Proc. of the ICSE. 2015. 459–470.

[326] Xuan J, Martinez M, De Marco F, Clement M, Marcote SL, Durieux T, Le Berre D, Monperrus M. NOPOL: Automatic repair of conditional statement bugs in Java programs. IEEE Trans. on Software Engineering, 2017,43(1):34–55.

[327] Briand LC, Daly JW, Wust J. A unified framework for cohesion measurement in object-oriented systems. Empirical Software Engineering, 1998,3(1):65–117.

[328] Briand LC, Daly JW, Wust JK. A unified framework for coupling measurement in object-oriented systems. IEEE Trans. on Software Engineering, 1999,25(1):91–121.

[329] Ethereum. 2018. https://www.ethereum.org/

[330] The DAO attacked: Code issue leads to 60 million ether theft. 2018. https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/

[331] Accidental bug may have frozen $280 million worth of digital coinether in a cryptocurrency wallet. 2018. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html

[332] Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proc. of the CCS. 2016. 254–269.

[333] Nikolic I, Kolluri A, Sergey I, Saxena P, Hobor A. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proc. of the CoRR. 2018. https://arxiv.org/pdf/1802.06038.pdf

[334] Hirai Y. Formal verification of deed contract in ethereum name service. Technical Report, 2016. 1–81.

[335] Bhargavan K, Delignat-Lavaud A, Fournet C, Gollamudi A, Gonthier G, Kobeissi N, Kulatova N, Rastogi A, Sibut-Pinote T, Swamy N, Zanella-Béguelin S. Formal verification of smart contracts. In: Proc. of the PLAS. 2016. 91–96.

[336] Bigi G, Bracciali A, Meacci G, Tuosto E. Validation of decentralised smart contracts through game theory and formal methods. In: Proc. of the Programming Languages with Applications to Biology and Security. 2015. 142–161.

[337] Kalra S, Goel S, Dhawan M, Sharma S. ZEUS: Analyzing safety of smart contracts. In: Proc. of the NDSS. 2018.

[338] Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Bunzli F, Vechev M. Securify: Practical security analysis of smart contracts. In: Proc. of the CoRR. 2018. https://arxiv.org/pdf/1806.01143.pdf

[339] Gurfinkel A, Kahsai T, Komuravelli A, Navas JA. The SeaHorn verification framework. In: Proc. of the CAV. 2004. 343–361.

[340] Masuda S, Ono K, Yasue T, Hosokawa N. A survey of software quality for machine learning applications. In: Proc. of the ICSTW. 2018. 279–284.

[341] Qin Y, Wang H, Xu C, Ma X, Lu J. SynEva: Evaluating ML programs by mirror program synthesis. In: Proc. of the QRS. 2018. 171–182.

[342] Sun Y, Wu M, Ruan W, Huang X, Kwiatkowska M, Kroening D. Concolic testing for deep neural networks. In: Proc. of the ASE. 2018. 109–119.

[343] Ruan W, Huang X, Kwiatkowska M. Reachability analysis of deep neural networks with provable guarantees. In: Proc. of the IJCAI. 2018. 2651–2659.

[344] Huang X, Kwiatkowska M, Wang S, Wu M. Safety verification of deep neural networks. In: Proc. of the CAV. 2017. 3–29.

[345] Ma S, Aafer Y, Xu Z, Lee WC, Zhai J, Liu Y, Zhang X. LAMP: Data provenance for graph based machine learning algorithms through derivative computation. In: Proc. of the FSE. 2017. 786–797.

[346] Gulzar MA, Interlandi M, Yoo S, Tetali SD, Condie T, Millstein T, Kim M. BigDebug: Debugging primitives for interactive big data processing in spark. In: Proc. of the ICSE. 2016. 784–795.

附中文参考文献:

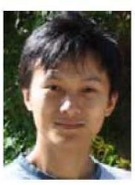
[1] 梅宏,王千祥,张路,王戟.软件分析技术进展.计算机学报,2009,32(9):1697–1710.

[97] 张健.精确的程序静态分析.计算机学报,2008,31(9):1549–1553.

张健(1969－),男,安徽庐江人,博士,研究员,博士生导师,CCF 杰出会员,主要研究领域为自动推理与约束求解,软件测试及分析.

张超(1986－),男,博士,副教授,博士生导师,CCF 专业会员,主要研究领域为软件与系统安全.

玄跻峰(1984－),男,博士,教授,博士生导师,CCF 专业会员,主要研究领域为软件分析与测试,软件数据分析,基于搜索的软件工程.



熊英飞(1982－),男,博士,研究员,博士生导师,CCF 专业会员,主要研究领域为软件工程,程序设计语言.

王千祥(1970－),男,博士,技术专家,CCF 高级会员,主要研究领域为软件工程,软件分析.

梁彬(1973－),男,博士,教授,博士生导师,CCF 专业会员,主要研究领域为信息安全,软件分析.

李炼(1977－),男,博士,研究员,博士生导师,CCF 专业会员,主要研究领域为程序分析,软件工程.

窦文生(1984－),男,博士,副研究员,CCF 专业会员,主要研究领域为软件工程,程序分析.

陈振邦(1981－),男,博士,副教授,CCF 专业会员,主要研究领域为程序分析,形式化方法.

陈立前(1982－),男,博士,副教授,CCF 专业会员,主要研究领域为程序分析与验证,抽象解释.

蔡彦(1986－),男,博士,研究员,CCF 专业会员,主要研究领域为软件测试,程序分析.