分布式符号执行平台①

孙盼盼, 董 威

(国防科技大学 计算机学院, 长沙 410073)

摘 要: 在软件工程学中, 符号执行技术是一门高效的程序缺陷检测技术. 符号执行使用符号值作为程序的输入, 将程序的执行转变为相应符号表达式的操作, 通过系统地遍历程序的路径空间, 实现对程序行为的精确分析. 然而, 因受路径爆炸问题与约束求解问题的制约, 符号执行技术也面临着可扩展性差的问题. 为了在一定程度上缓解该问题, 本文实现了一个分布式符号执行平台, 该平台在调度算法的调度下将任务从主节点分发给多个工作节点, 进而实现了任务的并行执行, 降低了符号执行的时间开销.

关键词: 并行符号执行; 分布式系统; WEB 平台; 缺陷检测; KLEE

Distributed Symbolic Execution Platform

SUN Pan-Pan, DONG Wei

(School of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract: In software engineering, symbolic execution technology is an efficient program defect detection technology. Symbolic execution uses symbolic values as the inputs, which transforms the execution of the program into the corresponding symbolic expressions, and the precise analysis of the program behaviors is realized by systematacially traversing routing space. However, due to the restriction of the path explosion and constraint solving problems, symbolic execution technology has poor scalability. In order to mitigate the problem, this paper implemented a distributed symbolic execution platform which realized tasks parallelly execute and reduced the symbolic execution time overhead through a scheduling algorithm distributes tasks from master to slaves.

Key words: parallel symbolic execution; distributed system; WEB platform; defect detection; KLEE

1 引言

程序缺陷检测技术保证了程序质量、提高了程序可靠性, 长期以来都受到学术界与工业界的关注. 目前, 针对程序缺陷检测技术的研究主要包括静态源代码检测、黑盒模糊测试、污点分析及符号执行等. 符号执行技术(Symbolic Execution)是一种程序分析技术, 它通过分析程序得到让特定代码区域执行的输入. 使用符号执行技术分析程序时, 该程序会使用符号值而非具体值作为输入, 在达到目标代码时, 分析器可以得到相应的路径约束, 然后通过约束求解器得到触发目标代码的具体值[1]. 符号执行技术相对于其他程序分析技术而言具有程序执行覆盖率高、检测结果无误报及低漏报等优点, 因此逐渐成为近年来学术研究的热点, 同时也产生了一批优秀的符号执行工具, 如微软公司研发的 SAGE、PEX[2]等工具, 斯坦福大学于 2008 年发布的 KLEE[3], 以及 NASA 针对 Java 语言开发的 JPF[4]等, 都是符号执行领域具有代表性的成果.

符号执行过程大致可以分为以下几个模块, 如图 1 所示. 其原理为, 在不实际执行程序的前提下, 把源程序翻译为一种中间语言, 用符号值表示程序变量的值, 然后基于中间语言模拟程序执行来进行相关的分析[5]. 其中, 约束求解模块是符号执行系统的核心模块之一, 它的主要任务是根据程序执行过程中产生的路径约束计算出所对应的路径条件 PC(Path Condition),

① 基金项目:国家自然科学基金(91118007)

收稿时间:2015-08-09;收到修改稿时间:2015-09-17

将 PC 作为新的具体输入, 利用符号执行引擎对新输入值进行新一轮的分析. 符号执行技术正是通过使用这种迭代产生输入的做法, 在理论上达到了遍历分析所有可行路径的效果.

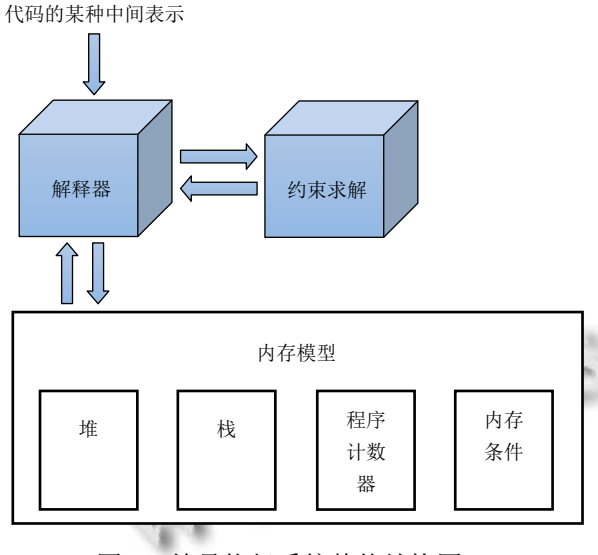

图 1 符号执行系统整体结构图

符号执行技术的发展也面临着诸多问题, 其中, 可扩展性是符号执行技术面临的主要问题之一. 符号执行的可扩展性问题主要是由路径爆炸问题与约束求解问题引起的. 路径爆炸问题是动态符号执行所面临的主要问题, 已成为符号执行实际应用于大中型软件系统的瓶颈, 原因在于程序所包含的路径数在最糟糕的情况下随程序中分支数的增长而呈指数级增长, 因此, 符号执行虽然在理论上可以遍历程序中每一条可达路径并生成测试用例, 但在实际执行过程中这一目标往往难以达到. 约束求解问题与软件规模及软件结构的复杂性紧密相关. 实际上, 随着软件规模的增大以及软件结构复杂性的提升, 约束表达式将变得愈加复杂, 使得求解器的求解变得极为困难. 另一方面, 实际程序也可能会包含各种各样的操作, 如非线性运算、移位等. 有研究指出, 含有复杂非线性运算的逻辑系统是不可判定的, 当约束求解器不能判定某一约束条件是否可满足时, 其相应的路径也就无法进行符号执行[6]. 此外, 现在的大多数求解器也都存在着对浮点操作支持不足的问题. 针对路径爆炸问题, 研究人员提出了使用状态合并、抽象、组合化思想、基于启发式策略的搜索、目标引导的搜索、并行化等技术来提高符号执行的效率. 这些技术从不同的角度出发, 试图减少符号执行需要探索的路径数或者减少探索路径空间所需的时间和空间资源消耗. 在约束求解方面的研究, 主要包括约束查询的优化技术以及特定应用领域的约束求解等[7].

并行化方法即所谓的并行符号执行[8]是符号执行技术发展的另一个趋势, 很大一部分原因在于近年来多核技术与“云计算”技术的快速发展所引起的计算方式的变革[9]. 常规上, 并行化方法根据相应的算法将程序的路径空间进行划分, 使用不同的计算单元来同时对路径空间的不同部分进行探索, 同时又考虑了多个计算单元之间的交互和负载均衡的问题[7]. 实际上, 并行化方法是一种典型的以空间和计算资源换时间的例子. 基于该理念已有研究者开发出了相应的工具, 如 Cloud9[10]与 MergePoint[11], 均取得了很好的效果.

为了在一定程度上缓解符号执行技术的可扩展性问题, 本研究基于开源的符号执行工具 KLEE 实现了一个 B/S 结构的分布式符号执行平台. 该平台通过使用多台虚拟机并行执行任务的方式, 大幅度降低了多任务情况下符号执行的总体求解时间开销. 本研究选用了开源的 KLEE 作为符号执行工具, 但相关工具的选择并不局限于 KLEE, 实际上, 将其他符号执行工具集成到该平台中也非常容易.

2 分布式符号执行平的设计与实现

2.1 分布式符号执行平台架构与功能

如图 2 所示, 本研究实现的分布式符号执行平台属于明显的 B/S 结构与 Master/Slave 架构相结合的产物, 主要分为客户端模块、Master 模块及检测模块. 该平台通过设置一个待检测队列, 一个 Slave 集合可用状态队列, 一个待解析检测结果队列, 在调度算法的调度下将任务从 Master 节点分发给 Slave 节点, 执行结束后, 再将产生的结果从 Slave 节点回传给 Master 节点, 进而实现了任务的并行执行. 具体说来, 系统中 Master 节点负责对整个系统的管理, 包括任务的分发与检测结果的回收, 对检测结果的进一步处理以及对 Slave 集合的管理等, Slave 则只负责处理从 Master 接收到的任务并将检测结果回传给 Master.

从平台的具体实现上来说, 其技术难点在于如何实现相应的算法分别对待检测任务队列、Slave 集合队列及待处理检测结果队列的管理, 如何提高平台的健壮性及防止非授权操作等. 本研究通过设置相应的定

时调度器、相应的监听器及相应的授权拦截器的策略来应对上述问题. 概括起来, 本研究共设置了三个定时调度器分别对应管理 Slave 集合队列、待检测任务队列及待处理检测结果队列, 设置了一个监听器用以提高平台的健壮性, 设置了一个授权拦截器以防止非授权操作, 下面对其一一进行介绍.

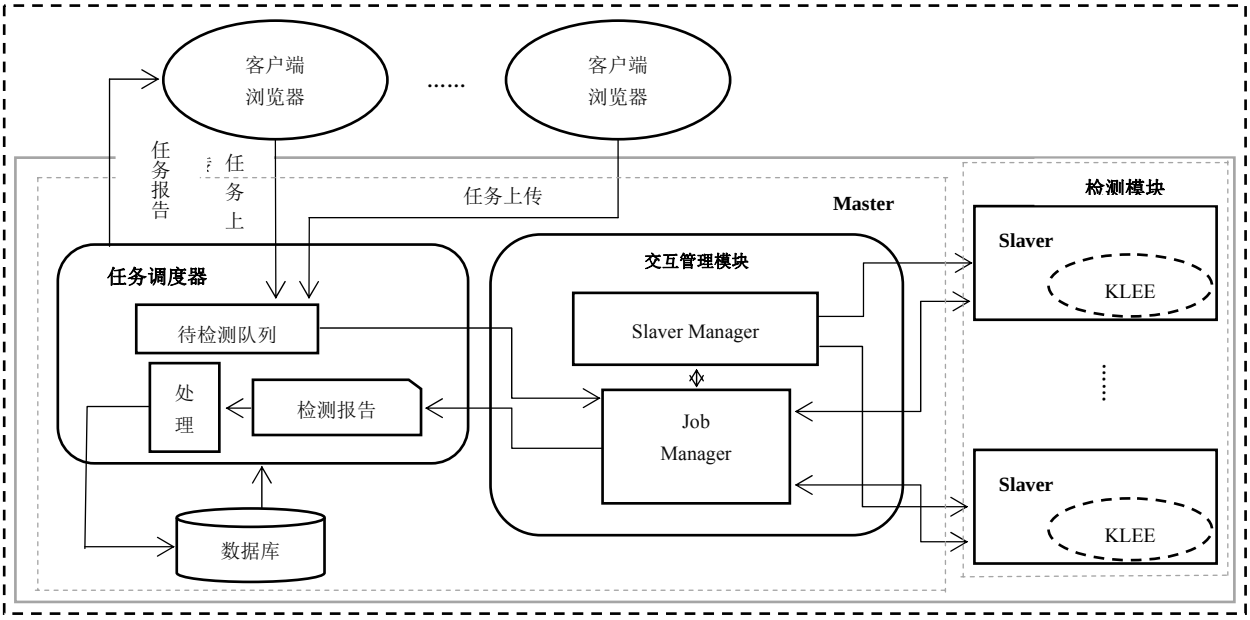

图 2 分布式符号执行平台系统架构图

(1)Slave 集合队列定时调度器用于周期性的获取每个 Slave 的“心跳”. 该调度器按照一定的时间间隔对整个 Slave 集合 IP 列表进行扫描, 根据 IP 地址向每个 Slave 发送一条消息, 按照相应的返回值更新该 Slave 的状态.

(2)待检测任务队列定时调度器的功能是调度分发任务. 该调度器也是按照一定的时间间隔扫描整个任务列表, 并将状态为“待检测”的任务分发给一个空闲的 Slave 执行.

(3)对于初步检测结果, 本研究所采取的策略是不立即进行解析而是先将其放入对应的队列, 等相应的调度器触发后统一进行处理. 因此, 待处理检测结果队列定时调度器的作用是定时触发相应的算法对检测结果进行处理.

(4)为了提高平台的健壮性, 本研究在系统启动时配置了一个监听器, 该监听器用于将处于异常状态的任务或 Slave 重置为正常状态, 比如系统重启时发现某个任务的状态是“正在执行”状态, 则应将其置为“检测失败”状态.

出于安全性的考虑, 本研究还设置了一个授权拦截器防止用户直接通过 URL 访问平台中的页面.

2.2 分布式符号执行平台实现

在平台的具体实现过程中, 本研究结合使用了多种技术. 其中, 使用了开源的符号执行工具 KLEE 作为符号执行器, 使用了轻量级的 struts + spring + hibernate 作为 Web 框架, 使用了 SSH(Secure Shell)+SCP(Security Copy)技术负责 Master 与 Slave 之间的通信, 使用了开源的作业调度框架 Quartz 负责任务的调度, 此外, 还使用到了 tcping、putty 等软件, 整个系统使用 JSP 与 Java 语言完成. 下面将分模块对其实现进行介绍.

客户端浏览器. 用户通过其完成注册、任务提交、任务修改及任务查询等功能. 本研究所开发的平台支持两种方式的任务提交, 分别为基于源代码的单任务提交与基于字节码的批量任务提交. 本研究中符号执行环境的初始化工作也是在这部分完成的, 初始化界面如图 3 所示.

如图 3 所示, 本研究在任务创建时可以指定符号执行的搜索策略、建模使用到的外部函数库、最大执行时间以及符号化的参数个数及长度等.

Master 模块又分为任务调度器模块及交互管理两大子模块. 任务调度器模块负责界面展示及收集用户

请求，并将用户提交的任务放入待检测队列，将返回的检测结果进一步处理后存入数据库. 交互管理模块负责对 Slave 集群与 Job 队列的管理. 它通过 Slave Manager 子模块负责对 Slave 的添加、删除及状态的查询，通过 Job Manager 子模块负责从待检测队列取出任务并根据从 Slave Manager 获得的 Slave 集合的忙闲状态将任务分发下去. 该模块使用到了三个核心调度器，分别为 Slave 集合队列定时调度器、待检测任务队列定时调度器以及待处理检测结果队列定时调度器. 其中，Slave 集合队列定时调度器对应平台的 Slave Manager 模块，它使用 tcping 工具定时的对 Slave 节点中部署的 SSH 服务端口进行探测，完成 Master 节点对 Slave 节点的监控管理；待检测任务队列定时调度器与待处理检测结果队列定时调度器则均使用 Quartz 框架的作业调度功能来完成任务的定时分发(通过 putty 工具的 pscp 功能)与检测结果的定时处理，它们分别对应平台中的 Job Manager 模块与检测报告处理模块. 三个核心调度器的处理过程如图 4 所示.

图 3　符号执行环境初始化界面图

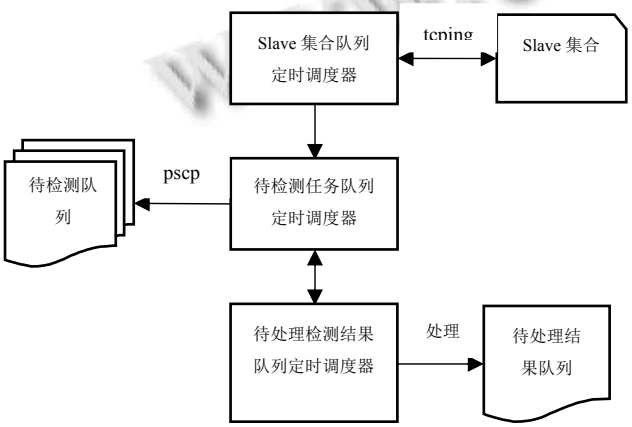

图 4　核心调度器处理过程图

检测模块. 该模块由多个 Slave 组成的集合，其中每个 Slave 均配置了 KLEE 执行环境，负责将接收到的任务进行检测并将检测结果返回. 这里用到了 putty 工具的 plink 功能与 pscp 功能.

3　实验与分析

本研究中 KLEE 的执行环境部署在了 Ubuntu 14.04 64 位的系统中，作为一个 Slave 节点. 基于此，本研究以虚拟机的形式在开源的云计算平台 CloudStack 上创建了 120 个 Slave 节点，每个节点均分配了 2GHZ 的处理器资源及 2GB 的 RAM 资源，Web 服务器即 Master 节点则又单独为其在 CloudStack 创建了一个节点，该节点具有 16 核共 16GHZ 的处理器资源及 32GB 的 RAM 资源.

3.1　分布式符号执行平台分析效率评价

为了对平台所能带来的分析效率的提升进行评价，本研究对比分析了四组实验. 四组实验分别选取 5、10、20、30 个程序作为四批任务分别使用本研究实现的平台与单个节点进行处理，结束后统计执行时间. 对于本研究的平台，由于各批任务中的程序是并行执行的，因此我们选取了每批任务中执行时间最长的程序作为基准，取其执行时间为相应组的执行时间 . 对于单节点，我们则选取了每批任务的总执行时间作为其执行时间. 同时，作为对比分析之用，我们还定义了理想情况下平台的执行时间，该时间是以单节点执行每批任务的总时间为基础的，将其定义为该批任务中每个程序的平均执行时间，即用执行总时间除以程序个数. 使用分布式符号执行平台并行处理各批任务与使用单节点串行处理各批任务所用时间的对照如图 5 所示.

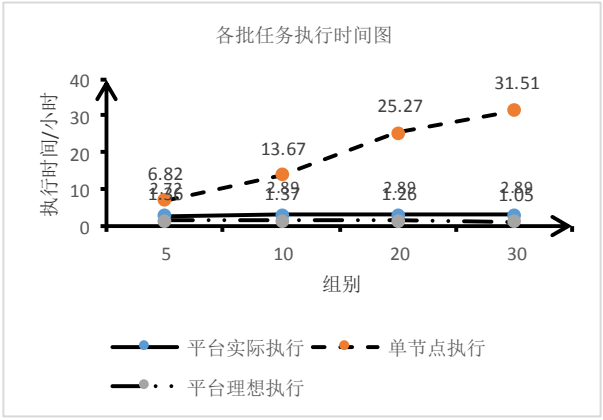

图 5　各批任务执行时间图

如图 5 所示, 横坐标对应的是实验组别, 纵坐标对应的是执行时间, 实线对应的是使用本研究的平台并行处理各批任务所用时间, 虚线对应的是使用单节点顺序处理各批任务所用时间, 间断的虚线对应的是平台理想情况下的执行时间. 可以看出我们的平台可以在很大程度上提升程序的分析效率, 但与理想情况还有一定的差距, 主要原因是, 一方面我们选取了各批任务中执行时间最长的程序作为基准, 将其执行时间作为平台的执行时间, 另一方面, 使用平台进行任务处理时与单节点或理想情况相比, 还要考虑其他方面的时间开销. 因此, 图 5 只能在一定程度上反映出平台分析效率的提升程度. 为了更全面的对平台所能带来的分析效率的提升进行评价, 本研究在下文讨论了理想情况下平台分析效率提升的上限值.

设节点数为 N, 任务总数为 n, 每个任务的平均分析时间为 t, 使用本研究开发的分布式平台进行分析所花费的总时间为 T_{ds}, 使用单个节点依次对测试对象进行分析所花费的总时间为 T_s, 则

$$T_{ds} = \left\lceil \frac{n}{N} \right\rceil t \tag{1}$$

相应的

$$T_s = nt \tag{2}$$

将 R 定义为使用本研究的平台相对于使用单个节点进行分析所带来的效率提升的倍数, 则由(1)与(2)得

$$R = \frac{n}{\left\lceil \frac{n}{N} \right\rceil} \tag{3}$$

公式(3)反映出使用本研究的平台可以大幅度提升程序的分析效率, 当然, 这种分析效率的提升是以增加硬件的花费为代价换来的, 是一种以空间换时间的策略. 显然, 实际情况下公式(3)是很难成立的, 其中一部分原因我们在前面已经讨论过, 还有一部分原因是在实际测试中可能会出现因参数设置不当如执行时间过长或其他原因导致某些任务执行失败的情况. 针对这种情况, 有时我们就需要将失败的任务提取出来重新打包成一批新的任务调整参数后再次进行检测, 这也会影响到平台的分析效率.

3.2 分布式符号执行平台案例分析

本研究考虑到平台提供了两种任务提交方式的特点, 分别从基于源代码的单任务提交方式与基于 LLVM 字节码的批量任务提交方式两部分进行了实验方案的设计, 从平台的实用性、易用性、界面友好性及实际发现软件缺陷的能力等方面对其进行评价.

3.2.1 基于源代码的单任务提交方式案例分析

为了对平台的实用性、易用性及界面友好性做出评价, 本文选取了 coreutils-6.10 中的 paste 程序作为实验对象进行分析, 该程序的 C 代码行数共有 496 行. 如前所述, 本研究中 KLEE 执行环境的初始化是在任务提交时进行的, 这一过程通过平台的可视化界面来做是一件很方便的事情, 这特别适用于对 KLEE 不是很熟悉的研究者, 当然, 对于专业人员来说这样做也可以节省时间, 从这点来说, 本研究所开发的平台具有较好的实用性与易用性.

将 KLEE 的最大执行时间设置为 15 分钟, 符号化参数个数设置为 0 到 3 个且最大长度为 10 的情况下进行分析, 执行结束后平台共报出了 10 个与内存相关的错误, 检测结果页面如图 6 所示.

图 6 检测结果页面图

图 6 所示的是任务执行结束后检测结果的详细情况, 其中, 上半部分显示出了任务执行结束后的统计信息, 包括指令数、指令覆盖率、分支覆盖率、代码覆盖率及执行时间等信息, 可以看到, 该任务的代码覆盖率达到了 90.66%. 下半部分则对应的是错误的具体情况, 包括错误类型、错误级别、错误内容、错误所在文件以及错误行号. 本研究所开发的平台经过对检测结果的解析与封装, 以相当友好的可视化界面将检测结果呈现了出来, 这为我们后续更大规模的分析程序提供了方便.

3.2.2 基于字节码的批量任务提交方式案例分析

同样的, 为了对平台实际发现软件缺陷的能力做出评价, 本研究选取了 40 个测试包共计 575 个字节码文件超过 2000K 的 C 代码进行了分析, 其中, 每个字节码文件作为一个任务, 共有 575 个任务.

将初始分析参数设置为最大执行时间 1 小时, 符号化参数个数从 0 到 3 且最大长度为 10 的情况下, 检测结果如表 1 所示.

表 1 测试包及发现缺陷数

名称	行数	缺陷数
acpi-1.5	902	0
acpid-2.0.23	5651	2
bc-1.06.95	14966	0
bsdmainutils-8.2.3	8906	2
bzip2-1.0.6	7326	0
ConsoleKit-0.4.1	17242	0
coreutils-6.10	132052	29
cpufrequtils-007	4572	0
dash-0.5.5.1	17479	0
desktop-file-utils-0.15	3749	0
diffutils-3.0	48457	0
ed-1.4	2877	0
fakeroot_1.18.4	4918	0
grep-2.6.3	38497	0
gutenprint-5.2.10	102134	0
haproxy_1.5.3	75922	0
hostname_3.04	550	1
liblockfile-1.08	1049	0
make-dfsg-3.81	33622	0
makejvf-1.1a	1009	1
memcached-1.4.24	12990	3
min12xxw-0.0.9	2706	0
module-init-tools-3.12	7637	3
ncurses-5.7	96894	1
netpbm-10.35.96	237926	0
pciutils-3.1.7	8993	0
pnm2ppa-1.13	50891	1
powermgmt-base_1.31	103	0
procps-3.2.8	18034	2
psmisc-22.11	4215	0
ruby-1.9.3-p484	651112	0
sed-4.2.1	27153	0
tcl8.5.8	226499	0
texinfo-5.2	48079	0
time-1.7	2011	0
traceroute-2.0.15	4527	0
util-linux-ng-2.17-rc1	81981	11
whois-5.0.10	1981	0
xdg-user-dirs-0.13	1392	0
xfonts-utils_7.5+2	6205	0
总计	**2013209**	**56**

如上表所示, 通过对超过 2000K 的 C 代码进行检测, 本研究共发现了 56 个错误.

4 结语

本研究在开源的符号执行工具 KLEE 的基础上实现了一个 B/S 结构的分布式符号执行平台. 该平台通过设置一个待检测队列, 在调度算法的调度下将任务从主节点分发给工作节点, 进而实现了任务的并行执行, 降低了符号执行的时间开销. 基于该平台, 首先, 我们通过四组对比试验对平台的分析效率进行了评价, 然后, 我们又对 40 个程序包超过 2000K 行的 C 代码进行了分析, 共检测出了 56 个程序缺陷, 通过以空间换时间的方式, 我们的分布式平台大幅度的提高了程序的分析效率.

参考文献

1 Schwartz EJ, 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). 2010 IEEE Symposium on Security and Privacy (SP). IEEE. 2010.

2 Tillmann N, De Halleux J. Pex-white box test generation for .NET. Tests and Proofs, 2008: 134–153.

3 Cadar C, Dunbar D, Engler DR. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. Symp. on Operating Systems Design and Implementation, 2008.

4 董齐兴.基于动态符号执行的测试用例生成技术研究[硕士学位论文].合肥:中国科技大学,2014.

5 娄坚波.面向宿主的嵌入式软件符号执行技术研究与实现[硕士学位论文].南京:南京航空航天大学,2011.

6 Enderton H. A Mathematical Introduction to Logic, second edition. Academic Press, 2001.

7 张羽丰.符号执行可扩展性及可行性关键技术研究[博士学位论文].长沙:国防科学技术大学,2013.

8 Staats M, Pasareanu C. Parallel symbolic execution for structural test generation. ISSTA'10. Trento, Italy. July 12-16, 2010.

9 Armbrust M, Fox A, Griffith R, et al. A view of cloud computing. Communications of the ACM, 2010, 53(4): 50−58.

10 Bucur S, Ureche V, Zamfir C, Candea G. Parallel symbolic execution for automated real-world software testing. Reprinted from EuroSys'11, Proc. of the 6th ACM SIGOPS/EuroSys Conference on Computer Systems. Salzburg, Austria. April 10-13, 2011. 1–15.

11 Avgerinos T, Rebert A, Cha SK, Brumley D. Enhancing symbolic execution with veritesting. ICSE'14. Hyderabad, India. May 31-June 7, 2014.