计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(9):67−76 [doi: 10.15888/j.cnki.csa.009230]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于解析树的 Java Web 灰盒模糊测试①

王　鹃[1,2], 张志杰[1,2], 杨鸿远[1,2]

[1](武汉大学 国家网络安全学院, 武汉 430072)
[2](空天信息安全与可信计算教育部重点实验室, 武汉 430072)
通信作者: 王　鹃, E-mail: jwang@whu.edu.cn

摘　要: 由于 Java Web 应用业务场景复杂, 且对输入数据的结构有效性要求较高, 现有的测试方法和工具在测试 Java Web 时存在测试用例的有效率较低的问题. 为了解决上述问题, 本文提出了基于解析树的 Java Web 应用灰盒模糊测试方法. 首先为 Java Web 应用程序的输入数据包进行语法建模创建解析树, 区分分隔符和数据块, 并为解析树中每一个叶子结点挂接一个种子池, 隔离测试用例的单个数据块, 通过数据包拼接生成符合 Java Web 应用业务格式的输入, 从而提高测试用例的有效率; 为了保留高质量的数据块, 在测试期间根据测试程序的执行反馈信息, 为每个数据块种子单独赋予权值; 为了突破深度路径, 会在相应种子池中基于条件概率学习提取数据块种子特征. 本文实现了基于解析树的 Java Web 应用灰盒模糊测试系统 PTreeFuzz, 测试结果表明, 该系统相较于现有工具取得了更好的测试准确率.

关键词: 漏洞挖掘; 模糊测试; Java Web; 解析树

引用格式: 王鹃,张志杰,杨鸿远.基于解析树的 Java Web 灰盒模糊测试.计算机系统应用,2023,32(9):67–76. http://www.c-s-a.org.cn/1003-3254/9230.html

Gray-box Fuzzing for Java Web with Parse Tree

WANG Juan[1,2], ZHANG Zhi-Jie[1,2], YANG Hong-Yuan[1,2]

[1](School of Cyber Science and Engineering, Wuhan University, Wuhan 430072 , China)
[2](Key Laboratory of Aerospace Information Security and Trusted Computing of Ministry of Education, Wuhan 430072, China)

Abstract: Due to the complexity of the Java Web application business scenario and the high requirement for the structural validity of the input data, the existing test methods and tools have the problems of low efficiency of test cases when testing Java Web. To solve the above problems, this study presents a gray-box fuzzing method for Java Web applications based on parse trees. First, the study models the syntax of the input packets of Java Web applications, creates a parse tree, distinguishes between delimiters and data blocks, and hooks up a seed pool for each leaf node in the parse tree. In addition, the study isolates the single data block of the test case and generates the input in line with the Java Web application business format by packet splicing, so as to improve the efficiency of test cases. In addition, in order to retain high-quality data blocks, each data block seed is assigned a separate weight during the test according to the execution feedback information of the test program. In order to break through the deep path, the seed features of data blocks are extracted based on conditional probability learning in the corresponding seed pool. This study implements a gray-box fuzzing test system for Java Web applications based on parse trees, namely PTreeFuzz, and the test results show that the system achieves better test accuracy compared with existing tools.

Key words: vulnerability mining; fuzzing; Java Web; parse tree

① 基金项目: 国家自然科学基金 (61872430)
收稿时间: 2023-02-21; 修改时间: 2023-03-22; 采用时间: 2023-04-07; csa 在线出版时间: 2023-07-17
CNKI 网络首发时间: 2023-07-18

Java Web[1–3] 是用 Java 技术来解决相关 Web 互联网领域的技术栈, 得益于相关开发框架的成熟, 及其自身的稳定性和开放性, Java Web 已经成为目前比较主流的 Web 应用开发技术, 并被广泛运用于电子商务, 金融证券, 医疗等各个行业的大型应用系统中. 但与此同时, Java Web 应用的安全问题也日渐突出, 这些潜在的安全威胁给企业和个人造成了信息窃取及隐私泄露等问题[4].

针对 Web 应用程序的漏洞检测一直是学术界和工业界的重要课题. 现有的 Web 应用安全检测方法主要有静态符号执行和污点分析等静态漏洞检测方法以及模糊测试、动态符号执行等动态漏洞检测方法. 传统的模糊测试工具在 Java Web 场景下存在一些问题, 基于变异的模糊测试不能满足 Java Web 复杂的输入数据格式, 而基于生成的模糊测试因为缺乏代码覆盖率信息作为引导程序测试的重要指标[5], 导致它们无法有效地提升对 Java Web 应用程序进行安全检测的效果, 测试容易陷入“停滞”. 目前针对 Java Web 的模糊测试大多是通过分析 Java Web 系统的函数调用图以灰盒反馈形式优化模糊测试, 但一方面受 Java Web 应用输入数据严格的结构要求影响, 模糊测试种子有效率不高, 另一方面模糊测试过程将种子视为整体导致低质量的数据块影响了整体的模糊测试性能.

针对上述问题, 本文设计和实现了基于解析树的 Java Web 灰盒模糊测试系统 PTreeFuzz, 主要的贡献如下.

(1) 针对传统基于变异的模糊测试工具应用于输入数据具有高结构性的 Java Web 场景难以有较好效果的问题, 提出一种高效的基于语法解析树的灰盒模糊测试方法, 为输入数据进行语法建模并建立解析树, 区分输入数据的分隔符和数据块, 通过数据包拼接生成测试用例, 优化测试用例的语法结构.

(2) 通过设置解析树中的多个种子池隔离每个数据块, 减少低质量数据块对模糊测试的整体负面影响, 通过灰盒反馈尽量保留高质量数据块的语义信息. 同时, 针对现有的模糊测试工具在运行一段时间之后, 很难再发现新的执行路径或者漏洞的问题, 本文为解析树模型设计了一套轻量级的基于条件概率的模糊测试指导方案, 通过提取学习高质量数据块的语义特征对深度路径进行重点“突破”.

实验表明系统在触发程序异常和测试覆盖率方面均优于现有的基于变异和基于生成的模糊测试工具.

1 相关工作

模糊测试是一种有效的动态漏洞挖掘方法, 按种子的生成策略可分为基于变异的模糊测试和基于生成的模糊测试[6]. 本文按应用场景将目前模糊测试工作分类为针对 Java Web 应用的模糊测试以及非 Java Web 场景下的模糊测试, 分别进行介绍.

目前针对 Java Web 应用进行模糊测试的优化工作主要是分析 Web 应用系统或者漏洞特征, 通过灰盒反馈信息优化测试用例生成. Dos Santos 等人[7] 设计了浏览器插件 Selenium, 驱动浏览器实现 Java Web 的模糊测试, 需要录制用户操作并回放生成测试用例, 因此随机性不足, 难以有效发现系统异常和漏洞. 张羿辰[8] 使用白盒模糊测试技术对 Web 应用进行调用关系的代码属性图分析指导模糊测试, 但其对 Web 场景下种子的变异策略设计不足, 导致产生大量的无效测试用例. 何杰等人[9] 结合 Web 前端请求和后端程序分析优化种子生成, 用来对命令注入漏洞进行挖掘. 倪萍等人[10] 通过爬取分析网页链接, 找到潜在的注入点, 使用模糊测试生成 Payload, 从而进行反射型 XSS 漏洞检测. Van-Thuan Pham[11] 设计了灰盒模糊测试工具 AFLNET, 将客户端与服务端的通信流量作为初始种子, 并通过代码覆盖率优化种子. WebFuzz[12] 与 Wfuzz[13] 是对 Web 协议的模糊测试工具, 它们都要求使用者具有完备的网络知识, 熟悉 HTTP 协议分析网络报文, 因此不能满足自动化模糊测试 Java Web 应用程序的需求. 基于生成的模糊测试工具 Peach[14] 是一款通过建模生成满足特定数据结构的 Web 模糊测试工具, 但其没有考虑到代码覆盖率等反馈信息, 执行路径以及代码覆盖率受到限制, 因此难以发现有价值的漏洞.

其他非 Java Web 场景下的模糊测试的工具在应用于 Web 场景时同样面临问题. AFL[15] 等基于变异的模糊测试工具利用代码覆盖率来引导测试, 在进行实际测试的时候, 难以生成结构性强的有效输入. 傅玉等人[16] 通过统计路径边的执行次数筛选测试低频边的种子, 没有考虑到 Web 应用复杂的结构性. 针对 Java 程序进行模糊测试的工作中, Kelinci[17] 将 AFL 应用于 Java 字节码, 但由于加了一层中间代理且测试用例源于 AFL, 模糊测试的执行速度较慢、准确率低、效率不高. Padhye 等人[18] 设计开发的 JQF 也是一款针对 Java

的模糊测试工具, 它拓展了 Java 的单元测试, 利用 ASM 获取代码覆盖率. 与 Kelinci 相似的是, JQF 的测试用例也来自于 AFL, 因此产生的大量无效数据使得该工具不适用于大型的 Java Web 应用程序.

综上, Java Web 模糊测试工具大多专注于分析系统或者漏洞特征优化模糊测试整体效果, 但缺乏在 Java Web 应用场景下提升种子语义和语法有效性的相关研究, 针对网络协议的模糊测试难以自动化地对 Java Web 应用进行测试. 普通基于变异的模糊测试针对 Java Web 常见的 XML, JSON[19] 等这一类结构性较强的传输数据格式, 难以产生符合规定的输入, 难以保证输入测试用例的有效率. 一些基于生成的模糊测试工具如 Peach 等由于缺乏代码覆盖率等反馈信息来指导模糊测试, 其测试效率也非常低下.

2 系统设计

PTreeFuzz 系统架构如图 1 所示. 系统核心分为两部分, Fuzzer 和位图监控模块. 客户端主要部署的是系统的模糊测试器 Fuzzer, 负责通过监控模块的反馈指导测试数据的变异, 它的功能包括: 生成解析树发送测试数据包, 计算权重, 对种子进行变异以及将种子组装成报文. 服务端部署被测试对象和系统的位图监控模块, 位图监控模块包括位图监控以及记录条件语句的链表, 指导 Fuzzer 中解析树单个叶子节点种子池的筛选, Fuzzer 根据位图监控模块反馈信息修正种子结构体. 少数种子可以执行到深度分支路径, 在模糊测试面临路径“阻塞”问题时, 需要重点关注深度分支路径, 通过分析条件语句链表, 依据条件概率提取执行深度分支种子的特征, 优化模糊测试整体效率.

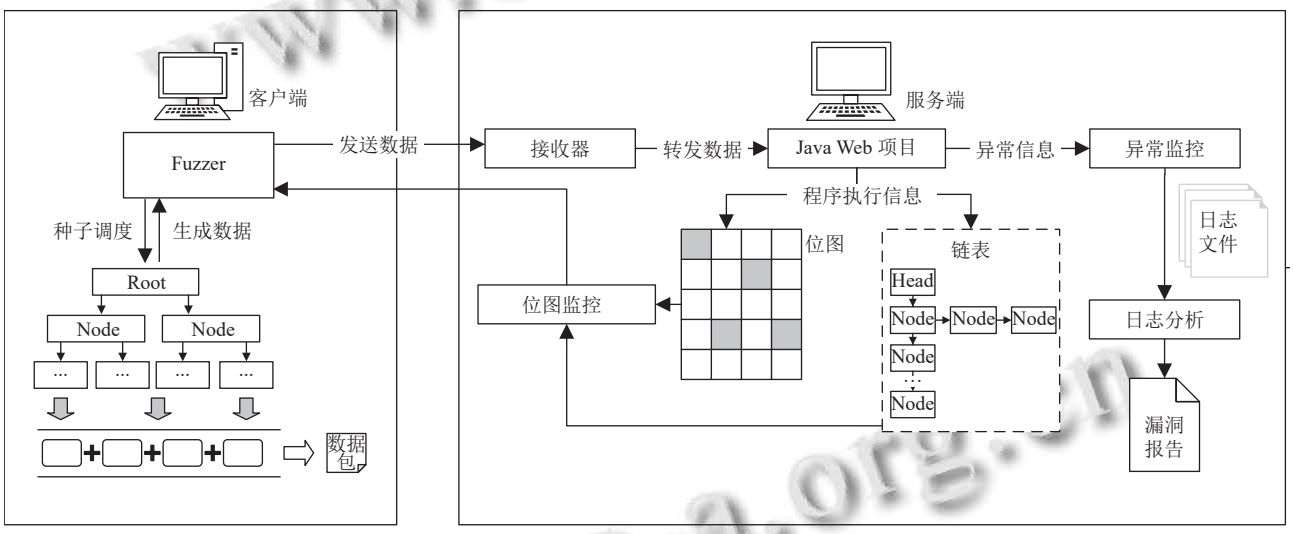

图 1 PTreeFuzz 系统整体架构

2.1 基于语法解析树的 Java Web 测试方法

Java Web 现有的传输数据格式主要是两种: XML 与 JSON, 以 JSON 格式为例, 进行具体的介绍: JSON 数据的书写格式是键/值对; JSON 值可以是: 字符串、数字 (整数或浮点数)、逻辑值等; 它有两种结构, 对象和数组; 通过这两种结构可以表示各种复杂的数据. 从数据包中字段的角度来看, 包含了多个键值对, 其对应的数据类型也不尽相同, 在目标 Java Web 程序里面对这些不同的数据块大部分采用不同的处理方式, 触发不同的执行路径.

如图 2 中给出了一个程序对数据包处理的控制流图的例子. 不同类型, 不同内容的数据部分 (在图 2 中用 $\alpha, \beta, \gamma, \delta$ 等表示) 会产生不同的执行路径 (用不同的英文字母以及不同的颜色表示), 这些执行路径也可能包含具有公共功能的共享代码块如 d、e. Java Web 程序在对报文进行处理的时候, 针对不同的数据部分给出不同的处理方式和代码逻辑, 如用户名查询、数据切片等操作. 虽然数据包的处理相互独立, 但数据块之间会相互影响, 如 d 代码块会同时处理 a 与 β, 因此即使 a 数据块具有很高的可用性和利用价值, 受 β 数据块影响可能会在后续的测试中受阻, 因此如果模糊测试器将输入报文数据部分视为一个整体, 将会影响变异数据的有效性.

根据上述分析可知, Java Web 应用程序对于报文的

不同数据部分, 采用的处理方式和执行路径并不相同, 通过解析树对不同类型, 不同功能的数据块设置对应的种子池, 并且通过变异组合成输入数据包, 可以更加有效地提升测试用例的整体质量, 测试到深层次的代码区域.

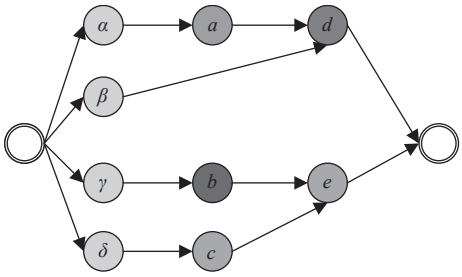

图 2 数据包处理的控制流图

但简单地使用分块处理仍然会存在问题, 如 Peach 这一类模糊测试工具, 在测试过程中, 需要不断地对种子进行解析, 当输入数据的结构较为复杂的时候, 解析所造成的资源消耗就更为明显.

基于以上发现和逻辑, 本文设计了基于解析树的灰盒模糊测试方法, 其主要执行流程如下.

1) 为服务端的输入数据包建模, 根据用户提供的测试用例创建输入数据的解析树, 并为每一个数据结构段建立对应的种子池. 初始种子可以由测试用例得到, 也可以在配置文件中提供. 然后生成如图 3 所示的解析树, 后续的种子选取和生成都会基于该树型结构进行处理. 在本文中定义这些 leaf 为叶子节点, 每个叶子节点下都挂接一个数据块对应的种子池.

2) 结合建立的解析树, 从每一个种子池中挑选种子, 并使用对应数据类型的变异方式进行变异, 最后将它们组合在一起生成新的输入数据. 采用组合的方法避免了对数据解析时候的复杂计算和时间消耗, 并且确保每次生成的数据都符合输入格式.

3) 使用插桩获取代码覆盖率. 如果有新的路径产生, 就将对应的数据块放入相应的种子池中, 并将执行时覆盖的代码块和路径的相关信息进行保存. 依据目标测试程序的反馈, 为种子分配权值.

4) 监测目标程序的日志和它返回的状态码, 判断是否产生异常. 如果产生异常, 就将它记录下来, 以便后续进行异常报告分析.

5) 当被测程序长时间没有异常或是没有新路径产生的时候, 系统会进入下一个阶段: 依据条件概率来指导后续的测试.

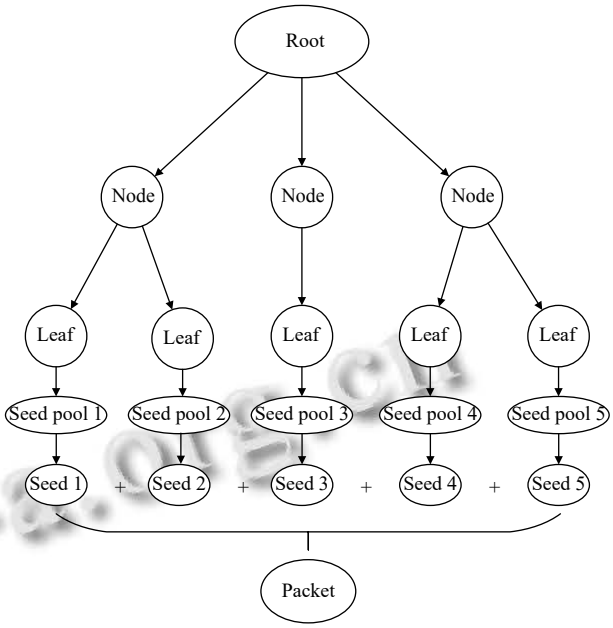

图 3 解析树与数据包拼接

图 3 所示为解析树与数据包拼接, 每一个 leaf 叶子节点为一个单独的结构体, 它挂载对应的种子池, 而种子池中存放的是该数据部分所对应的一些种子, 每一个种子都有一个记录自己相关信息的结构体, 以便后续进行种子权值计算. 其相关信息包括: 该种子的权值, 种子可以覆盖的代码块, 执行路径等. leaf 叶子节点中的节点数据块以及结构体信息通过数据包拼接算法产生测试用例, 设计的数据包拼接算法如算法 1.

算法 1. 数据包拼接生成算法

1) 获取语法解析树根节点.
2) 层序遍历语法树的叶子节点, 从叶子节点数据结构中选取种子, 判断数据类型, 根据数据类型分别转至第 3)–5) 步.
3) 如某些输入数据要求携带系统时间或者含有 CRC (循环冗余校验), 采用随机生成的方法难以满足实际需求, 针对这种情况生成正确的数据产生代替后续拓展的值添加至数据队列中.
4) 如数据分隔符等不进行改变, 直接添加至数据队列中.
5) 如是字符串、数字 (整数或浮点数)、逻辑值等, 则进行相应数据类型的变异, 添加至数据队列中.
6) 从数据队列中依次选取数据等进行数据拼接, 生成初始测试用例.
7) 进行测试用例合理性校验, 输出最终的测试用例.

生成数据包的过程中, 需要进行种子的变异. 本文针对不同数据类型的数据有不同的变异方式, 使得种子变异会更具针对性. 当有新路径产生的时候, 需要将数据块添加到对应的种子池中. 此时需要先创建一个该种子对应的结构体, 并修正其可以覆盖的代码块, 然后将它加入种子池中. 此处的种子记录下自己可以覆

盖到的代码块便于后续的权值计算, 其主要的思想在于将各个数据块与程序中的代码块进行绑定, 理清数据部分与代码块之间的关系. 因此设计了种子添加算法, 如算法 2 所示.

算法 2. 种子添加算法

1) 层序遍历系统创建的解析树, 判断遍历到的节点是否为叶子节点, 为叶子节点转至第 2) 步, 不是叶子节点则继续层序遍历.

2) 如果遍历到的节点是叶子节点, 进行代码块覆盖的修正流程. 将叶子节点的值替换为对应种子池中其他的随机值, 并将其拼接为新的测试用例发送给目标测试程序, 判断是否还能覆盖到这个新的代码路径, 可以则转至第 3) 步, 不能则转至第 4) 步.

3) 如果可以, 说明该数据部分的种子值与产生该新路径没有必然联系, 可以不用将该部分的值加入种子池中.

4) 如果不能, 则意味着该种子值与该新路径的产生有关联. 且修正过程实则也是对目标程序的一次测试, 并不是多余的操作, 故并没有增加额外的消耗.

5) 创建该种子对应的结构体, 填充该部分的值以及它可以覆盖到的代码块, 并将该种子加入种子池中. 通过对种子覆盖代码块的修正可以提高种子携带的信息的准确性, 提高后续种子的权值分配的合理性.

2.2 面向数据包拼接的种子调度权值分配算法

模糊测试的种子调度权值分配是影响测试效果的关键, 代码覆盖率信息是一种被广泛应用于传统软件模糊测试的反馈信息, 并已被证实是有效的. 因此, 本文使用反馈循环来优化基于模糊测试, 并使用代码覆盖率以及执行路径作为反馈评估种子是否有价值, 通过在目标测试程序的分支点插桩来获取这一信息.

但仅考虑到代码的边缘覆盖不能满足解析树拼接实际情况. 图 4 代表了某个程序代码处理 JSON 格式数据流程, JSON_A 和 JSON_B 数据经过 FunC, 在 FunD 中存在一个可能的漏洞点 leak, 且已存在执行路径: JSON_A→FunC→FunE, JSON_B→FunC→FunE, JSON_B→FunC→FunD.

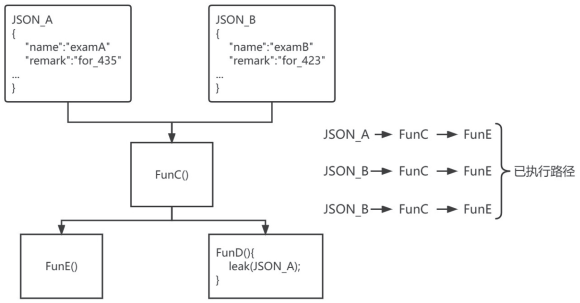

图 4 某段程序执行流程

当存在 JSON_A→FunC→FunD 路径时, 如果使用传统统计代码覆盖率计算方法, 只统计执行到的函数, JSON_A→FunC→FunD 全部被包含了, 但是实际上之前并未有过相同的执行流程, 通过数据包拼接生成的测试用例 JSON_A 的“name”“remark”键值对具有更新至叶子节点种子池中的价值, 所以在程序执行完成后, 需要将执行流程通过哈希运算记录在位图中.

在程序执行的时候, 一些如有多个 if 语句的嵌套语句, 深层次的代码块由于语句成立条件比较苛刻, 难以执行到, 故在测试期间, 该部分的代码块被执行的次数比较少. 对这类可以执行到深层次的代码块的种子组合需要分配更多的权值, 以便之后可以多次被调度. 此外一些低频的代码块在后续测试的时候也可以通过调整种子的权值来增加执行的次数. 因为这类代码和前文的嵌套代码块类似, 由于执行的次数较少, 发现程序漏洞的可能性会高于多次执行过的代码块.

本文在确定数据包结构的前提下, 设计了相应的权值分配策略. 每一个叶子结点对应的种子池中的单个种子的权值计算公式如式 (1):

$$p = p(s_1) + p(s_2) + p(s_3) + \cdots \tag{1}$$

$$p(s_i) = (d_i + \alpha_i)\mathrm{e}^{-\beta c_i + \gamma} \tag{2}$$

其中, d 代表该基本块的执行深度, α 代表执行边界的权值, β 为衰减系数, c 是种子到达该基本块的次数, γ 为一定值, 代表初始的种子系数. $p(s_i)$ 代表的是该种子的覆盖到的某一条执行路径的权值, s_i 代表执行边界或者基本块之间的边. 当某个种子覆盖的代码块越多, 覆盖的代码边界执行过的次数越少, 其权值越大, 后续从各个种子池中进行调度的时候被选择的概率也越高.

为了实现上述的策略, 图 1 中的模糊测试主程序所在的主机端需要拷贝一份目标测试机上的全局位图, 除了产生新的执行边界需要记录在全局位图中, 每一次新得到的代码执行路径也需要记录在全局位图对应的位置上. 在后续进行权值计算的时候, 各个种子依据自己可以覆盖的代码路径到这个全局位图中查找该边界相应的执行次数.

2.3 基于条件概率的模糊测试指导方法

模糊测试器执行一段时间后, 可能会陷入“阻塞”的情况, 即目标测试程序运行了许多次以后依然没有产生新的路径或者再没有出现新的异常, 此时 PTreeFuzz 系统针对这一问题, 不再使用前文提到的权值分配算法, 设计了依靠解析树中叶子节点种子池特征, 采用条

件概率指导后续测试的策略, 突破难以执行的分支路径.

基于条件概率的模糊测试指导方法重点关注在测试过程中执行或者成立次数较少的条件语句, 通过概率分布分析这些能抵达对应代码块的测试用例数据块特征, 通过条件概率优化单个数据块内容, 将后续的测试目标导向这些代码语句, 大量增加测试次数, 对它们进行“重点”突破.

如图 5 是某个程序的代码片段, 其中变量 *a*、*name*、*num* 的值主要与程序的输入 JSON 数据相关, 程序运行到第 19 行代码的条件比较苛刻.

```
…
if (a == '%'){
    name=JSON.getName();
    num=JSON.getNumber();
    if (name==“user” && num<=max ){
        if (num>min):
            v=hex_values[num] + hex_values[name];//line19
        …
    }
}
```

图 5 某程序代码片段

JSON 数据需要满足: *a*==“%”、*name*==“user”且 min<*num*<max 才可测试到 19 行代码, 如使用传统模糊测试方法只可通过不断地变异或者动态符号执行试图突破该“阻塞”路径, 耗费资源较多. 但通过解析树的叶子节点挂接的种子池, 可依据概率提取单个 JSON 键值对特征, 如 *name*=“user”, *a*=“%”等特征, 通过解析树数据包拼接优化语义信息, 提高测试用例突破“阻塞”路径的能力.

如图 6 所示, 采用顺序链表记录在测试的时候遇到的条件语句, 以及在测试过程中该语句的成立情况. *True_times* 和 *False_times* 分别记录下条件语句两种情况的产生次数, 二者中值较小的一个将作为该条件语句的权重, 链表按照各个条件语句的权重从小到大排序除. 当程序运行了许多次以后依然没有新路径产生或者没有产生新的 crash, 就进入模糊测试系统的下一个阶段, 首先重置种子池中的所有种子的权值为 1. 接着依次获取链表中条件语句的权重, 并进行条件语句的条件概率计算.

Record 记录可以到达对应条件语句的种子的组合方式, 包括每一个数据块所采用的变异方式以及使用的种子. 现对该方案的具体流程进行介绍, 在程序进行测试的时候.

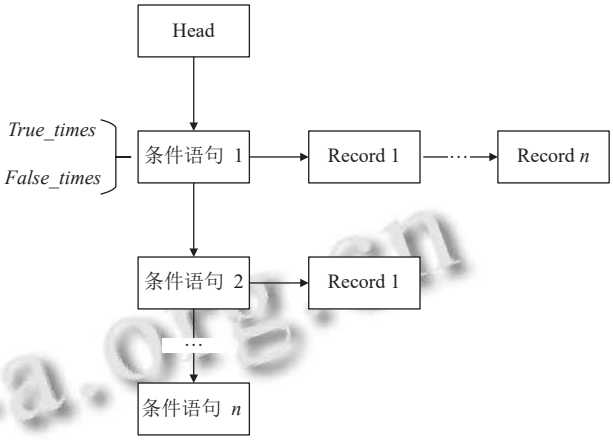

图 6 测试时候使用的链表

1) 首先判断该语句是否在全局位图中记录过, 即是否是第一次执行到, 如果是就将它放在链表头部. 否则就到链表中查询, 如果未能查到, 则说明该语句已经从链表中删除了, 不再继续后续的操作.

2) 如果链表中查询到了, 就按照该语句的成立情况, 在它的变量 *True_times* 或者 *False_times* 上加入此次执行记录 Record, 并根据它的权重插入到链表中合适的位置.

3) 依据 Record 记录的种子及变异信息进行条件语句的条件概率计算, 生成数据包. 其中的条件概率计算部分:

$$p\left(\frac{\varepsilon}{a_{1i}}\right)=\frac{p(a_{1i},\varepsilon)}{p(a_{1i})} \tag{3}$$

$$\begin{aligned} p\left(\varepsilon/a_{1i},b_{1j}\right)&=\frac{p\left(a_{1i},b_{1j},\varepsilon\right)}{p\left(a_{1i},b_{1j}\right)}\\ &=p\left(\frac{\varepsilon}{a_{1i}}\right)\times\frac{p\left(a_{1i},b_{1j},\varepsilon\right)}{p(a_{1i},\varepsilon)}\times\frac{1}{p\left(b_{1j}\right)} \end{aligned} \tag{4}$$

其中, $p(b_{1j})$ 为在种子池 b 中选择了种子 b_1, 采用变异方式 j 生成的测试用例占生成的所有的测试用例的占比. $p(a_{1i})$ 为在种子池 a 中选择了种子 a_i, 采用变异方式 i 生成的测试用例占生成的所有的测试用例的占比. $p(a_{1i}, \varepsilon)$ 为在种子池中 a 中选择了种子 a_1, 采用变异方式 i 生成, 并且测试时候执行到了指定位置 ε 的测试用例占比. $p(a_{1i}, b_{1j}, \varepsilon)$ 为选择了种子 a_1, 采用变异方式 i, 种子 b_1 采用变异方式 j 生成, 并且测试时候执行到了

指定位置 ε 的测试用例占比.

生成数据包时, 需要按照拼接输入的顺序选取第 1 个数据部分, 如果没有顺序的硬性要求则随机选出第 1 个变异的 a_{1i}, 之后通过上述方式计算出 $p(\varepsilon/a_{1i}, b_{nj})$, 接着利用贪心策略选取最优的 b_{nj}, 这样能够保证组合出来的输入数据包可以大概率覆盖需要测试的目标语句 ε.

通过不断的学习之前的测试用例的概率分布, 指导整个后续测试的方向, PTreeFuzz 程序工具可以对一些之前难以达到的测试次数较少的代码块进行逐项突破. 通过之前的测试用例学习其概率的轻量级方案, 不需要大量额外的计算和极具时间消耗的程序分析, 也可以使得程序完成一定程度上的定向测试功能, 并且不会额外增加程序运行需要的时间.

3 系统实现和评估

3.1 系统实现

系统实现如图 7 所示.

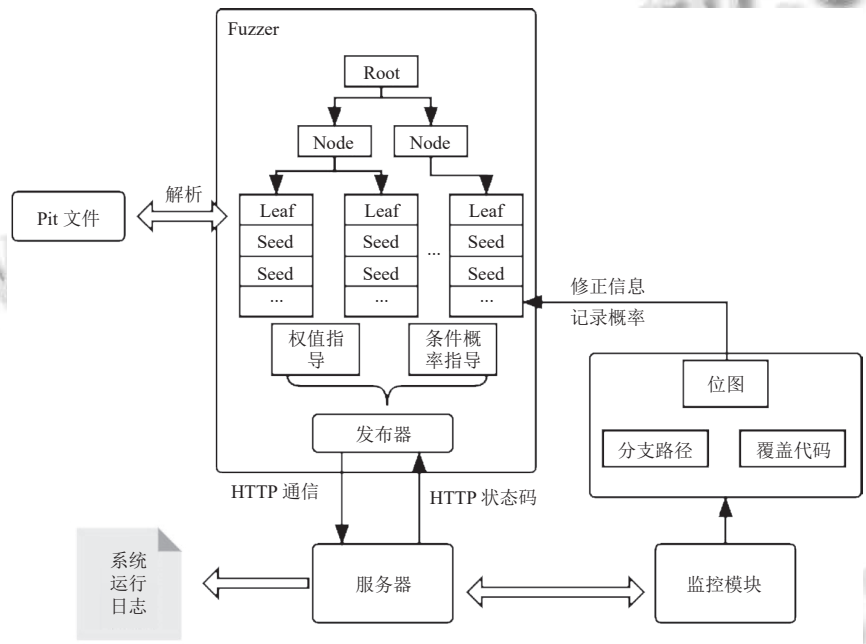

图 7 PTreeFuzz 系统实现

PTreeFuzz 通过发布器模块来配置 HTTP 通信, 它的作用在于发送 PTreeFuzz 系统变异得到的请求报文, 并接受被测 Java Web 应用程序返回的 HTTP 信息. Fuzzer 引擎是整个测试系统的核心, 包含大多数核心代码, 控制整个模糊测试的循环测试, 另外它还负责解析用户编写的 Pit 文件, 通过这个配置文件利用用户提供的测试用例创建解析树, 隔离各个数据块, 解析树将用于后续数据包的拼接生成. 系统还有一个监视器模块, 它负责监测目标被测程序的位图和分支路径信息. PTreeFuzz 主要从两个方面判断 Java Web 应用程序的执行状态, 一个是日志文件, 文件中会记录 Java Web 应用程序在整个测试过程中的运行情况, 包括如果程序出现异常的时候的出错信息等, 这对于判断目标程序情况很有帮助, 另一个是 HTTP 状态码, 可以判断系统是否出现异常. 通过记录下来的这些信息, 可以对这个异常问题进行回溯.

3.2 测试环境

测试环境分为两个部分: 客户端是本文设计的 PTreeFuzz 系统, 服务端是被测目标 Java Web 应用, 即 Java Web 应用程序. 在测试的时候, 由模糊测试工具充当客户端, 被测程序作为服务端对前者发送的数据包进行处理. 客户端和服务端的部署环境相同, 均为: Intel Core i7; 1.80 GHz 处理器; 8 GB 内存; 64 位 Win 10 操作系统.

在使用系统前, 需要对被测 Java Web 程序的输入数据进行建模. PTreeFuzz 系统的 Pit 文件与 Peach 的 Pit 配置文件类似. 系统在使用的时候需要用户提供初始的测试用例, 或者是为每一个数据部分设置指定的

默认值. PTreeFuzz 系统很好地继承了 Peach 的状态模型, 根据配置文件, 系统可以初始化并维护一个有限状态机. 该状态机可以保证系统按照用户指定的执行条件以及定义的顺序进行操作.

3.3 功能测试

为了更好地检测系统的具体效果, 本节会从代码覆盖率测试, 程序异常触发测试, 条件概率指导方案测试, 测试用例的有效率等 4 个方面对该系统进行功能测试.

为了验证 PTreeFuzz 系统在提升代码覆盖率上的效果, 选择 Peach、Kelinci 作为对比对象. Peach 是一款基于生成的模糊测试工具, 通过建模产生种子和测试用例; Kelinci 是一款基于变异的模糊测试工具, 将 AFL 应用于 Java 语言, 通过 AFL 产生种子及测试用例, 因此这两个对比工具分别代表了主流的基于生成、基于变异的模糊测试方法.

本文对 6 款不同的 Java Web 应用程序进行检测, 并且对这些应用的大部分核心类文件进行插桩, 同时在实验测试时添加 Java 代码执行信息记录的功能. Web 应用程序具有许多不同功能的接口, 本文依据其代码覆盖情况挑选数个接口进行测试, 测试结果是这些接口的测试数据的均值. 每次测试持续 10 个小时, 并且重复 10 次, 对这 10 次的结果取其平均值, 具体的测试情况如图 8 所示.

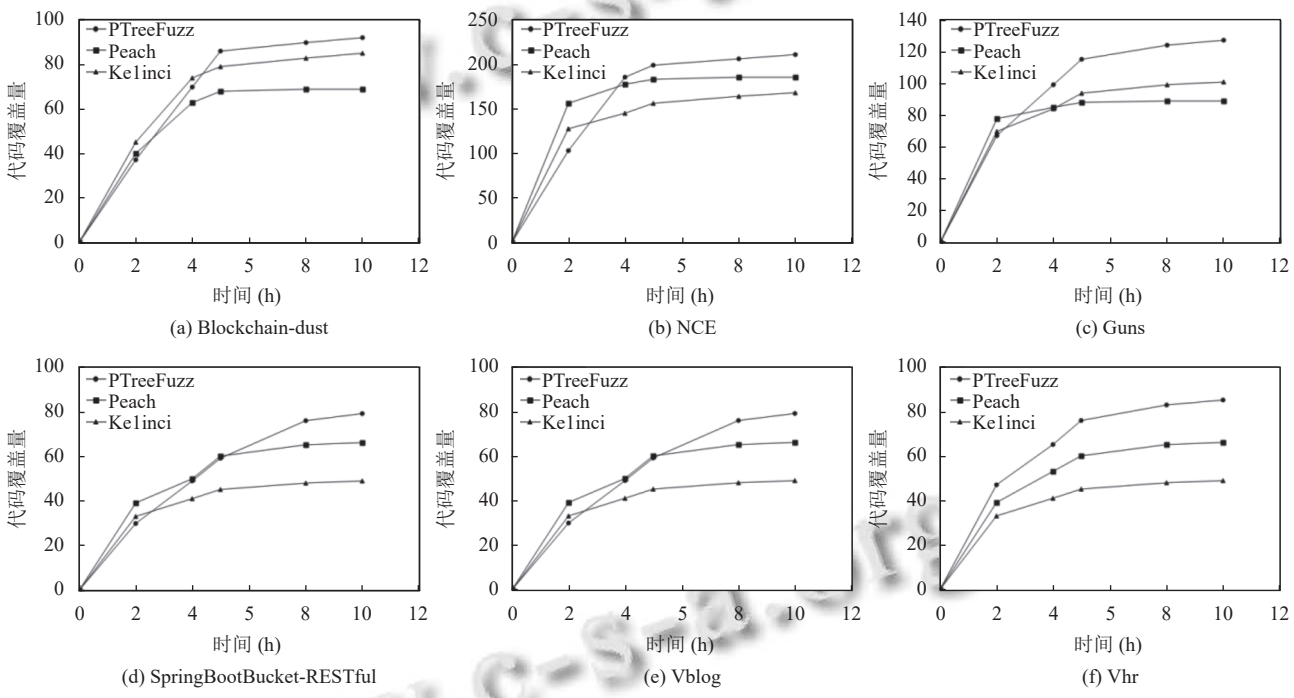

图 8 代码覆盖量对比

图 8 中的 x 轴是执行时间, y 轴是代码覆盖的数量. 可以看到在测试初期, PTreeFuzz、Kelinci 和 Peach 因为测试对象的不同, 覆盖量增长各有优劣, 但随着测试时间的持续, PTreeFuzz 的代码覆盖率逐渐超过 Kelinci 和 Peach, 并在后来扩大优势, 在 Kelinci 和 Peach 测试遇到“瓶颈”陷入路径“阻塞”时, PTreeFuzz 仍然可以稳步增长. 此外, 从图 8 中还可以发现当 Kelinci 和 Peach 在测试一段时间后会陷入“瓶颈”, 代码覆盖率难以再产生突破, 而本系统依靠学习概率分布, 指导测试的方法来缓解这种情况, 在覆盖量趋于平稳之后仍然优于 Kelinci 和 Peach.

被测目标程序的异常触发次数是模糊测试工具是否有效的一个重要指标. 本文同样用 Peach、Kelinci 和 PTreeFuzz 系统进行对比. 针对 5 款不同系统进行 24 h 测试, 每个对象选取数个接口进行测试, 对出现的异常进行简单的去重, 并将数据进行统计, 最后的结果如表 1 所示. 从表中可以看到, 在运行相同时间的情况下, 异常的触发次数和项目本身的质量有密切联系, 代码质量高的项目出现错误的次数明显要少于质量低的项目. 并且本系统触发的异常次数要显著多于 Peach

和 Kelinci 触发异常的次数, 这说明该系统的测试效果更为突出. 经过手动排查系统日志以及触发异常的测试用例, PTreeFuzz 可触发 Peach 以及 Kelinci 发现的所有异常, 并且发现触发的异常大多由有效字符触发, 这表明 PTreeFuzz 大大提升了测试用例的语法语义有效性. 大多系统异常是字符串导致的系统崩溃, 本文定位了这些崩溃大多是由代码健壮性不足引起, 例如 NCE 中发现的空指针异常导致的系统崩溃, NCE 验证 JSON 键值对时因传入的某个键值对为 NULL, 调用 NULL 的 equals 导致的系统崩溃, 这表明 PTreeFuzz 可以有效地发现 Web 应用系统问题. PTreeFuzz 在 Nacos 开源系统中发现了已公开的认证绕过漏洞 CVE-2021-29441, 触发该漏洞需要遵循严格 JSON 语句的语法语义, 在规定 24 h 时间内只有 PTreeFuzz 触发了该漏洞, 这表明通过解析树及数据包拼接技术可以提高测试用例发现系统漏洞的可能.

表 1 程序异常检测情况

应用程序	PTreeFuzz	Peach	Kelinci
SpringBootBucket-RESTful	7	2	3
Blockchain-dust	13	7	5
某Vblog个人管理平台	11	4	3
NCE系统	2	0	0
Nacos	15	7	4

为了验证 PTreeFuzz 系统的基于条件概率的模糊测试指导方案的有效性, 本文将随机产生测试用例的方案和本文的条件概率指导方案进行对比, 比较二者在覆盖指定代码块上的能力. 如图 9 是依次对 Blockchain-dust 的指定代码块 A, B, 和某 Vblog 个人管理平台的指定代码块 C, D 进行 24 h 测试的结果, 其中方案 1 指 PTreeFuzz 使用随机产生测试用例的方案, 方案 2 是指 PTreeFuzz 使用基于条件概率的模糊测试指导方案, x 轴是测试对象, y 轴是覆盖到指定代码块的测试用例占全部测试用例的占比. 图 9 说明由于基于条件概率的模糊测试指导方案会利用之前随机测试时产生的大量测试用例的执行信息, 故它可以在后续执行的时候基于条件概率的模糊测试指导方案让 PTreeFuzz 覆盖指定代码块的占比提高了接近一倍.

为了评估 PTreeFuzz 产生测试用例的质量和效率, 本文收集了 PTreeFuzz、Peach、Kelinci 产生的 1 000 个测试用例, 通过统计 PTreeFuzz、Peach、Kelinci 产生 1 000 个测试用例的时间以及统计 1 000 个测试用例在开源系统 Nacos 中的有效率, 如表 2 所示, PTreeFuzz 可以显著提高 Java Web 场景下种子的有效率, 并且与 Peach 相比产生测试用例的速率更高, 因此单位时间内测试次数更多, 可以在相同的时间内进行更多有效测试.

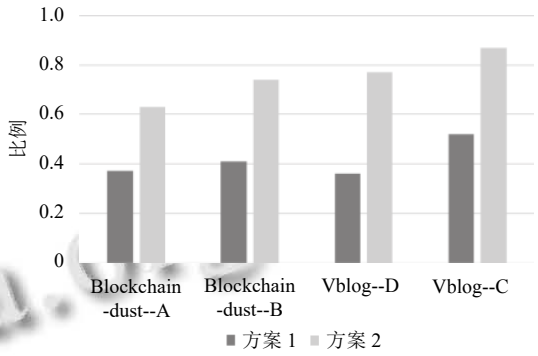

图 9 覆盖到指定代码块的占比

表 2 测试用例质量与效率

测试工具	PTreeFuzz	Peach	Kelinci
时间 (s)	108	437	104
有效率 (%)	68	43	27

4 总结

本文针对现有的模糊测试工具在面向 Java Web 应用漏洞检测时种子有效率低的问题, 提出一种基于语法解析树的 Java Web 灰盒模糊测试方法. 通过用户提供的建模配置文件以及初始样本创建解析树, 并围绕解析树隔离数据中的数据块, 通过面向数据包拼接的种子调度权值分配算法和基于条件概率的模糊测试指导机制, 优化了测试用例中数据块的质量, 并提高模糊测试系统生成测试用例的有效率. 系统实现和评估结果表明, 本文的方法具有更高的代码覆盖率和更好的测试效果.

参考文献

1 Jayaraman K, Harvison D, Ganesh V, *et al*. jFuzz: A concolic whitebox fuzzer for Java. NASA Formal Methods, 2009: 121–125.

2 Sahab A, Trudel S. COSMIC functional size automation of Java Web applications using the Spring MVC framework. Proceedings of the 30th International Workshop on Software Measurement and the 15th International Conference on Software Process and Product Measurement (IWSM

Mensura 2020). Mexico City: CEUR-WS.org, 2020.
3 Martínez S, Cosentino V, Cabot J. Model-based analysis of JavaEE Web security misconfigurations. Computer Languages, Systems & Structures, 2017, 49: 36–61.
4 中国信息安全测评中心. 2022 上半年网络安全漏洞态势观察. http://www.itsec.gov.cn/zxxw/202209/P020220902118368141314.pdf. [2022-12-06].
5 Zou QC, Zhang T, Wu RP, *et al*. From automation to intelligence: Survey of research on vulnerability discovery techniques. Journal of Tsinghua University (Science and Technology), 2018, 58(12): 1079–1094.
6 张阳, 佟思明, 程亮, 等. 模糊测试改进技术评估. 计算机系统应用, 2022, 31(10): 1–14. [doi: 10.15888/j.cnki.csa.008680]
7 Dos Santos JPR, Da Silva KP, Gonçalves BP, *et al*. Selenium as a free tool to test for Java Web application. International Journal of Advanced Engineering Research and Science, 2020, 7(4): 135–139. [doi: 10.22161/ijaers.74.15]
8 张羿辰. 基于代码属性图的 Web 白盒模糊测试方法研究 [硕士学位论文]. 武汉: 武汉大学, 2020.
9 何杰, 蔡瑞杰, 尹小康, 等. 面向 Cisco IOS-XE 的 Web 命令注入漏洞检测. 计算机科学, 2023, 50(4): 343–350. [doi: 10.11896/jsjkx.220100113]
10 倪萍, 陈伟. 基于模糊测试的反射型跨站脚本漏洞检测. 计算机应用, 2021, 41(9): 2594–2601. [doi: 10.11772/j.issn.1001-9081.2020111770]
11 Pham VT, Böhme M, Roychoudhury A. AFLNET: A greybox fuzzer for network protocols. Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification (ICST). Porto: IEEE, 2020. 460–465.
12 徐玲. WebFuzz 的 Web 软件漏洞测试. 软件导刊 (教育技术), 2012, (8): 84–85.
13 Mendez X. Wfuzz—The Web fuzzer. https://github.com/xmendez/wfuzz. [2023-01-05].
14 PEACHTECH. Peach fuzzer platform. https://peachtech.gitlab.io/peach-fuzzer-community/. [2023-01-05].
15 Gutmann P. Fuzzing code with AFL. The Usenix Magazine, 2016, 41(2): 11–14.
16 傅玉, 石东辉, 张阳, 等. 基于覆盖频率的模糊测试改进方法. 计算机系统应用, 2019, 28(1): 17–24. [doi: 10.15888/j.cnki.csa.006714]
17 Kersten R, Luckow K, Păsăreanu CS. POSTER: AFL-based fuzzing for Java with Kelinci. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas: ACM, 2017. 2511–2513.
18 Padhye R, Lemieux C, Sen K. JQF: Coverage-guided property-based testing in Java. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. Beijing: ACM, 2019. 398–401.
19 Nurseitov N, Paulson M, Reynolds R, *et al*. Comparison of JSON and XML data interchange formats: A case study. Proceedings of the 22nd ISCA International Conference on Computer Applications in Industry and Engineering, Caine 2009. San Francisco: ISCA, 2009. 157–162.

(校对责编: 牛欣悦)