软件安全漏洞检测技术

李舟军[1)] 张俊贤[1)] 廖湘科[2)] 马金鑫[1)]
[1)](北京航空航天大学计算机学院 北京 100191)
[2)](国防科学技术大学计算机学院 长沙 410073)

摘 要 软件安全漏洞检测技术是提高软件质量和安全性、减少软件安全漏洞的重要方法和基本手段,受到学术界和工业界的广泛关注和高度重视.其主要途径包括软件测试、程序分析、模型检验与符号执行等.近年来,综合利用多种研究方法和技术手段来检测软件安全漏洞已成为软件安全领域的研究热点.文中首先回顾了程序分析与软件安全漏洞检测的基本概念、核心问题和传统手段.然后重点介绍该领域的最新进展,主要包括轻量级动态符号执行、自动化白盒模糊测试以及其实现技术和相应的工具.最后,指出了其所面临的挑战和发展趋势.

关键词 安全漏洞;静态分析;动态分析;符号执行;白盒测试
中图法分类号 TP309 **DOI 号** 10.3724/SP.J.1016.2015.00717

Survey of Software Vulnerability Detection Techniques

LI Zhou-Jun[1)] ZHANG Jun-Xian[1)] LIAO Xiang-Ke[2)] MA Jin-Xin[1)]
[1)](*School of Computer Science and Engineering, Beihang University, Beijing* 100191)
[2)](*School of Computer, National University of Defense Technology, Changsha* 410073)

Abstract Software vulnerability detection is one of the most important methods to improve software quality and is the key to insuring software security, which leads grant concerns from researcher and industry. Its main content includes software testing, program analysis, model checking and symbolic execution etc. In recent decade years, how to utilize various classic methods synthetically to detect software vulnerability holes is becoming a new hot research direction. This paper reviews the basic concepts, key challenges and some classic solutions of software vulnerability detection, and beyond this, it tries to introduce some new promising improvement in this research area, including lightweight dynamic symbolic execution, automatic white-box fuzz testing, their implementation technologies and corresponding tools. In the last, it provides here a survey of some key challenges and new research trends in future research work.

Keywords software vulnerability; static analysis; dynamic analysis; symbolic execution; white box testing

1 引 言

软件质量是软件产品的生命线,由于不合理的设计和软件开发人员的疏忽而引入的软件缺陷是致使软件品质下降的根源.软件缺陷导致软件运行时失效,软件故障的频发会带来极大危害.以美国为例,软件故障每年造成几十亿美金的经济损失[1],严

收稿日期:2014-02-18;最终修改稿收到日期:2014-10-31.本课题得到国家自然科学基金(61170189,61370126,90718017,60973105)、高等学校博士学科点专项科研基金(20111102130003)资助.**李舟军**,男,1963 年生,博士,教授,博士生导师,主要研究领域为形式化方法与技术、网络与信息安全技术、数据挖掘.E-mail:lizj@buaa.edu.cn.**张俊贤**,男,1981 年生,博士研究生,主要研究方向为信息安全、程序分析.**廖湘科**,男,1963 年生,硕士,教授,博士生导师,主要研究领域为并行与分布式操作系统、安全操作系统.**马金鑫**,男,1986 年生,博士研究生,主要研究方向为信息安全、程序分析.

重时甚至危及人们的生命安全.

由于某些特定程序缺陷的存在,计算机程序在运行时刻会出现一些设计时非预期的行为,其非但不能完成预期的功能,反而会出现意料之外的执行状况.这种预期之外的程序行为轻则会损害程序的预期功能,重则会导致程序崩溃,使其不能正常运行.更为严重的情况下,与安全相关的程序缺陷可以被恶意程序利用,使程序宿主机器受到侵害.前两种情况还只是损害程序本身的质量和可靠性,后者却可能致使系统被黑客程序控制,以至于泄露与程序本身完全无关的信息,如银行账号等私密数据.我们把这种导致软件系统出现安全性问题的缺陷称为软件的安全漏洞.

安全漏洞检测技术是提高软件质量和安全性、减少软件安全漏洞的重要方法和基本手段,受到学术界和工业界的广泛关注和高度重视.为此,学术界和工业界投入了大量人力和科研经费,从不同角度提出各种检测方法和技术,并开发了相应的检测工具.随着对问题研究的深入,新的方法、技术和工具不断出现,一种发展趋势是结合多种方法、综合利用诸多新出现的技术进行软件安全漏洞检测.本文首先回顾了程序分析与软件漏洞检测的基本概念、核心问题和传统手段,然后重点介绍该领域的最新进展,主要包括轻量级动态符号执行、自动化白盒模糊测试以及其实现技术和相应的工具.最后,指出了其所面临的挑战和发展趋势.

2 问题陈述

计算机程序代码是符合程序设计语言文法规则的有穷长度字符串,它以文本形式描述程序运行时预期出现的一系列计算行为.计算机程序总是在特定环境下执行,这个环境包括程序输入、内存状态、系统配置等等,由于执行环境的易变性,程序每次执行过程中出现的行为及其顺序通常并不完全相同,所有可能出现的程序行为和可能达到的程序状态构成计算机程序的语义域.

早期的研究更多集中于如何确保程序实现了预期的功能,即程序正确性问题,并在这个领域取得了许多重要的理论成果,如程序正确性证明等.比程序正确性弱一点的问题,则是关注如何确保程序在任意执行环境中不会出现损害程序和系统安全的行为,如除零错误和溢出错误等,解决这些问题的关键在于如何定性和定量地描述软件的行为并刻画其语义性质,这是软件理论的基本问题之一.软件的静态语法与其动态语义的分离是造成软件行为难以描述和推理的根本原因[2].为描述计算机程序的行为并研究其语义性质,研究者提出了多种不同的语义模型,并发展为一个专门的研究领域,其中被使用最多的有操作语义、指称语义和公理语义等.以某种特定的程序语义性质描述方式为前提,安全漏洞检测问题可以重新描述为:给定程序和需要检测的安全性质,设计算法能够回答:程序中所有可能出现的行为和状态是否都满足待检验的安全性质,若程序在某次执行过程中存在违反该安全性质的行为,则说明存在相应的安全漏洞.

3 理论和方法

判断程序是否满足某种安全性质是一个逻辑命题,程序正确性证明一般使用 Hoare 逻辑[3-4]等公理系统从语法推导的角度证明程序的公理语义是否满足待检验的安全性质;模型检验则使用有穷自动机表示程序的状态迁移系统,并从语义的角度验证所建立的状态迁移系统是否为待检验性质的一个模型.基于程序正确性证明方法开发的工具通常需要事先给出程序的安全性规约,是一件费时费力的事情;另一方面,模型检验只能应用于有穷状态系统,且存在状态爆炸问题.程序分析是指,对软件进行人工或者自动分析,以验证、确认或发现软件性质(或者规约、约束)的过程或活动[5].将程序分析的方法应用于软件安全性检测,可有效发现和检测软件中存在的安全缺陷或漏洞.

1953 年 Henry Rice[3] 证明,在普遍意义上,程序分析不可能完全判定程序的任意非平凡性质,即无法构造出停机的通用算法,能够对任意给定的程序判定其是否满足指定的性质.尽管通用算法不存在,但是若对问题稍加限制,在有限范围内解决这一问题却是可行的.经过不断努力和尝试,科研人员提出了多种程序分析方法和支撑工具,可根据对程序安全性的实际需求从不同的角度和程度解决问题.

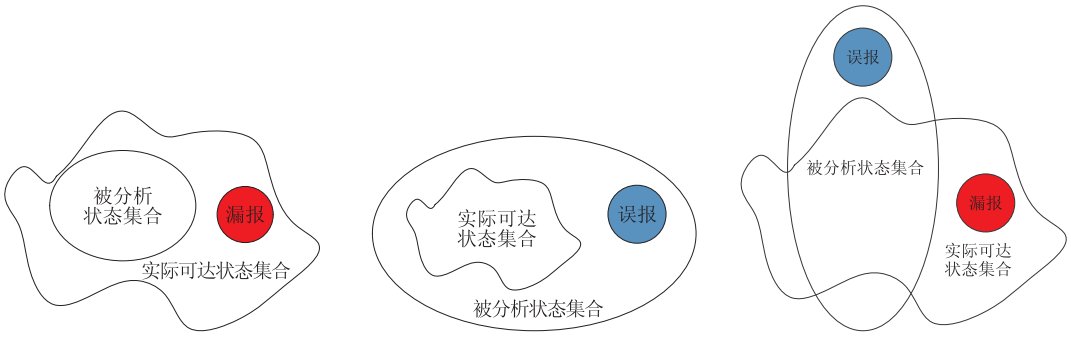
评价程序分析与软件安全漏洞检测方法和工具的优良与否有多种指标,其中最重要的是误报率和漏报率[5],分别表示误报和漏报现象发生的概率.误报是指算法报告了实际不存在的错误,漏报是指算法遗漏了本来存在的错误.

为了对各种程序分析方法做较全面的了解,对

其进行合理的分类是必要的，常见的分类方式有以下两种：一种按照程序代码的文本形式分为二进制和源代码分析，这种分类法更多是从技术实现的角度出发；另一种按照是否运行程序代码，将程序分析方法分为动态和静态分析，这种分类方式更容易说清漏报和误报的发生根源.

静态分析方法从语法或语义的层面分析程序文本（源代码或二进制），以推导其语法或语义性质. 其目的较偏重于说明程序不包含某种错误，多数静态方法存在误报，有的也同时存在漏报. 动态分析方法通过运行待测程序以获取和分析程序运行过程中产生的动态信息，以判断其运行时语义性质. 由于运行时动态信息是准确的，因此动态分析方法本质上不存在误报，但是由于很难完全枚举程序所有的执行情况，动态分析方法通常存在漏报. 需要指出的是，二进制分析多采用动态分析方法，源代码分析多采用静态分析方法.

对动、静态方法引起漏报和误报的原因做深入的考察则可以发现，动态分析只获取程序的实际可行路径和可达状态，这是保证分析结果没有误报的根本原因. 但由于其大多时候并不能遍历程序的所有可行路径，因而可能错过了某些可以引发程序错误的执行路径，进而导致漏报，如图 1(a)所示. 换言之，动态分析方法对程序的实际可达状态做下近似处理(under-approximation)[6]. 大部分静态分析方法为了建立用于分析的模型需要对程序的动态语义做某种形式的抽象，其抽象结果难免会引入实际不可行路径和不可达状态，如图 1(b)所示，而静态分析方法难以在有限时间内判定抽象路径的可行性，这是导致误报的主要原因. 换言之，这类方法对被分析程序的实际可达状态做上近似处理(over-approximation)[7]. 也存在一些静态方法同时做上近似和下近似，因而同时引入漏报和误报，如图 1(c)所示.



(a) 下近似导致漏报　　(b) 上近似导致误报　　(c) 同时有误报和漏报的近似

图 1　不同近似方式引起漏报和误报

总而言之，各种具体的分析方法均须隐式或显式地对被分析程序的可达状态集合建立分析模型，这个模型总可以被视为是对实际语义域的上近似或下近似. 清楚区分分析方法对程序可达状态集的近似方式，是考察各种分析方法是否存在误报和漏报的根本角度. 从这点出发，本节后续部分将分别对软件测试、静态程序分析和将程序性质表达为逻辑公式的方法做介绍和对比，图 2 给出各种方法分类.

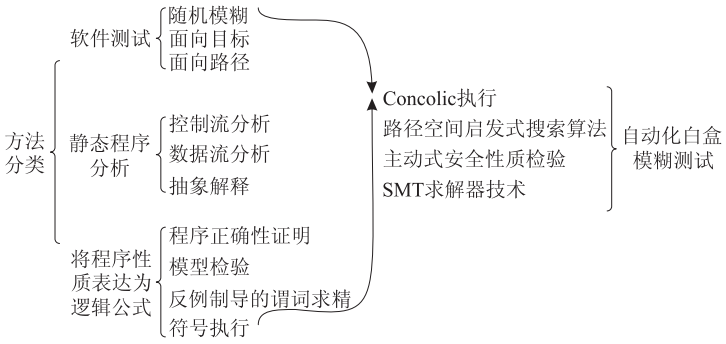


图 2　方法分类

3.1　软件测试

迄今为止工业界采用最多的依然是软件测试方法，据统计，在美国每年耗费在测试上的费用高达几十亿美金，在那些对软件质量要求较严格的公司，项目开发总投入中花费在测试上的部分甚至达到 50%[1,6]. 软件测试目的有多种，功能性测试考察程

序是否正确实现了预期的功能;可靠性测试则尽量选取广泛的测试数据运行程序以观察程序是否出现异常行为;由于软件开发是个进化的过程,期间会不断产生新的程序版本,回归测试用于检验新版本是否保留了旧版本的正确部分;安全性测试则可检测并识别程序中潜在的安全性缺陷,并验证该缺陷是否可被利用进而导致系统出现安全性问题.测试需要准备测试用例,为了考察测试用例集合是否能够真正说明问题以及能在多大程度上说明问题,研究者从覆盖率的角度提出多种测试用例的评价标准,主要包括语句覆盖率、分支覆盖率、分支组合覆盖率和路径覆盖率.手工创建测试用例需要测试人员花费大量精力和时间,因此如何设计算法自动生成测试用例以减轻测试工作的负担,成为测试方法研究的核心问题之一.

1996 年 Ferguson 和 Korel[8-10] 将自动化测试用例生成的方法分为三类:面向目标(Goal-Oriented)的测试用例生成、面向路径的(Path-Oriented)测试用例生成以及随机化模糊测试(Random Fuzz).前两种方法通过考察程序内部结构来选择测试数据,以覆盖某条指定语句或程序执行路径,因此也被称作结构化测试或白盒测试.

随机测试又称为黑盒模糊测试[11],是一种将大量随机数据输入到目标程序中,通过监视程序运行过程中的异常来挖掘软件漏洞的方法.由于模糊测试不需要了解程序源代码,能较快地定位和确认安全漏洞,已成为软件安全性检测的重要方法和有效手段.自模糊测试方法提出以来,已挖掘出大量的高危漏洞,因而吸引了安全领域的研究者和公司的高度重视,各种模糊测试通用框架和专用工具不断涌现.限于篇幅,有兴趣者可参考文献[11-22].需要指出的是,极小语义缺陷是指只能由程序输入中极小比例数据揭示的软件缺陷[23],而模糊测试则很难发现此类缺陷.

如图 3 所示,典型的测试数据生成系统包含三个部分:程序分析器、路径选择器和测试数据生成器.程序分析器对源代码作预处理分析,产生必要的分析结果提供给路径选择器,较重要的分析信息包括过程调用图、数据依赖图和控制流图等;路径选择器以提高某种覆盖率为目标,根据路径选择策略选取适当的程序路径并提供给测试数据生成器使用,路径选择问题对整个测试数据生成过程都有极大影响[10];测试数据生成器根据这些路径生成测试数据,有时也给路径选择器提供反馈信息,如发现某条路径不可行等.需要注意的是,测试过程本身是动态的,但测试用例的生成不必运行被测试的程序.

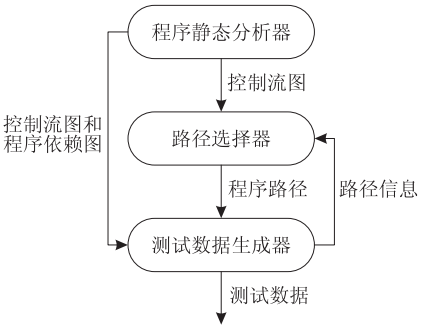


图 3 测试数据生成的基本框架

做为一种动态分析方法,测试不会产生误报,但存在漏报,正如 Dijkstra 所言,“测试可有效地揭示程序包含错误,但并不能确保程序无错”.相对于静态分析的高误报率,很多公司为保证软件开发效率而容忍少量缺陷的存在,即接受漏报,拒绝误报[6],这使得测试方法成为工业界保证软件质量的最常用方法.

3.2 静态程序分析和抽象解释

静态程序分析[24-25] 源自于编译优化技术,它通过分析程序代码来求解关于程序特定性质的问题,是多种分析技术的集合,其中最基本的两类是控制流分析和数据流分析.控制流分析[25] 提取代码中的控制结构,以控制流图作为对程序中分支跳转关系的抽象,描述程序的所有可能执行路径.数据流分析[24] 内容较多,大致可以按照三个角度考察:流敏感/流不敏感,路径敏感/路径不敏感,上下文敏感/上下文不敏感.流敏感的分析考虑程序中语句的顺序,举例来说,一个流不敏感的指针分析可能认为“变量 x 和 y 可能指向了同一位置”,而一个流敏感分析会认为“在某条语句执行后,变量 x 和 y 可能指向了同一位置”;路径敏感的分析考虑了程序的控制流,比如一个分支条件是 $x>0$,那么在条件不满足的分支下分析过程会假设 $x\leq 0$,而在满足条件的分支则会假设 $x>0$ 成立;上下文敏感的分析则处理过程调用,在分析目标函数的调用时,它将根据调用的上下文信息确保被调用函数能够返回到正确的调用点,而如果没有这种信息,返回时就必须考虑所有可能的调用点,因而丧失潜在的精度.

数据流分析以格做为问题的解空间模型,以格理论做为考察问题的角度,以不动点计算做为求解方法.很多时候,数据流分析所建立的解空间格模型过于庞大,致使问题的求解实际不可计算.Cousot P 和 Cousot R[26-27] 于 1977 年提出基于格理论的抽象

解释(Abstract Interpretation)方法，用于简化和逼近(Approximation)程序不动点计算，其方法使用另一个抽象对象域上的计算逼近程序指称的对象域(具体对象域)上的计算，使得程序抽象执行的结果能够反映出程序真实运行的部分信息，本质上是在计算效率和计算精度之间取得均衡，以损失计算精度换取实际计算的可行性[27].

基于语法制导的静态错误检查工具不需要运行程序，而是使用静态分析方法从语法角度分析程序代码，并报告可能的软件缺陷. 比较知名的工具有 Lint 工具族、Prefix、Coverity、Fortify 和 Klocwork 等，此类工具通常会产生大量误报[28].

3.3　将程序性质表达为逻辑公式的方法

将程序性质表达为逻辑公式的方法在 20 世纪 60 年代就已经被 Floyd[29] 和 Hoare[4] 提出，是程序正确性研究的主要方法，研究者在已有成果的基础上不断做出新的发展，本节介绍其中比较重要的 4 种方法.

3.3.1　程序正确性证明

程序正确性证明试图从语法推导的角度证明程序满足其功能规约. 程序的功能规约是一对谓词公式，包括前置谓词和后置谓词，它们用内涵的方式分别定义了程序执行前后的可能状态集合. 由于造成状态变迁的唯一原因是程序的执行，因此这样一对谓词公式足以描述程序的功能.

程序的正确性包括部分正确性和完全正确性，区别在于是否保证程序终止. 假设程序从满足其前置谓词所的某一状态开始执行，若能终止执行则其终止状态必满足其后置谓词，则称程序满足部分正确性. 以同样假设为起点，若程序必然终止且其终止状态满足其后置谓词，则称程序满足完全正确性. 可用于证明程序部分正确性的方法有 Floyd 的不变式断言法、Manna 的子目标断言法和 Hoare 的公理化方法[30]；可用于证明程序终止性的方法有 Floyd 的良序集方法、Manna 等人的不动点方法和 Knuth 的计数器方法. 为证明程序的完全正确性，Manna 和 Pnueli 对 Hoare 的公理化方法做出推广，另外 Dijkstra 提出最弱前置谓词转换方法以及验证方法. 需要说明的是，Hoare 逻辑是一个完整的程序逻辑系统，每个公式由一个程序语句及其前后置断言构成，其理论只包含一条赋值公理，其形式演算系统在一阶谓词逻辑的基础上分别为顺序、分支和循环指令添加了三条演算规则. 已经证明，Hoare 逻辑具有可靠性和相对完备性，但其形式演算系统半可判定.

程序正确性证明的主要问题有两个，一是需要程序设计人员使用逻辑公式描述程序的功能规约；二是需要为循环体寻找循环不变式.

3.3.2　模型检验

模型检验方法在计算机硬件、通信协议、控制系统、安全协议的分析与验证中，取得了令人瞩目的成就[31]，并且可以验证程序是否满足指定的安全性质. 它使用有穷状态迁移图对程序的行为建模，同时使用模态或时序逻辑公式描述待检验的安全性质，程序的每条执行路径及其相应的状态对应于迁移图中的一条状态迁移轨迹. 模型检验方法通过穷举状态迁移图中每一条状态迁移轨迹，逐次判定该轨迹上的所有状态是否满足待检验性质，若都满足则认为程序满足了待检验性质，也即获得了程序满足待检验安全性质的一个证明，否则模型检验器给出使性质为假的系统状态迁移轨迹做为反例.

在软件验证领域，模型检验方法的主要困难是状态爆炸问题，目前的解决思路有四类：一类是抽象方法，通过深度发掘所建立模型的状态空间结构特征和相关知识对模型进行抽象(编码和压缩)，消减原模型中与待检验性质无关的信息，以获取模型的简约表示，以最终减少计算代价，抽象方法包括符号化模型检验、对称模型检验和偏序模型检验等. 需注意的是，抽象必须是保持性质的，即若简化的模型满足性质，则原来的模型亦满足该性质；一类在生成模型的同时检验安全性质，绕过状态空间过大的问题，如 On-the-fly 方法；另一类对程序中的循环结构做有限次展开，并利用 SAT 技术只对有限长度轨迹做检验，如限界模型检验；最后一类是组合验证方法，基于分而治之的思想把整个模型分解为可处理的子模型，先验证各个子模型的局部性质，并可保证当各个子模型都满足其局部性质时，整个模型必然满足待检验的安全性质.

3.3.3　反例制导的谓词抽象求精

谓词抽象是一种特殊的程序抽象方法，它以一组谓词为准则对程序进行约减，只保留程序的控制流和程序中涉及到谓词集合中所包含的自由变量的语句，得到原程序的一个简化版本，可以看作原程序在谓词集合上的投影. 抽象后所得程序的可达状态集合是原程序可达状态集合的超集.

反例制导的谓词抽象求精方法[7,32]认为程序的抽象模型的建立是一个逐步逼近的过程. 首先根据初始谓词集合对程序进行谓词抽象，并通过对抽象

模型的迭代求精逐步建立程序的精确模型.对每次迭代得到的抽象模型进行模型检验,若存在反例,则根据其反例路径在原程序中找到对应的程序路径,并判断该路径的可行性.如果该路径可行,则该反例为真实的,否则即为虚假反例.根据虚假反例产生的原因,指导谓词集合的进一步求精,进而得到一个不再含有该虚假反例的新的抽象模型.重复这一过程,直到模型不再存在反例.

在模型检验技术中,使用谓词抽象能够对程序状态空间进行有效压缩,很大程度上缓解了模型检验所面临的状态空间爆炸问题.在基于这种方法开发的工具中,比较著名的有微软的 SLAM[7,32] 和 CMU 的 Blast 等,其中 SLAM 被用做检验 Windows 驱动程序可靠性的基本工具之一.但由于抽象过程是对程序实际可达状态的上近似,这种方法不可避免地引入误报问题,如 SLAM 的误报率高达 30%[33].

3.3.4 符号执行

最早提出符号执行概念的文献首见于 1969 年,随后即获得研究者大量关注[34-36],但是直到最近十几年才开始逐渐步入实用化[37-39],这种方法的核心在于:通过描述程序的执行路径和伴随的状态变化来表示程序语义并揭示程序的内部结构.程序测试和程序正确性证明可被视为两个极端.测试容易实现自动化,但即使被测试程序能够正确执行测试用例并产生预期的结果,也并不能保证没有被测试到的数据都能使程序正确工作;另一方面,程序证明不需要运行程序,只需要证明程序所有可能的执行情况都满足程序规约,但程序规约需要程序的设计者理解程序的功能需求,只能手工注释而难以实现自动化生成.符号执行可被视为介于二者之间的一种实际可行的折中方法,程序不以实际的数据作为输入,而是将输入向量空间做等价划分,以符号变量所表达的等价类作为输入,并"符号化"执行程序.每次符号化执行相当于该等价类中的数据均被执行一次,而一次普通执行则可被视为一次符号执行的一个特例,可以说符号执行用新的方式捕获程序的执行语义,这是对普通执行的扩展和一般化.需要注意的是,等价类的划分由程序对输入的控制依赖所决定,由于循环和递归等程序结构的存在,很多时候等价类的划分是无限的,所以在有限时间内要穷尽所有情况通常是不可能的[34].

对于符号执行而言,执行路径、路径约束和执行树是其核心概念.假设程序只由分支语句集合 C、赋值语句集合 A、*abort* 语句(异常终止,对应于程序错误)、*halt* 语句(程序正常终止)构成,则执行路径是一个只包含以上 4 种语句的有穷序列 $(A \cup C)^* (abort|halt)$[6],若将每条语句视为一个树节点,则全部执行路径构成程序的符号执行树,其中程序入口为执行树的根节点,每条赋值语句对应的树节点只有一个后继结点,每条分支语句对应的树节点有一或两个后继结点,*abort* 或 *halt* 对应的树节点为执行树的叶子节点.

符号执行使用符号变量表示程序输入参数,执行过程中使用符号值代替实际值作为程序输入数据,并使用符号表达式(Symbolic Expression)描述程序变量取值情况,于是程序的执行结果可以被表示为符号变量的函数.符号执行过程为每个符号变量维持一个符号状态(Symbolic State),它将符号变量映射为符号表达式以及与此相关的路径约束 PC (Path Condition).其中,路径约束是一个以符号变量为自由变量的、不带量词的一阶谓词逻辑公式.路径约束沿着当前执行路径收集关于符号输入的约束,每当遇到一条赋值语句,则修改被赋值变量在符号状态中所对应的符号表达式,而每当遇到一条分支语句如:if (e) $S1$ else $S2$,分支条件 e 的取值决定执行路径应选择的分支,路径约束按照所选分支得到更新.若为 then 分支,则路径约束 PC 更新为 $PC \wedge \sigma(e)$;若为 else 分支,则路径约束 PC 更新为 $PC \wedge \sigma(\neg e)$,其中,$\sigma(e)$ 表示在当前符号状态 σ 下对 e 符号求值的结果.具体执行在分支节点的两条分支只能选取其中之一,与之不同的是,符号执行在分支节点可以同时选取两者,并因而得到两组不同的路径约束.当一条路径符号执行结束时,使用谓词公式判定过程对所获取的路径约束进行求解以得到一组具体输入值,若有解,则此解可作为这条路径所对应等价类的代表元,以此作为测试用例重新执行程序,则程序必定会沿相同路径执行.由于路径约束是以谓词公式的形式存在,自然需要考虑其可满足性问题,可满足的路径约束对应于某条实际可行路径,不可满足的路径约束则对应于不可行路径.一阶谓词逻辑公式的可满足性问题属于半可判定问题,因此路径可行性问题同样属于半可判定问题.

早期的基于符号执行的测试生成只针对源代码作分析,属于静态方法,其过程首先使用静态程序分析方法中的控制流分析构造控制流图(Control Flow Graph),之后通过选取控制流图中某条路径并针对此路径作符号执行和路径约束求解生成测试用例.实践表明,静态符号执行选取的路径中很大一部分是实际不可行路径,这是导致基于静态符号执

行的测试方法存在误报的根源.另一方面,程序代码中的分支语句的存在决定了程序的路径数相对于程序的规模必然呈指数型增长,不仅如此,代码中包含的循环和递归结构更可能导致路径数目达到无穷,这种情况被称为路径爆炸问题,是符号执行的核心问题之一[37-39].

4　自动化白盒模糊测试

多年来,研究者和产业界对基于各种方法开发的支撑工具获取了大量研究和使用经验,通过综合对比,对各种方法的优势和劣势有了比较清晰的认识.2005 年 Bell 实验室的 Godefroid 等[6]学者提出了基于轻量级动态符号执行的自动化白盒模糊测试方法,近十年来得到了学术界大量关注,是本领域当前的研究热点之一,研究者基于这种方法开发了多种工具如 DART[6]、CUTE[40-42]、KLEE[43] 和 SAGE[44-45] 等,初步经验表明这是很有前景的方法和技术.新方法借鉴了动态和静态分析的经验和不足,在更深层次上对两类方法进行结合,其核心内容为:

(1) 对静态符号执行进行扩展,混合具体执行,提出轻量级动态符号执行方法[6,40]:Concolic 执行;

(2) 借助于逐渐实用化的 SMT 求解器(Satisfiability Modulo Theory Solver)技术对路径约束求解;

(3) 使用新的启发式算法探索程序的路径空间(程序执行树)[44-46].

4.1　整体框架

图 4 给出总体框架图,新方法结合了传统的模糊测试,使用模糊器产生随机输入数据并 Concolic 执行需要测试的程序,同时使用 SMT 求解器检验所执行的程序路径是否满足要验证的性质.每次执行结束后,使用启发式算法搜索路径空间,并在搜索结果中按照路径选择策略选取下一条需要执行的路径,迭代整个过程,直到所有路径执行完毕或者系统运行时间超过了提前设置好的时间阈值.

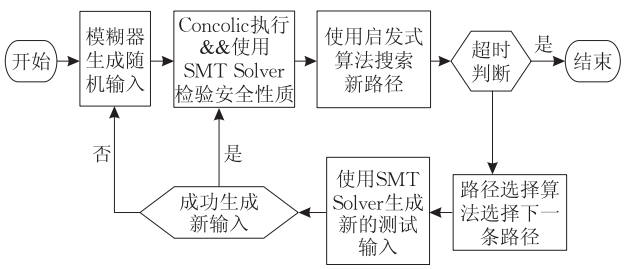


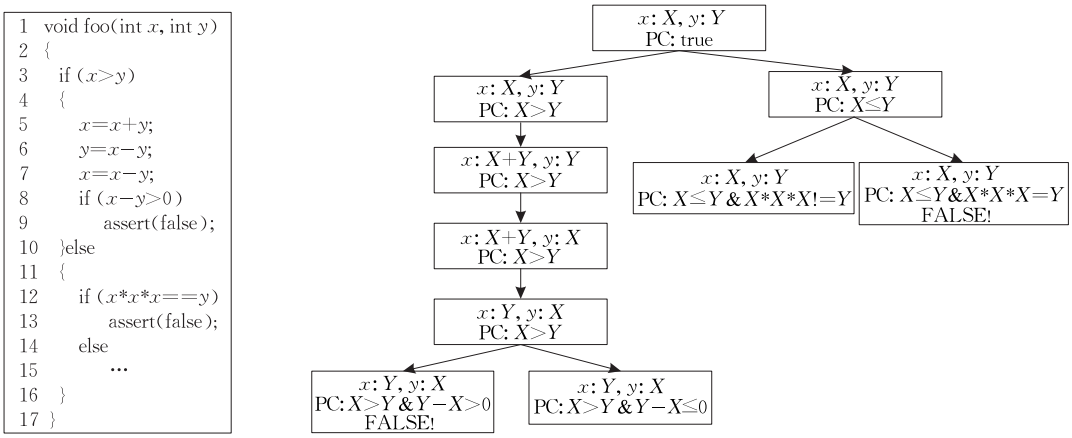
图 4 总体框架图

4.2　轻量级动态符号执行方法:Concolic 执行

基于具体执行的动态分析工具如 Valgrind[47] 或 Purify[48] 等只针对某次具体执行实例作分析,由于具体执行是精确的,因而不产生误报.此类工具的缺点在于过于依赖所使用的具体输入的典型性,只有在所选取的具体输入能够触发程序错误的情况下,动态分析工具才能捕获错误,程序的输入空间过大导致程序输入的选取不可能覆盖整个空间,而只能是其中一部分,这是导致动态分析工具产生漏报的根源.符号执行由于实质上是以执行树对程序输入空间做等价划分,并以每组等价类作为程序输入,所有能使程序沿同一条路径执行的输入落入同一等价类中,因此能够对输入数据做到最大的覆盖,但其作为一种静态方法,静态符号执行不能获取精确的运行时信息,其分析目标包含了实际不可行路径,导致误报的产生.

Concolic 执行在动态与静态分析技术之间做出了新的取舍和折中,混合两种执行方式,在具体执行的同时对所执行到的代码施行符号执行,具体执行的特性决定了每次 Concolic 执行获取的路径都是可行路径,因此避免了误报,这是 Concolic 执行相对于静态符号执行的主要优势之一.一次 Concolic 执行结束时,其路径约束是一组约束的合取,对其中某个或某几个约束取反的同时保持其余部分不变,则可以得到路径约束的一个变体,利用 SMT Solver 求解这个变换后的新约束,若有解,则意味着它可能对应于另一条可行路径.理论上,若程序不存在无限路径(没有死循环和无穷递归),迭代此过程可以遍历执行树中所有路径.

Concolic 执行使用 SMT Solver 求解路径约束,尽管近年来 SMT Solver 有了长足发展,逐渐步入实用化,但其能够求解的约束类型依然有限,比如,就目前所知并没有发现任何一款求解器能够完全支持非线性算术理论.在程序执行到某条分支时,若其产生的约束超出了求解器的求解能力范围,传统的静态符号执行就会卡壳,于是丢弃掉这条路径,进而可能导致漏报.而 Concolic 执行在此处则会使用当前的具体值代替符号值,因而得到一个不完全的、包含了具体值的简化的符号表达式,这就使简化后的路径约束落回到 SMT Solver 所支持的理论中,这样一方面降低了 SMT 求解器的使用限制,另一方面则保证算法能够继续运行进而,提高了代码覆盖率,这个过程被称为路径约束的部分具体化,是 Concolic 相对于静态分析方法的另一个主要优势.为了更清楚地阐述,考虑图 5(a)所示 C 代码.



(a) 示例C程序片段 (b) 对应的符号执行树

图 5 程序运行示例图

图 5(b)为相应的符号执行树,假设第 1 次运行这段代码时,随机生成 $x==2$,$y==33$,执行到第 12 行时,分支条件 $x*x*x==y$ 不满足,因此下一步选择 else 分支,因此第 1 次执行遗漏了 then 分支(第 13 行).第 1 次执行结束后,我们期望得到满足 $x*x*x==y$ 的 x 和 y 以在之后的执行中覆盖第 13 行,然而,$x*x*x==y$ 是非线性约束,超出了求解器的求解能力范围,静态符号执行在此时会丢弃掉这个分支,因而导致漏报. Concolic 由于混合了具体执行,可以获取 x 和 y 在上一次运行中的具体取值分别为 2 和 33,如果固定 x 的值,则可以得到部分具体化的路径约束 $8==Y$,对其求解可得:$y==8$.第 2 次运行时使用 $x==2$,$y==8$ 作为新的测试输入,使第 13 行得到运行,触发(并捕获)了程序运行时错误.

4.3 路径空间启发式搜索算法和路径选择

自动化白盒测试过程遍历程序执行树中的路径,并针对每条路径生成测试数据,由于路径爆炸问题的存在,完全遍历整个路径空间常常是不可能的,因此如何设计启发式算法,在采用某种覆盖率标准的前提下,尽可能覆盖到更多的代码和最可能触发程序错误的路径就变得极为重要.研究者提出了多种启发式算法,这里简单介绍其中较有影响的两种:对路径约束变异和对基本块覆盖打分方法.

(1)对路径约束变异. Concolic 执行在每次执行完毕一条程序路径后会得到一组路径约束,DART 和 CUTE 对其中最后一个约束取反以产生新的路径约束,实质上是对符号执行树做深度优先搜索,这意味着对一条路径上的非叶子节点的符号执行将会执行多次,同样,对相同的中间性路径约束需要多次求解,这给符号执行引擎带来了很大负担. SAGE 对此改进并提出按代搜索算法,每得到一组路径约束后,按照不同组合对其中多个约束取反,因而可以同时得到多组新的路径约束,这种方法一方面极大减少了重复性计算,另一方面可以更方便地使用并行求解算法在多台机器上同时对路径约束进行求解.

(2)对基本块覆盖打分. 这种方法被 KLEE[43] 采用,程序在每个分支处使用打分函数给当前路径已经执行的部分设定分值,打分的依据可以为:距离未覆盖指令的距离、调用栈高度或被调用符号进程的最近一次执行是否覆盖了新基本块. KLEE 被设计为可以同时执行多条路径,可以视为对程序路径的并发执行,每次使用调度算法在现存的路径集合中选取一条执行,其选取的标准就是路径集合中分值最高者.这种方法借鉴了贪心算法的思想,即最重要的应该被优先处理.

4.4 安全性质检验

模糊测试能触发很多类型的程序错误,导致程序崩溃,但在有的情况下出现的异常行为并不会使程序崩溃,如数组溢出等,这种隐蔽的程序漏洞对黑客更有价值.异常行为的难以观察导致模糊测试工具错过了发现漏洞的机会,对程序出错的类型进行总结和深入理解,获取能够刻画程序错误本质的安全性质,并检验一组输入是否破坏了此性质是解决此问题的基本思路.

传统的模糊测试通常结合 AppVerifier 或 Purify[48] 等动态分析工具进行运行时性质检验,由于使用的是具体执行,这种方式一次只能针对一组具体输入检查其是否破坏了安全性质,因而被称为被动式运行时性质检验. Godefroid 等人[49] 对被动运行时检验方法进行扩展,提出了主动式运行时检验的概念,期望能够检验沿同一路径执行的所有输入是否都满足待检验性质.其方法为:在待检验程序点注入描述安全性质的运行时符号约束,若此约束

的否定可被求解，则求解器生成的测试输入即为能破坏安全性质的反例. 在模型检验中，针对特定系统，确定需要检验哪些安全性质才能使人信服的度量系统的可信程度和安全质量，是一件需要花费精力仔细思考的事情. 而对于软件漏洞检测而言，该问题得到了很大程度的简化，待检验安全性质通常来自于对已知漏洞类型和模式的总结.

较典型的安全性质包括数组越界访问、空指针引用、除零错误、整数溢出、栈溢出等等. 以除零错误为例，若一次混合符号执行过程中遇到一条除法指令，且本次执行中除数不为零，主动式安全性质检验将产生安全性质约束并向求解器询问，是否存在一组输入，使程序沿本次执行路径执行时使除数为零？再以数组越界访问错误为例，设数组上下界分别为 lb 和 hb，数组索引变量为 i，其中 lb 通常为 0，hb 可以为固定值(静态分配)，或者为变量(动态分配)，求解是否存在一组输入使 $I<LB$ 或 $I>HB$ 成立，其中 LB、HB 和 I 分别为程序变量 lb、hb 和 i 所对应的符号变量，若存在这种输入，则表示程序存在数组越界访问错误.

安全性质检验伴随着符号执行，需要求解器的参与，检验的安全性质越多意味着算法运行的速度越慢. 若预先知道或者猜测待测试程序中某种特定类型的漏洞存在较多，或者我们最关心哪种类型的漏洞，则可以有针对性地检查相应的安全性质，这种轻量级的检验方法能够极大地减轻计算的负担，被 CatchCov 等工具采用[50].

5　实现技术

自动化白盒模糊测试的实现需要综合利用多种技术，内容较为繁多. 本文着重介绍其中的程序插装、程序执行环境模拟和求解器技术.

5.1　插　装

Concolic 执行和安全性质检验通过对程序的实际执行情况进行观察，以获取其运行时信息和动态语义，并以此为基础对其安全语义性质进行推理. 是否能够准确可靠地捕获程序执行时的动态语义信息，势必将影响到程序语义推理的有效性. 程序插装技术是实现这一目的的主要技术手段，本文主要介绍 4 种插装方式：

(1) 基于源代码

源代码插装技术可分为手动和自动两种，手动插装需要分析人员手工对代码插入附加代码，其优点是较为灵活，且不需要额外的技术支持，但程序规模过大时，插装工作会过于繁琐. 自动插装首先使用词法分析和语法分析对源代码进行处理，确定插装点，之后根据插装点的类型加入附加代码，典型工具如 Rational Rose，它通过修改被插装源代码所对应的抽象语法树以达到源代码到源代码的转换，实现插装目的.

(2) 基于中间代码

基于中间代码的安全漏洞检测通常借助于工具先将源代码编译为某种中间语言代码如 CIL[51]、LLVM[52]等，之后针对每条中间语言指令插入调用指令，对事先编写的符号执行引擎库的函数进行调用，最后将插装后得到的中间语言代码编译成可执行程序，在其实际执行过程中实现动态符号执行. 相比于源代码插装，这种方式的通用性得到较大提高，但必须以能够获取源代码为前提，因此不能较好地处理程序对未开源的第三方库的调用.

(3) 动态二进制插装

在不能获取程序源代码的情况下，可以借助于动态插装工具对可执行程序做动态二进制插装，比较知名的工具有 Valgrind[47]、Pin[53]等. 以这种方式工作的系统，如 BitScope[54]等，在监视程序实际执行的同时，捕获指令序列，根据指令的种类，调用预先编写的符号执行引擎和分析模块.

(4) 二进制离线插装

对于大型程序，一条执行路径的指令数可以达到数十亿条之多，这对分析工作带来极大困难，超出了求解器的正常工作能力范围，微软的 SAGE 工具针对这种情况，提出离线分析方式，大致步骤为：

① 抓取二进制执行轨迹；

② 对其静态插装；

③ 重放插装后轨迹并获取需要的约束条件.

5.2　程序执行环境模拟

测试的根本目的是通过生成不同的测试输入数据来控制和观察程序的执行，从程序设计的角度看，这些输入数据会以实参的形式传递给被调用的函数，但就实际执行的角度而言，多数程序输入来自于外部环境如键盘、磁盘文件、网络数据包等. 不与外部环境交互的程序是罕见的，数据表明使用面向对象语言设计的软件中 90.7% 的类会产生环境交互. 文献[55]指出，很多研究性项目声称的覆盖率很高是由于其选择的测试集太小且没有环境交互，环境交互能够极大影响覆盖率.

程序执行环境模拟技术通过建立虚拟的程序执行环境，模拟程序的外部输入并精确控制程序输入数据的内容，以达到测试目的. 从实现的角度来看，

环境建模主要有两种方式：(1) 虚拟机方法；(2) 重写库函数并截取系统调用的方法. S2E[56] 是以虚拟机形式实现环境建模的典型，这种方法能对程序执行的外部环境做到完全控制，灵活而强大，但对工程和技术的要求非常高，实现起来工作量很大. KLEE 是后者的典范，优点在于可以灵活选择环境建模的强度，从技术的角度说，实现起来比较灵活.

5.3 求解器

近十年来，布尔可满足性(Boolean Satisfiability, SAT)求解技术飞速发展，从 2000 年的 Grasp 到 2007 年的 Rsat，主流的 SAT 求解器的效率提高了近 200 倍[57]，目前 SAT 求解器可以高效地处理数百万变量的问题，已成功应用于模型检验、定理证明等领域. SAT 求解器用于判定命题逻辑公式的可满足性，命题逻辑的表达能力相对较弱，SMT 求解器以 SAT 求解技术为基础，引入基于一阶谓词逻辑的各种理论，将能力范围扩充到判定一阶谓词逻辑公式的可满足性. 表 1 列举 4 种比较常见的求解器，包括其支持的操作系统、支持的理论和 API.

表 1 求解器举例

求解器	操作系统	支持的理论和特性	API
CVC4	Linux Mac OS	整数线性算术 数组 元组 记录归纳数据类型 比特向量 未解释函数	C/C++
STP	Linux OpenBSD Windows Mac OS	比特向量 数组	C/C++ Python OCaml Java
uclid	Linux	空理论 整数线性算术 比特向量	
Z3	Linux Mac OS Windows FreeBSD	空理论 整数线性算术 整数非线性算术 比特向量 数组 数据类型 带量词的谓词公式	C/C++ .NET OCaml Python Java

6 关键性挑战和解决现状

Concolic 执行是对静态符号执行的发展，由于混合了动态执行和静态符号执行，传统意义符号执行方法自身难以克服的诸多难题也同时被引入到新的方法，如何面对并解决这些问题，是决定 Concolic 方法是否能获得成功的关键，以下就其中研究的一部分热点问题简要介绍：

(1) 对指针和数组的处理. 早期的静态符号执行工具较少对指针和数组建模，如何对程序中的指针和数组进行建模是符号执行工具必须要处理的难题之一. DART 使用具体执行绕过了这一问题，但是具体执行是对程序可达状态集合的下近似，因而存在漏报. CUTE 在 DART 的基础上针对指针建立了较简单的处理，支持指针之间相等和不等关系的比较. EXE[58-60] 认为指针和数组可以被统一地看待为 BYTE 类型的数组，并利用其附带的 STP 求解器对 BYTE 类型数组的内置支持来解决这一问题. SAGE 利用 Z3 求解器做了类似的处理，Z3[61] 求解器支持数组理论，且在 BYTE 类型数组之外也支持其他类型的数组. 需要特别指出的是，C 语言中的指针问题可以转化为数组的上下界问题，可以使用整数区间来进行表示，进而可以采用统一的算法加以处理.

(2) 对过程调用的处理. 静态符号执行工具的另一个问题是如何处理过程调用，DART 和 CUTE 工具对过程调用只做具体执行，其优点在于能够利用精确的运行时信息避免静态过程间分析导致的误报. 其问题是，一次执行结束后，这些信息就被丢弃，之后对同一函数的调用不能利用以前对函数已经获取的知识，造成了计算资源的浪费. Godefroid[62] 提出使用函数摘要来处理这一问题，其过程是：先通过静态结构分析获得程序调用图并使用图算法将其划分为多个子图，每个子图作为后续处理的单元；然后使用谓词公式以部分(不完全)程序规约的形式对函数的功能编码，在后续的符号执行过程中可以重用已经获取的函数摘要做为路径约束的一部分以避免重复计算. Joshi 等人[63] 提出对函数调用先抽象后求精的思路，程序在 Concolic 执行过程中遇到函数调用时只对被调用函数做具体执行，符号执行引擎并不进入被调用的函数，而是在函数返回后，以其输出作为新的符号变量加入到符号状态，并继续运行程序. 本次执行结束后，回到调用点，根据执行的结果搜索被调用函数的路径空间，利用本次执行已获得的信息只搜集所需部分的路径约束从而避免了枚举所有路径，达到节省计算的目的.

(3) 路径爆炸问题. 符号执行可以被视为程序执行树上各条分支的枚举过程，由于循环和调用等程序结构的存在，程序的执行树的分支可能无限. 处理循环的直接方法是设置最高阈值，即只对其做有限次数的搜索，这种方法类似于限界符号执行. 一种比较新颖的方法是路径约束抽象方法，通过对循环体先抽象后求精的方式来处理循环问题[64]. Boonstoppel 等人[65] 在 EXE 工具的基础上，根据程序的执行特点提出了路径剪裁算法，实验结果表明该方法可较好地缓解路径爆炸问题，由于符号执行的实现机制

不同，EXE 的升级版本 KLEE 并没有使用这个算法. 此外，实现 Concolic 执行的并行化，借助于大规模并行计算能力提高解题的规模和速度[66]，在多台计算机上并行处理程序的多条路径是另一种缓解路径爆炸问题的有效途径.

(4) 覆盖率和路径选择问题. 关注如何选取较少的路径就能够覆盖大多数代码. Concolic 执行需要不断选取下一条需要执行的程序路径，因此路径空间搜索和路径选择算法决定了整个系统最终能达到的覆盖率. 较好的路径选择方法如 KLEE 使用的基本块打分方法，能够尽量执行并覆盖更多基本块的代码. 另外，恰当的路径空间搜索算法能够及早发现程序错误，比如 SAGE 在 DART 的深度优先搜索算法上进行改进提出了按代搜索算法，作者声称[44-45]大多数发现的错误都是在四代变异之前发现的. Burnim 等人[46]提出了 CREST 系统，并对多种启发式算法做了综合比较，文献[46]是关于如何选取启发式算法搜索路径空间的较好的综述文章. 如何通过更多的工程实践观察到不同类型程序的执行特征和现象，结合并利用其他分析方法的思路和成果，从新的角度对问题建模并提出更先进的算法，是值得长期研究的问题.

(5) 性能问题. Concolic 执行通过程序插装实现符号执行，并使用 SMT 求解器对符号执行过程中获取的路径约束进行求解. 据统计，2000 行 C 代码插装后能达到 40 000 行[60]，这不仅意味着需要执行的代码量的增加，更意味着对求解器的调用次数也要随之增加，而求解器的性能决定了约束求解的效率，是基于新方法所开发工具的主要性能瓶颈. 解决这一问题的关键在于如何减少对求解器的依赖，目前主要技术途径有两种[43-44]，一是对路径约束进行优化，主要包括表达式重写、约束集合化简、隐含值具体化和无关约束消去等；二是使用缓存技术减少重复求解，主要包括符号表达式缓存、局部约束缓存和反例缓存等. 这两种手段的共同之处在于它们都利用了约束求解的以下 3 个特点：

① 若约束集合无解，则其超集也无解；

② 若约束集合有解，则此解也是其子集的解；

③ 约束集合的解很多时候也是其超集的解.

据称，几种优化方法一起使用能提高求解效率 30 倍左右[43].

7　工具和应用

2005 年，Bell 实验室和斯坦福大学分别独立提出混合符号执行的概念，并各自开发出原型工具 DART 和 EGT，其后 DART 的合作者之一 Sen 等人[40]对 DART 进行扩展开发了 CUTE 工具以处理 C 代码中的指针、别名和线程以及针对 JAVA 程序的 jCUTE 工具. 斯坦福的团队对 EGT[58]进行完善开发出 EXE，值得一提的是，斯坦福的开发小组同时开发了 STP 求解器以供 EXE 使用，获得了很大成功. 尽管这些工具在提出时还处于原型阶段，但是初步的试验已经表明这种方法十分有效，并引起了研究者和工业界的极大兴趣.

随后，DART 的主要作者 Godefroid 被微软聘请并帮助其设计了 SAGE 工具. SAGE 是基于二进制代码的重量级文件模糊测试工具，其作者声称[39]，在 Windows7 的开发过程中，通过模糊测试发现的全部程序错误中有三分之一是使用 SAGE 发现的. 从 2008 年开始，SAGE 被配置在微软的安全测试实验室的 100 多台机器上 7×24 不间断地持续运行，并针对几百个程序进行自动测试，这是迄今为止所知道的对 Z3 求解器的最大规模的使用.

除此之外，微软又将 Concolic 执行与单元测试领域结合设计了 PEX[67-68]工具，PEX 做为微软的 visual studio 的插件，可以在开发工作的同时进行单元测试工作，针对于一个代码单元(函数或类)自动生成测试用例，并指出可能的错误. SLAM 是微软开发的基于抽象解释和反例制导的求精方法的模型检验工具，用于检验驱动程序的可靠性，在多年来的实践中获得了很大成功，微软基于 Concolic 执行对 SLAM 进行改进，重新设计并推出了 YOGI[69-72]工具，相比于 SLAM、YOGI 的执行效率得到了极大提高.

斯坦福的团队基于对 EXE 的开发和使用所获取的实践经验，对 EXE 重新设计和实现，开发出新的自动化白盒测试工具 KLEE. KLEE 是个里程碑式的工具，极大地推动了这个领域的进展，其面世之后，基于 KLEE 开发的、针对不同领域和目的的工具竞相出现，比如漏洞利用自动生成工具 AGE[73]、性能分析工具 S2E[56]和逆向工程分析工具 RevNIC[74]等.

此外还存在许多其他工具，如 UIUC 提出的检查 SQL 注入的工具[75]、MIT 提出的检查 JavaScript 安全性的工具[76]、CMU 提出的检查僵尸网络的二进制动态分析工具 BitScope[54]等等.

国内，如南京大学[77]、北京大学[78-81]、国防科技大学[82]、北京航空航天大学[83]和北京邮电大学[84-85]等众多研究团队，在软件安全漏洞检测领域进行

了系统而深入的研究.特别值得一提的是,北京大学王铁磊等在该领域取得了重要研究进展,首次提出了一种绕过校验和机制的模糊测试方法,并开发了相应的工具 TaintScope,发现了 Adobe Acrobat、Google Picasa、Microsoft Paint 和 ImageMagick 等软件中的多个未知漏洞,得到领域内研究人员的广泛关注.

8 未来研究展望

基于轻量级动态符号执行的自动化白盒模糊测试方法的提出不足十年,仍然属于较新的方法,这个领域的发展方兴未艾,尽管取得了初步的成果,但依然有大量的问题亟待解决,未来研究的主要内容大概可以分为 3 个方面:

(1)从方法的角度来看,现有研究主要集中在如何将 Concolic 执行与模糊测试、模型检验、单元测试等方法结合并开发出新的工具,如 SAGE 基于 Concolic 执行对传统的文件模糊测试做出改进;Yogi是 Concolic 执行与模型检验结合的一次尝试;PEX 将 Concolic 执行引入单元测试.如何深刻认识问题和方法的本质,同时吸收并利用其他方法的优秀成果,相互启发,对传统方法进行改进,扩大新方法的作用范围并提高其使用效果,是进一步研究方向之一.

(2)从技术的角度来看,现有工具实现主要是针对有限的几种操作系统、语言和指令系统.近年来,物联网、工控系统和移动计算等领域的蓬勃发展对安全问题提出了严峻挑战,特别的,这些领域对安全漏洞检测的需求也日益增加.如何针对这些领域中应用的特性,将新方法有效引入到这些领域的安全漏洞检测工作中,具有重要研究意义.

(3)发现程序存在的问题是软件测试的主要目的,然而单纯通过测试通常只能发现浅层次程序错误.不仅如此,不同领域的程序中存在的问题类型也大不相同.例如,跨站脚本漏洞和 SQL 注入漏洞是 Web 应用领域的主要安全问题,与传统的溢出漏洞呈现出不同特征.如何对更多领域内所独有的程序漏洞类型进行细致地观察并刻画出其本质特征,是查找程序漏洞的关键之一.一些新工具已经使用本文所述新方法,对跨站脚本漏洞和 SQL 注入漏洞的检测做出了初步尝试,实验性结果揭示了新方法的有效性.因此,如何针对更多应用领域的程序,总结和整理其经常出现的漏洞类型和安全性约束,是开发新方法或改进上述方法适用范围的有效途径.

致 谢 北京航空航天大学计算机学院张玉平教授,为本文的修改提供了大量帮助,在此表示衷心感谢!

参 考 文 献

[1] Tassey G. The economic impacts of inadequate infrastructure for software testing. Gaithersburg: National Institute of Standards and Technology, Planning Report 02-3, 2002

[2] Chen Huo-Wang, Wang Ji, Dong Wei. High confidence software engineering technologies. Acta Electronica Sinica, 2003, 31(12A): 1933-1936(in Chinese)
(陈火旺, 王戟, 董威. 高可信软件工程技术. 电子学报, 2003, 31(12A): 1933-1936)

[3] Sipser M. Introduction to the Theory of Computation. Boston, USA: Thomson Course Technology, 2006

[4] Hoare C A R. An axiomatic approach to computer programming. Communications of the ACM, 1969, 12(10): 576-580

[5] Mei Hong, Wang Qian-Xiang, Zhang Lu, Wang Ji. Software analysis: A road map. Chinese Journal of Computers, 2009, 32(9): 1697-1710(in Chinese)
(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)

[6] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, USA, 2005: 213-223

[7] Ball T, Cook B, Levin V, Rajamani S K. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft//Proceedings of the Integrated Formal Methods. Canterbury, England, 2004: 1-20

[8] Korel B. A dynamic approach of test data generation//Proceedings of the IEEE Conference on Software Maintenance (ICSM). San Diego, USA, 1990: 311-317

[9] Korel B. Automated software test data generation. IEEE Transactions on Software Engineering, 1990, 16(8): 870-879

[10] Edvardsson J. A survey on automatic test data generation//Proceedings of the 2nd Conference on Computer Science and Engineering. Linköping, Sweden, 1999: 21-28

[11] Miller B P, et al. An empirical study of the reliability of UNIX utilities. Communications of the ACM, 1990, 33(12): 32-44

[12] Miller B P, Koski D, Lee C, et al. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Wisconsin: Computer Sciences Department, University of Wisconsin, Technical Report 1268, 1995

[13] Oehlert P. Violating assumptions with fuzzing. IEEE Security & Privacy, 2005, 3(2): 58-62

[14] Howard M. Inside the windows security push. IEEE Security & Privacy, 2003, 1(1): 57-61

[15] Whittaker J A. Stochastic software testing. Annals of Software Engineering, 1997, 4(1): 115-131

[16] Whittaker J A. What is software testing? And why is it so hard?. IEEE Software, 2000, 17(1): 70-79

[17] Giffin J, Jha S, Miller B. Efficient context-sensitive intrusion detection//Proceedings of the 11th Annual Network and Distributed Systems Security Symposium (NDSS). San Diego, USA, 2004: 255-269

[18] Godefroid P. Random testing for security: Blackbox vs. whitebox fuzzing//Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007). New York, USA, 2007: 1-1

[19] Watkins A, Berndt D, Aebischer K, et al. Breeding software test cases for complex systems//Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04). Hawaii, USA, 2004: 122-136

[20] Hedberg S R. Evolutionary computing: The rise of electronic breeding. IEEE Intelligent Systems, 2005, 20(6): 12-15

[21] Michael C C, McGraw G E, Schatz M A, Walton C C. Genetic algorithms for dynamic test data generation. RST Corporation, Sterling, VA: Technical Report RSTR-003-97-11, 1997

[22] Oehlert P. Violating assumptions with fuzzing. IEEE Security & Privacy, 2007, 3(2): 58-62

[23] Offutt A J, Hayes J H. A semantic model of program faults//Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). San Diego, USA, 1996: 195-200

[24] Nielson F, Nielson H R, Hankin C. Principles of Program Analysis. Heidelberg: Springer, 2005

[25] Muchnick S S. Advanced Compiler Design and Implementation. India: Morgan Kaufmann/Elsevier Science, 1997

[26] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints//Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). Los Angeles, USA, 1977: 238-252

[27] Li Meng-Jun, Li Zhou-Jun, Chen Huo-Wang. Program verification techniques based on the abstract interpretation theory. Journal of Software, 2008, 19(1): 17-26(in Chinese)
(李梦君，李舟军，陈火旺. 基于抽象解释理论的程序验证技术. 软件学报，2008，19(1)：17-26)

[28] Wagner R, Feiman J, MacDonald N, et al. Cool vendors in application security, 2010. Stanford, USA: Gartner Inc, Technical Report G00174954, 2010

[29] Floyd R. Assigning meaning to programs. Mathematical Aspects of Computer Science, 1967, 19(2): 19-32

[30] Winskel G. The Formal Semantics of Programming Languages. Cambridge, MA: MIT Press, 1993

[31] Lin Hui-Min, Zhang Wen-Hui. Model checking: Theories, techniques and applications. Acta Electronica Sinica, 2002, 30(12A): 1907-1912(in Chinese)
(林惠民，张文辉. 模型检测：理论、方法与应用. 电子学报，2002，30(12A)：1907-1912)

[32] Ball T, Rajamani S K. Automatically validating temporal safety properties of interfaces//Proceedings of the SPIN Workshop on Model Checking of Software (SPIN). Toronto, Canada, 2001: 103-122

[33] Nori A V, Rajamani S K, Tetali S, Thakur A V. The yogi project: Software property checking via static analysis and testing//Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS). York, UK, 2009: 178-181

[34] Boyer R S, Elspas B, Levitt K N. SELECT — A formal system for testing and debugging programs by symbolic execution//Proceedings of the International Conference of Reliable Software (ICRS). New York, USA, 1975: 234-245

[35] King J C. Symbolic execution and program testing. Communications of the ACM, 1976, 19(7): 385-394

[36] Howden W. Symbolic testing and the DISSECT symbolic evaluation system. IEEE Transactions on Software Engineering, 1977, 3(4): 266-278

[37] Pasareanu C S, Visser W. A survey of new trends in symbolic execution for software testing and analysis. International Journal on Software Tools for Technology Transfer, 2009, 11(4): 339-353

[38] Schwartz E J, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)//Proceedings of the IEEE Symposium on Security and Privacy (S&P). Berkeley/Oakland, USA, 2010: 317-331

[39] Cadar C, Godefroid P, Tillmann N, Visser W. Symbolic execution for software testing in practice preliminary assessment //Proceedings of the International Conference on Software Engineering (ICSE). Honolulu, USA, 2011: 1066-1071

[40] Sen K, Agha G. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools//Proceedings of the Computer Aided Verification (CAV). Seattle, USA, 2006: 419-423

[41] Sen K, Marinov D, Agha G. CUTE: A concolic unit testing engine for C//Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE). Lisbon, Portugal, 2005: 263-272

[42] Majumdar R, Sen K. Hybrid concolic testing//Proceedings of the International Conference on Software Engineering (ICSE). Minneapolis, USA, 2007: 416-426

[43] Cadar C, Dunbar D, Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs//Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08). San Diego, USA, 2008: 209-224

[44] Godefroid P, Levin M, Molnar D. Automated whitebox fuzz testing//Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS). San Diego, USA, 2008: 63-79

[45] Bounimova E, Godefroid P, Molnar D. Billions and billions of constraints: Whitebox fuzz testing in production//Proceedings of the International Conference on Software Engineering (ICSE). San Francisco, USA, 2013: 122-131

[46] Burnim J, Sen K. Heuristics for scalable dynamic test generation//Proceedings of the International Conference on Automated Software Engineering (ASE). L'Aquila, Italy, 2008: 443-446

[47] Nethercote N, Seward J. Valgrind: A framework for heavy-weight dynamic binary instrumentation//Proceedings of the ACM SIGPLAN Symposium on Programming Language Design & Implementation (PLDI). San Diego, USA, 2007: 89-100

[48] Hastings R, Joyce B. Purify: Fast detection of memory leaks and access errors//Proceedings of the Usenix Winter 1992 Technical Conference. San Francisco, USA, 1992: 125-138

[49] Godefroid P, Levin M, Molnar D. Active property checking//Proceedings of the International Conference on Embedded Software (EMSOFT). Atlanta, USA, 2008: 19-24

[50] Molnar D, Wagner D. Catchconv: Symbolic execution and run-time type inference for integer conversion errors. Electrical Engineering and Computer Sciences University of California at Berkeley, California: Technical Report No. UCB/EECS-2007-23, 2007

[51] Necula G C, McPeak S, Rahul S P, Weimer W. CIL: Intermediate language and tools for analysis and transformation of C programs//Proceedings of the Conference on Compiler Construction (CC). Grenoble, France, 2002: 213-228

[52] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis and transformation//Proceedings of the International Symposium on Code Generation and Optimization (CGO'04). San Jose, USA, 2004: 75-88

[53] Luk Chi-Keung, Cohn R, et al. Pin: Building customized program analysis tools with dynamic instrumentation//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Chicago, USA, 2005: 191-200

[54] Brumley D, Hartwig C, Liang Zhenkai, et al. Automatically identifying trigger-based behavior in malware. School of Computer Science, Carnegie Mellon University, Pennsylvania: Technical Report CMU-CS-07-133, 2007

[55] Fraser G, Arcuri A. Sound empirical evidence in software testing//Proceedings of the 34th International Conference on Software Engineering (ICSE). Zurich, Switzerland, 2012: 178-188

[56] Chipounov V, Kuznetsov V, Candea G. S2E: A platform for in-vivo multi-path analysis of software systems//Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). Newport Beach, USA, 2011: 265-278

[57] Prasad M, Biere A, Gupta A. A survey of recent advances in SAT-based formal verification. International Journal on Software Tools for Technology Transfer, 2005, 7(1): 156-173

[58] Cadar C, Engler D. Execution generated test cases: How to make systems code crash itself//Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05), San Francisco, USA, 2005: 2-23

[59] Cadar C, Ganesh V, Pawlowski P M, et al. EXE: Automatically generating inputs of death//Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006). Alexandria, USA, 2006: 322-335

[60] Yang J, Sar C, Twohey P, et al. Automatically generating malicious disks using symbolic execution//Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P). Berkeley, USA, 2006: 243-257

[61] de Moura L, Bjorner N. Z3: An efficient SMT solver//Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Budapest, Hungary, 2008: 337-340

[62] Godefroid P. Compositional dynamic test generation//Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). Nice, France, 2007: 47-54

[63] Joshi P, Sen K, Shlimovich M. Predictive testing: Amplifying the effectiveness of software testing (short paper)//Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE). Dubrovnik, Croatia, 2007: 561-564

[64] Strejček J. Abstracting path conditions//Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). Minneapolis, USA, 2012: 155-165

[65] Boonstoppel P, Cadar C, Engler D. RWset: Attacking path explosion in constraint-based test generation//Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Budapest, Hungary, 2008: 351-366

[66] Bucur S, Ureche V, Zamfir C, Candea G. Parallel symbolic execution for automated real-world software testing//Proceedings of the 6th ACM SIGOPS/EuroSys Conference on Computer Systems (EuroSys). Salzburg, Austria, 2011: 1-15

[67] Tillmann N, de Halleux J. Pex-white box test generation for .NET//Proceedings of the 4th Tests and Proofs, International Conference (TAP). Málaga, Spain, 2008: 134-153

[68] Tillmann N, Schulte W. Parameterized unit tests//Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE). Lisbon, Portugal, 2005: 253-262

[69] Gulavani B S, Henzinger T A, Kannan Y, et al. Synergy: A new algorithm for property checking//Proceedings of the

Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE). Portland, USA, 2006: 17-27

[70] Beckman N E, Nori A V, Rajamani S K, Simmons R J. Proofs from tests//Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). Seattle, USA, 2008: 3-14

[71] Godefroid P, Nori A V, Rajamani S K, Tetali S D. Compositional may-must program analysis: Unleashing the power of alternation//Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). Madrid, Spain, 2010: 43-56

[72] Nori A V, Rajamani S K. An empirical study of optimizations in YOGI//Proceedings of the International Conference on Software Engineering (ICSE). Cape Town, South Africa, 2010: 355-364

[73] Avgerinos T, Cha S K, Hao B L T, Brumley D. AEG: Automatic exploit generation//Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS). San Diego, USA, 2011

[74] Chipounov V, Candea G. Reverse engineering of binary device drivers with RevNIC//Proceedings of the 5th ACM SIGOPS/EuroSys Conference on Computer Systems (EuroSys). New York, USA, 2010: 167-180

[75] Fu Xiang, Qian Kai. SAFELI-SQL injection scanner using symbolic execution//Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008). Seattle, USA, 2008: 34-39

[76] Artzi S, Kiezun A, Dolby J, et al. Finding bugs in Web applications using dynamic test generation and explicit state model checking. Massachusetts Institute of Technology, Cambridge, MA, USA: Technical Report MIT-CSAIL-TR-2009-010, 2009

[77] Cui Zhan-Qi, Wang Lin-Zhang, Li Xuan-Dong. Target-direced concolic testing. Chinese Journal of Computers, 2011, 34(6): 953-964(in Chinese)
(崔展齐，王林章，李宣东. 一种目标制导的混合执行测试方法. 计算机学报，2011，34(6)：953-964)

[78] Wang T, et al. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection//Proceedings of the 31st IEEE Symposium on Security and Privacy (SP) 2010. Berkeley/Oakland, USA, 2010: 497-512

[79] Wang Tielei, Wei Tao, Lin Zhiqiang, Zou Wei. IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution//Proceedings of the 16th Network and Distributed System Security Symposium (NDSS'09). San Diego, USA, 2009: 1-14

[80] Wang Tielei, Wei Tao, Gu Guofei, Zou Wei. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. ACM Transactions on Information and System Security, 2011, 14(2): 15.1-15.28

[81] Wang Tie-Lei. Research on binary-executable-oriented software vulnerability detection [Ph. D. dissertation]. Peking University, Beijing, 2011(in Chinese)
(王铁磊. 面向二进制程序的漏洞挖掘关键技术研究[博士学位论文]. 北京大学，北京，2011)

[82] Li Gen. Dynamic test case generation based automatic defect mining for binaries[Ph. D. dissertation]. National University of Defense Technology, Changsha, 2010(in Chinese)
(李根. 基于动态测试用例生成的二进制软件缺陷自动发掘技术研究[博士学位论文]. 国防科技大学，长沙，2010)

[83] Hu Chao-Jian. Vulnerability detection based on program analysis [Ph. D. dissertation]. Beihang University, Beijing, 2012(in Chinese)
(忽朝俭. 基于程序分析的漏洞检测技术研究[博士学位论文]. 北京航空航天大学，北京，2012)

[84] Zhong Jin-Xin. Research on key technologies of malicious code binary program behavior analysis [Ph. D. dissertation]. Beijing University of Posts and Telecommunications, Beijing, 2012(in Chinese)
(钟金鑫. 恶意代码二进制程序行为分析关键技术研究[博士学位论文]. 北京邮电大学，北京，2012)

[85] Zhao Yun-Shan. Research on symbolic analysis based static defect detection technique [Ph. D. dissertation]. Beijing University of Posts and Telecommunications, Beijing, 2012 (in Chinese)
(赵云山. 基于符号分析的静态缺陷检测技术研究[博士学位论文]. 北京邮电大学，北京，2012)

LI Zhou-Jun, born in 1963, Ph. D., professor, Ph. D. supervisor. His main research interests include formal method and technology, information security, data mining.

ZHANG Jun-Xian, born in 1981, Ph. D. candidate. His main research interests include information security, program analysis.

LIAO Xiang-Ke, born in 1963, M. S., professor, Ph. D. supervisor. His main research interests include operating system and high performance computing.

MA Jin-Xin, born in 1986, Ph. D. candidate. His main research interests include information security, program analysis.

Background

For the purpose of detecting software vulnerability holes and improving software quality and reliability, various dynamic and static analysis methods and techniques are used to solve the problem; some of them are even automated and have been widely accepted in practice. Dynamic methods execute the program with different input data repeatedly, watch its behaviors at runtime, and try to capture the exceptional execution circumstances like runtime-crash, buffer-overflow, or out-of-memory etc. This sort of methods, e. g. fuzz-testing which is well known already, is resource-hungry, time-consuming, labor-intensive, and as it cannot enumerate all possible input data in usual, is prone to human omission and error. In contrast, static methods analyzing the source code of the program without running it practically, can reach much more higher analysis speed, however, as this kind of methods are commonly implemented using some form of abstraction, they can lead to big account of false negatives, thus enormous human efforts are spent to check whether a warning is a real bug. Despite massive investments in quality assurance, serious code defects and software vulnerability holes are routinely discovered after software has been released, and fixing them at so late a stage carries substantial cost. In order to overcome the limitations of classical solutions, a new method named concolic testing which compounds dynamic and static analysis techniques has been proposed in recent decade years, and is becoming a very hot research topic. The new method improves the old static symbolic execution and model checking to a new dynamic symbolic execution method; moreover, it brings the classical fuzz testing technique brand-new intelligence aspects. As a dynamic method, concolic testing has no false negatives but only false positives, meanwhile, comparing to the fuzz-testing, this new method can reach very high speed as it utilizes some new effective heuristic searching algorithms to cope with path-explosion problem that the number of paths through a program is roughly exponential in program size severely limits the extent to which large software can be thoroughly tested, thus it can reach high coverage rate. This paper reviews the basic concepts, key challenges and some classic solutions of software vulnerability detection problem; moreover, it introduces some new promising improvement in this research area, including lightweight dynamic symbolic execution, automatic white-box fuzz testing, their implementation technologies and their corresponding tools. Beyond this, it provides here an overview of some key challenges and new research trends in future research work.

This paper is supported by the National Natural Science Foundation of China under Grant Nos. 61170189, 61370126, 90718017, 60973105, Specialized Research Fund for the Doctoral Program of Higher Education of China under Grant No. 20111102130003.