



成绩

(采用四级记分制)

西北大学

本科毕业论文（设计）

题目：基于模糊差分测试的 JIT 测试方法研究

学生姓名 杲时雨

学 号 2019116022

指导教师 叶贵鑫

院 系 信息科学与技术学院

专 业 软件工程

年 级 2019 级

教务处制

二〇二〇年六月

诚信声明

本人郑重声明：本人所呈交的毕业论文（设计），是在导师的指导下独立进行研究所取得的成果。毕业论文（设计）中凡引用他人已经发表或未发表的成果、数据、观点等，均已明确注明出处。除文中已经注明引用的内容外，不包含任何其他个人或集体已经发表或在网上发表的论文。

特此声明。

论文作者签名：_____

日 期： 2022 年 6 月 7 日

摘 要

JIT (Just-In-Time) 编译器也称即时编译器, 用于动态编译 Java 类字节码。作为 JVM 中最重要的动态编译模块, JIT 编译器承担着越来越多代码优化、程序性能提升方面的工作。经过研究发现, 尽管近 10 年 JVM 的 Bug 总数呈下降趋势, 但其中的 JIT Bug 占比逐年上升, 同时这些 JIT Bug 的严重程度也在不断增加。如果不对 JIT 编译器进行安全性测试, 那么产生的软件缺陷将对 JVM 和 Java 应用程序的运行产生不可预见的异常, 严重危害 JVM 及其相关程序的稳定性和安全性。

为了有效地对 JVM 的 JIT 编译器进行缺陷检测, 本文设计了一种基于差分模糊测试技术的 JIT 漏洞自动化测试方法, 主要研究内容如下:

(1) 本文期望触发 JIT 优化的场景下, 具有针对性地设计或变异测试用例, 旨在更高效、更深层次的测试到 JIT 的相关问题。本文提出了两种变异算子和一个变异算法, 根据设置的变异迭代次数进行突变, 突变过程受原始种子代码特性的指导。突变后的种子可以增加测试的执行深度, 提升检测效率。

(2) 利用差分模糊测试规则, 本文设计出基于差分模糊测试技术的 JIT 测试方法, 并独立进行差分模糊测试系统的代码实现。与此同时, 在差分模糊测试框架基础上引入“代码特性与 JVM 优化配置”相关性矩阵的思想, 旨在通过输入特定的 JVM 优化选项来更高效的测试 JIT 缺陷。

(3) 本文获取差分模糊系统得到的差分报告, 对存在差异输出的测试用例进行分析, 并讨论其是否存在潜在漏洞。

关键词: 差分模糊测试 JIT 编译器 种子变异 “代码特性与 JVM 优化配置” 相关性

Abstract

The JIT (Just-In-Time) compiler, also known as a just-in-time compiler, is used to dynamically compile Java class bytecode. As the most important dynamic compilation module in the JVM, the JIT compiler is taking on more and more work in code optimization and program performance improvement. After research, we found that although the total number of JVM bugs has been decreasing in the past 10 years, the percentage of JIT bugs in it has been increasing year by year, and the severity of these JIT bugs has also been increasing. If security tests are not performed on the JIT compiler, the resulting software defects will produce unforeseen exceptions to the operation of the JVM and Java applications, seriously jeopardizing the stability and security of the JVM and its associated programs.

In order to effectively detect defects in the JVM's JIT compiler, this paper designs an automated testing method for JIT vulnerabilities based on differential fuzzy testing techniques, with the following main research elements:

(1) This paper expects to trigger JIT optimization scenarios with targeted design or variant test cases, aiming at more efficient and deeper testing to JIT-related problems. In this paper, two mutation operators and a mutation algorithm are proposed to mutate according to the set number of mutation iterations, and the mutation process is guided by the characteristics of the original seed code. The mutated seeds can increase the execution depth of the test and improve the detection efficiency.

(2) Using differential fuzzy testing rules, this paper designs a JIT testing method based on differential fuzzy testing technology and independently carries out the code implementation of the differential fuzzy testing system. At the same time, the idea of "code characteristics and JVM optimization configuration" correlation matrix is introduced on the basis of the differential fuzzy testing framework, aiming to test JIT defects more efficiently by inputting specific JVM optimization options.

(3) In this paper, we obtain the differential reports from the differential fuzzy system, analyze the test cases with differential output, and discuss whether there are

potential vulnerabilities in them.

Keywords: Differential Fuzzy Testing; JIT Compiler; Seed Variation; "Code Features and JVM Optimization Configuration" Correlation

目录

1 绪论.....	1
1.1 研究背景和意义.....	1
1.1.1 选题背景.....	1
1.1.2 选题意义.....	2
1.2 国内外研究现状和进展.....	3
1.2.1 差分模糊测试的相关研究.....	3
1.2.2 JIT 编译器相关研究	4
1.3 拟解决的问题.....	5
1.4 论文主要内容和结构.....	5
2 理论与实验论证.....	7
2.1 JVM 的基本原理.....	7
2.2 JIT 编译器的基本原理	8
2.3 差分模糊测试的基本原理.....	10
3 JIT 编译器测试框架设计	13
3.1 系统概述.....	13
3.1.1 差分框架介绍.....	13
3.1.2 用例变异程序介绍.....	19
3.2 针对 JIT 编译器漏洞的测试设计思路	20
3.2.1 JIT 编译器常见的漏洞	21
3.2.2 测试方法.....	21
4 实验.....	26
4.1 实验设置.....	26
4.1.1 实验对象.....	26
4.1.2 实验环境.....	26
4.1.3 实验设计.....	26
4.1.4 评估指标.....	27
4.2 实验结果分析.....	27

4.2.1 测试用例变异实验.....	28
4.2.2 差分测试实验.....	29
4.2.3 对比实验分析.....	33
5 结论与展望.....	36
5.1 研究结论总结.....	36
5.2 存在的问题与展望.....	37
参考文献.....	38

1 绪论

1.1 研究背景和意义

编译器是当前软件工程领域公认的基础设施之一，大部分高级语言编写的应用都需要经过编译才能运行。作为计算机软件系统的核心应用，编译器的质量决定着其他软件的稳定性，具有非常重要的作用。当前一些流行的编译器（例如 C/C++ 的 GCC 编译器、Java 的 JVM 虚拟机、Go 语言的 GoLand 编译器）都有数以万计的用户在使用，能高效地检测出对应编译器的漏洞是软件工程领域仍需解决的问题。

由于编译器的重要作用，目前业界提出了很多检测技术手段，包括基于变异或规约的测试代码生成技术、差异测试、等价取模测试、不同优化级别测试等。本文通过从差分模糊测试和 JVM 的 JIT 编译器开展研究，针对 JIT 编译器设计特定的测试规则和测试架构，并将差分模糊测试与 Classming^[1]等方法进行比较，发现并验证差分模糊测试对 JIT 编译器测试的有效性。

具体来说，本章先介绍本文的选题背景与意义，再讨论本项目要解决的问题，最后阐述实验步骤和文章结构。

1.1.1 选题背景

Java 虚拟机（JVM）是目前软件工程领域一种常用的虚拟机，用于执行 Java 语言编写的程序。随着软件规模和问题复杂度的不断增加，JVM 在各种应用领域中被广泛的使用，例如移动应用、web 服务器应用、本地管理系统和游戏等。与此同时，随着 Java 应用程序的不断增长，虚拟机的安全性和质量也面临着越来越大的挑战。由于 JVM 的复杂性和高度动态性，使得传统的测试方法难以发现潜在的漏洞和缺陷。因此，需要一种更加高效的测试技术来提高 JVM 的质量和安全性。

模糊测试^[2]是一种广泛应用于软件测试领域的技术，通过自动化生成随机测试用例，将测试用例输入到被测程序当中去探测目标的潜在漏洞和缺陷。在本文

研究中，利用模糊测试技术测试 JVM 的 JIT 编译器相关组件和功能，旨在全方位的去覆盖 JVM 的 JIT 编译器的各个模块。

差分测试^[3] (Differential Testing)，是指检测遵循同一规约的多个软件实现，通过选择相同的测试用例代码 P 和测试输入 I ^[4]，获得不同的应用 C_i 的输出 O_i 。若 O_i 间存在差异，就代表产生差异的应用可能存在一定的漏洞或者缺陷。在模糊测试的基础上。本文将模糊测试技术与差分测试技术结合，实现了差分模糊测试技术，以此提高测试的效率和测试预言的准确性。

1.1.2 选题意义

近年来，JIT 编译器错误变得越来越普遍。从图 1-1 显示了 HotSpot 近 20 年的 Bug 数量分布和 JIT 错误占 Bug 总数百分比^[5]，其中 JIT 错误都是通过 Bug 报告组件的标签来识别的。尤其是近 10 年来，Bug 总数呈下降趋势。但是与之相对来比较，JIT 错误在 Bug 总数内占比增高，这反映着 JIT 模块的缺陷所带来的影响越来越大。

与此同时，JVM 开发人员都会为 Bugs 赋予危险程度^[6]，分为五个级别：P1-P5。P1 的危险程度最高，其他依次减弱。根据 IBM 和 Oracle 公司的统计，JIT 漏洞大部分集中于 P1-P3 程度，并且 P1+P2+P3 错误比例已过 60%。

种种迹象表明，JIT 错误会影响更多版本的 JVM，其漏洞的危险紧急程度也属于 JVM 组件中（包括垃圾回收器、JavaRuntime 等）最高的。因此对 JIT 模块的重点测试与维护，已经成为了 JVM 测试领域不可或缺的一项工作。

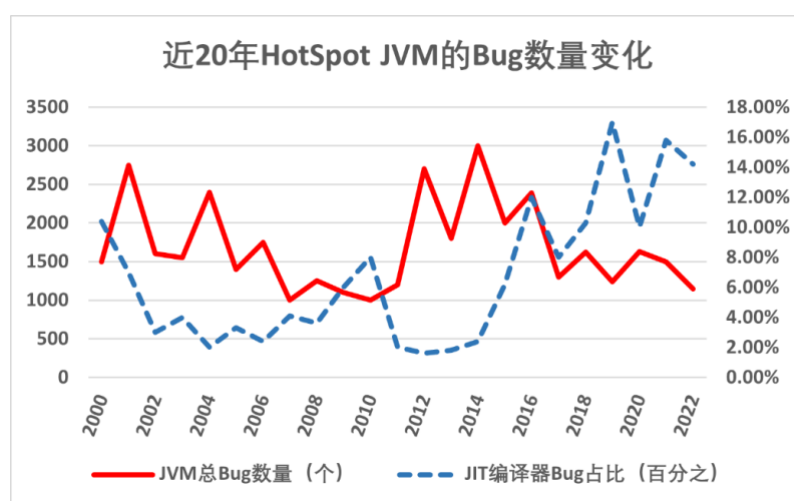


图 1-1 近 20 年 HotSpot JVM 的 Bug 数量变化图

与传统的测试方法相比,通过模糊测试^[7]可以更加全面地覆盖 JVM JIT 编译器中的各种功能和组件,并能够发现更多的漏洞和缺陷。此外,基于模糊测试的测试方法还可以提高测试效率,减少测试时间和成本。本文引入的差分模糊测试技术,对多种 JVM 进行有针对性的差异性检验,通过分析各种 JVM 不同的测试输出结果去探究 JVM 的潜在漏洞。通过差分模糊测试技术,可以在保证测试用例质量的前提下,尽可能全面的去探测编译器的潜在漏洞,以发现更多的缺陷。因此,研究基于差分模糊测试的 JVM 测试技术具有重要的理论和实践意义。

1.2 国内外研究现状和进展

本节首先对差分模糊测试与 JVM 内 JIT 编译器的相关研究现状进行介绍,总结当前领域的发展前景和仍需解决的问题。

1.2.1 差分模糊测试的相关研究

软件漏洞^[8]是计算机系统或者应用程序中存在的故障点或安全隐患,它主要包含三方面的内容:系统敏感性或缺陷、攻击者访问缺陷的可能性和攻击者对缺陷的利用能力。在软件复杂度逐渐升高的背景下,检测软件漏洞^[9]是提升程序质量不可或缺的一步。通过传统的人工编写测试用例并手工传入被测系统^[10],这种方法耗时耗力;使用一些其他的测试方法,如单元测试、功能测试、静态测试等,也不能很全面的去测试出目标的漏洞,所以选择一种高效的测试方法迫在眉睫。

Fuzzing^[11]即为模糊测试,是一种目前较为流行的测试技术。它可以通过不断生成测试用例,连续地将测试用例输入到系统中,旨在暴露系统的未知缺陷。第一个模糊测试工具是于在 1990 年由 Miller^[12]等人开发的,主要目的是寻找 UNIX 系统工具的相关缺陷。后来,发展出了 Web 漏洞挖掘^[13]等技术。如今 fuzzing 技术在已有的概念下进行了发展,已形成了规范的模糊测试系统。许多技术已经用在了 fuzzing 系统当中,主要包括覆盖率反馈^[14],静态分析^[15],污点分析,动态符号执行^[15, 16],Classming 测试技术以及基于覆盖率和导向性的测试用例生成技术^[17]。

而对于差分测试框架,最重要的是如何设计高效的差分测试策略,寻找到同

一测试用例不同的执行路径，以此来发现软件缺陷。基于此，近年来很多团队都进行了更深入的研究。

首先国内相关研究包括：《基于机器学习的差分模糊测试技术研究》^[18]，作者高凤娟、王豫、王林章等，发表时间 2021 年。主要提出了基于深度学习与符号执行的测试与模糊测试相结合的混合测试方法，此方法要求在测试启动前获取模糊测试的路径集，指导模糊测试抵达更深的测试深度。这种方法相较单独符号执行或者模糊测试，提升了 20% 多的测试分支覆盖率，检测出更多缺陷。

国外发表的研究包括：（1）《History-driven test program synthesis for JVM testing》^[19]发表于第 44 届 ICSE 大会，它提出了一种名为 JavaTailor 的历史驱动测试程序合成技术，该技术通过将 JVM 历史漏洞揭示测试程序中提取的成分编织到种子程序中，以覆盖更多的 JVM 行为/路径来合成多样化的测试程序。最后，JavaTailor 使用这些合成的测试程序对 JVM 进行差异测试，取得了不错的测试效果。（2）《Deep Differential Testing of JVM Implementations》^[1]，发表于 2019 年 IEEE/ACM 第 41 届国际软件工程大会(ICSE)上，它介绍了一种名为 Classming 的新颖、有效的深度差异 JVM 测试方法。Classming 的关键是一种技术，即实时字节码变异，用于从种子字节码文件生成有效的、可执行的字节码文件。Classming 有效的揭示了 JVM 的深层次差异，并且触发了 30 多个 JVM 崩溃。

（3）《DifFuzz: Differential Fuzzing for Side-Channel Analysis》^[20]，作者是 Shirin Nilizadeh 等人，发表于 2019 年 IEEE / ACM 第 41 届国际软件工程会议（ICSE）上。它提出了一种基于模糊测试的方法——DifFuzz，可被用以检测时间和空间的旁道漏洞。通过大量评估，发现 DifFuzz 可以揭示流行 Java 应用中未知的旁道漏洞。

1.2.2 JIT 编译器相关研究

JVM 是当前 Java 语言程序最基本的软件基础部件，其中 JVM 的运行效率决定着 Java 程序的质量、高效性。该如何高效的设计和使用 JVM 是至关重要的问题。JIT 编译器为我们提供了有别于通过编译、解释执行 Java 代码的其他方法。JIT 编译器也称即时编译器，是一种在程序执行过程当中进行编译的装置。它可

以将热点代码从 class 字节码转化成机器语言，提升运行速度。

国外的相关研究有：《JVM Fuzzing for JIT-Induced Side-Channel Detection》^[21]，作者为 Tegan Brennan, Seemanta Saha 和 Tefvik Bultan。它发表在 2020 年 IEEE/ACM 第 42 届国际软件工程大会（ICSE）上。它提出了一种用于自动检测 Java 程序中 JIT 引发的时序旁路通道的技术。最终证明 JIT 引发的旁路通道是普遍存在的，并且可以自动检测到。

1.3 拟解决的问题

（1）如何对测试用例进行有效的变异，以此来提高 JIT 检测的效率。

（2）JIT 编译器比 javac 编译器更复杂，对 JVM 的影响程度更深。需要重点测试 JVM 内部 JIT 编译器模块，关注其可能出现的漏洞。探究测试用例特性与 JVM 优化配置相关性，采用 -Xcomp 等优化选项，对 JIT 可能出现的边界检查错误、堆溢出、恶意代码注入、JIT spraying 等错误进行更高效的测试。

（3）如何评估 JVM 实现的安全性和稳定性。通过差分模糊测试，我们可以发现 JVM 中的漏洞和缺陷。但如何准确评估 JVM 底层实现的安全性和稳定性，以及漏洞发现率、漏洞类型等指标，都需要我们进一步探讨。

1.4 论文主要内容和结构

本文主要分为五个部分，具体如下：

第一章 绪论。 本章主要提出本文的选题背景、选题意义，并对目前差分模糊测试领域的研究做简要介绍。在介绍本文基本内容后，再提出本文拟解决的问题。

第二章 理论与实验论证。 本章主要介绍差分模糊测试需要的基本理论支撑。对基本的差分模糊测试概念、JVM 运行概念、JIT 编译器概念做详细介绍，方便后续自动化差分模糊测试框架的编写。

第三章 JIT 编译器测试框架设计。 本章主要介绍本文设计的自动化差分模糊测试框架，以及针对 JIT 编译器测试编写设计思路。最后从历史触发过 JIT 编译器缺陷的测试用例中总结出代码特征和系统设计介绍本章内容。

第四章 实验结果分析。 首先介绍本文基于差分模糊测试的 JIT 测试方法的实验设置，再分析差分模糊测试所得结果，探究所得漏洞的类型和错误原因，最后得出实验结论。

第五章 结论与展望。 本章通过已经完成的实验结果进行分析，得出差分模糊测试对 JIT 模块安全性、鲁棒性以及高效性的实际保障和有效提升。同时发现本文的一些不足之处，对未来本方向的工作研究进行展望。

2 理论与实验论证

2.1 JVM 的基本原理

JVM 指的是 Java 虚拟机，它是一个为可执行 Class 字节码提供运行时环境的抽象机器。设计 JVM 的理由，就是方便高级语言编写的程序在不改变自身代码内容的情况下，可以简单高效的移植到多种操作系统和硬件架构的计算机系统上安全运行。从广义范围^[22]上看，Groovy、Kotlin、Java 等能运行在 Java 虚拟机上的编程语言都是 Java 语言体系的一员。所以我们将 Java 编程语言、JVM、Java 类库三部分合称为 JDK，作为软件工程师开发软件的最小开发环境。其中，JVM 位于 JDK 最底层那一部分，作为开发工具包的基础架构。

JVM 主要由以下部分组成，包括：类加载器、内存空间、垃圾收集器、指令计数器、执行引擎、本地方法接口等内容组成。JVM 的体系结构如图 2-1 所示，箭头代表控制流与数据流标识。

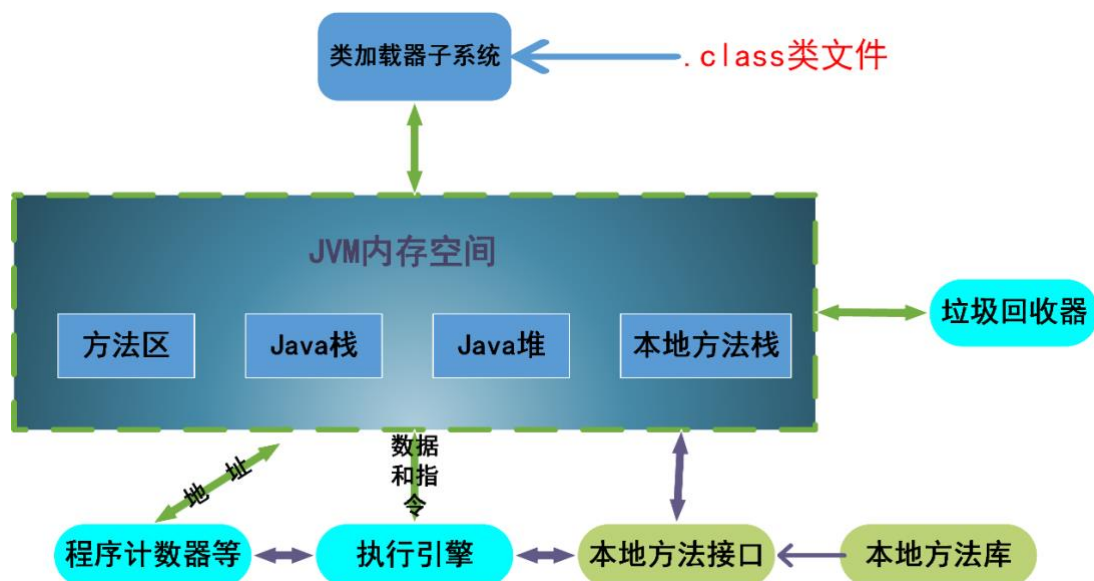


图 2-1 JVM 系统结构图

根据图 2-1 的描述，JVM 工作流程如下：

(1) 生成字节码文件：对 .java 后缀文件（源代码）进行编译，编译器生成 .class 后缀的字节码文件。

(2) 类加载：对于一个大项目来说，可能存在很多类文件以及其他依赖包。此时就需要类加载器对各个类进行有序的导入，最终将所有相关的字节码导入到

JVM 内存空间当中，这一步可以叫做类加载。

(3) 字节码验证：JVM 内设置字节码验证器，可对导入内存空间的字节码进行基本的校验，以确保字节码有效且不违反任何安全约束。这一步称为字节码校验。

(4) 初始化空间：随后在内存空间中，系统将会进行初始化准备工作，首先为类变量分配内存并通过默认值初始化变量。同时对一些特殊的符号引用，JVM 将会选择实际的内存地址分配给符号引用。对于程序中声明的 `static` 变量或者方法，JVM 采用静态初始化并且执行静态代码块。

(5) 执行引擎运行代码：进入执行引擎步骤。选择程序的主入口，启动程序。此阶段将使用 JVM 解释器和 JIT 编译器直接执行字节码，但解释器和 JIT 编译器两者的使用情况不同。对于大部分程序字节码来说，都会先使用解释器执行程序，每次运行到相关序号的字节码时都需要重新解释字节码才能运行；而对于热点代码段来说，JVM 会主动选择新的优化方式来进行编译，显然 JIT 编译器就是此时被使用到的一个功能模块。JIT 编译器可以直接将字节码编译成本机机器码，提高其运行速度。这种做法绕过了每次都需要 JVM 解释器解释字节码的繁琐步骤，对高频使用的代码进行了高效的编译处理。

(6) 监控 JVM 状态：程序不间断运行当中，内存管理机制一直监视 JVM，帮助 JVM 管理内存分配和垃圾回收。它为 Java 对象分配内存，并在不再需要时释放内存。

(7) 运行时性能分析：JVM 内部包含监控和分析工具，帮助开发人员监控程序的行为和性能。

总体来说，JVM 为 Java 程序的运行提供了一个稳定、高效、独立于硬件和操作系统的平台。通过此平台，开发人员可以管理内存分配、代码移植和配置优化提高程序的性能。

2.2 JIT 编译器的基本原理

JVM (Java 虚拟机) 中的 JIT (Just-In-Time) 编译器将频繁执行的字节码动态编译为本机机器码，加速目标程序的执行，以获得更好的性能。对 JIT 的工作

流程进行简要的介绍，流程图如图 2-2。首先 JVM 会先使用它的解释器对.class 文件代码进行解释^[23]，在运行时将字节码翻译成机器码。然后 JVM 会去探测类中的方法执行频率，对执行频率高的代码，JVM 会将其识别为热点代码。在 JVM 的相关配置条件下，热点代码将会被 JIT 编译器进行编译和优化，最终热点代码被编译成本机代码，并存储在代码缓存当中。后续若发现 JIT 优化代码出现错误执行信息，再进行“去优化”回退到字节码解释阶段即可。JIT 流程的优势在于，所有热点代码只需要编译一次，减少了重复解释的步骤，提升了目标程序的执行效率。

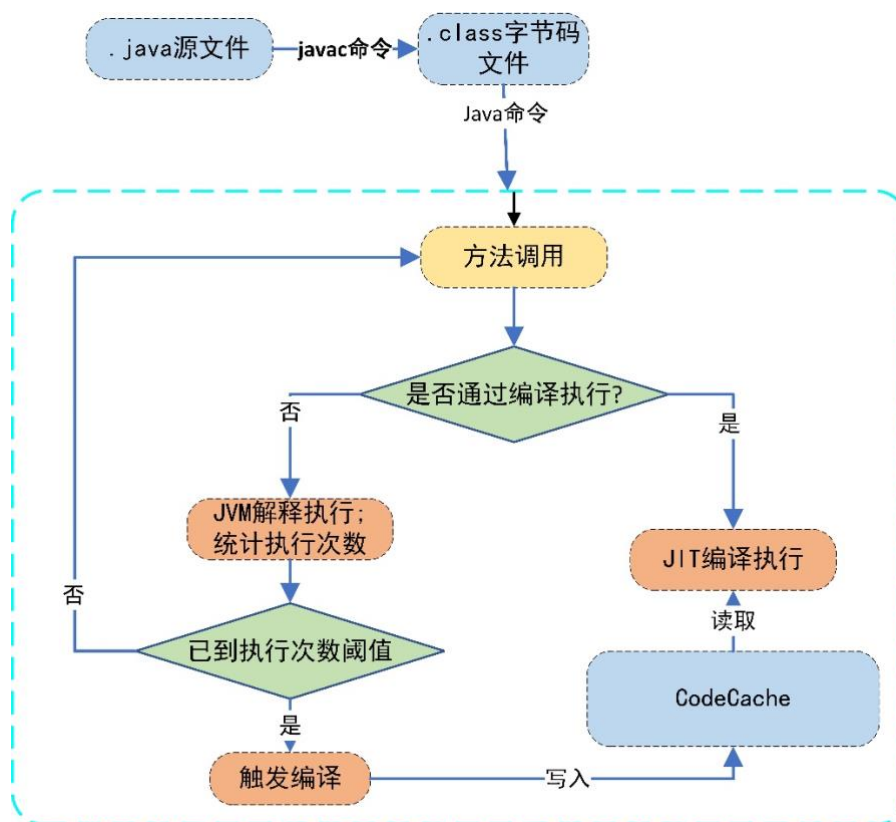


图 2-2 JIT 工作流程图

显然，要实现 JVM 通过控制数据流或控制流高效的执行程序，就需要 JIT 机制搭配多种优秀的优化技术。通常来讲，业界广泛使用的各种 JVM，例如 OpenJ9、OpenJDK HotSpot 和 ZuluJDK 都实现了常用的优化技术，如循环展开、函数内联^[24]、边界检查消除^[25]、死代码剔除、逃逸分析等。

以下是 JIT 编译器的常用优化技术：

- **循环展开：**循环展开会多次复制循环体，减少循环控制结构和数据流的开销和性能占用。

- **函数内联：**编译器会监控方法函数的调用频率，将调用频率高的方法调用直接转换成方法本体。这种做法可以减少方法调用的堆栈开销，将大部分高频方法以代码本体的形式呈现，提高 JVM 的运行速度并减少资源消耗。
- **边界检查：**在保证访问不会越界的情况下，可以消除 JIT 编译器访问数组或其他数据结构时对边界的检查。因为普通情况下，引入边界检查会提升系统开销。
- **死代码剔除：**死代码剔除能实现不去编译那些始终不会执行的代码，减少代码缓存开销。

总体而言，JVM 中的 JIT 编译器使用分析和优化技术将字节码动态编译为本机机器码以提高性能。

2.3 差分模糊测试的基本原理

模糊测试通过向软件输入大量意外的测试用例来暴露软件问题。它的基本思想是通过自动生成测试用例，自动的或者手动的将测试用例输入到系统中，寻找被测系统的缺陷。目前学界已形成了规范的模糊测试系统。该系统包括五个主要的部分^[2]：测试用例生成器、交付模块、目标程序、缺陷检测器和缺陷过滤器。这些模块相互联系，组成了一个完整的模糊测试系统。主要的模糊测试框架展现在图 2-3 当中。

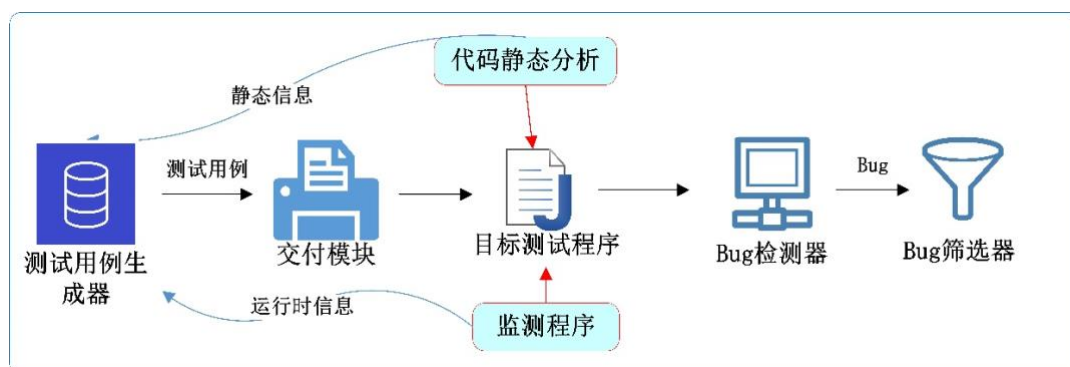


图 2-3 模糊测试系统框架图

首先，对模糊测试框架工作原理进行简要描述。（1）测试用例生成器生成文件，保证用例的语法正确性和覆盖率，以求能达到预期的测试深度。（2）通过交

付模块，将大量用例输入指定的被测系统。同时开启监测模块和代码分析模块，保障测试进行和信息记录。(3) 根据被测系统的类型，选择测试执行的方法。终端命令行程序可直接通过终端指令键入执行，网络 Web 程序需要发送接受 Http 请求来进行测试。(4) 监测模块实时监控，出现错误记录在对应数据库并反馈。

(5) 分析：测试完全执行后，对所有错误结果以及潜在危险结果进行分析，理解程序出错的原因。(6) 报告：向开发人员报告，确认错误。通过以上流程即能实现 Fuzzing。

其次，本文在模糊测试基础上引入差分测试，而 JVM 的差分测试旨在检测遵循同一规约的多个编译器，系统结构如图 2-4。基本工作流程^[26]是，通过选择相同的测试用例代码 P 和测试输入 I ，把它们传入不同的编译器 C_i 当中，生成对应版本的可执行文件 E_i 。将每个可执行文件 E_i 的输出 O_i 记录下来，标记为 $\{O_1, O_2, O_3, \dots, O_n\}$ 。因为本文所测试的一组编译器应该都具有相同的规约，所以对同一个的输入来说，应该会产生相同的输出。所以我们将所有输出记录 $\{O_1, O_2, O_3, \dots, O_n\}$ 进行比较，并开展测试预言投票。一般来说，如果有一半的输出结果都相同，那么我们就确定这种占多数的输出结果为正确结果，而其他存在差异输出的编译器就是差分技术值得去深度分析是否存在漏洞的地方。

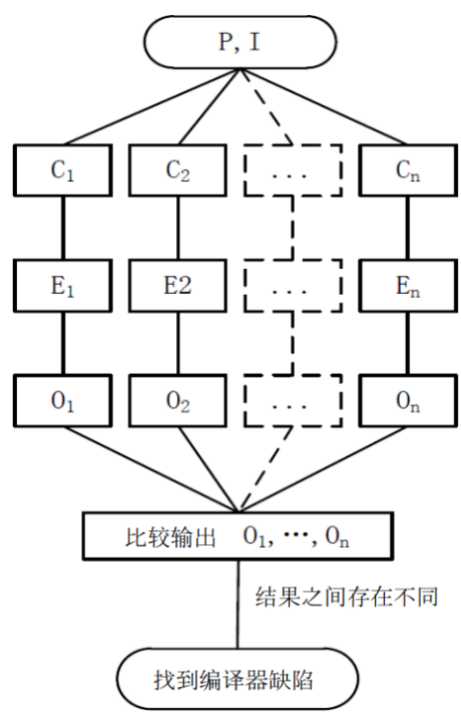


图 2-4 差分测试系统框架图

将模糊测试与差分测试相结合，既可以保证测试用例的充足性和 Bug 监测的高效性，又能保证多被测目标的同时检测。在差分模糊测试技术下，对 JVM 的 JIT 编译器模块测试也能更加深入。本文通过探索 JVM 优化配置选项与测试用例代码特性之间的相关性^[5]，来提升模糊差分测试的高效性。此方法保证对每一个测试用例，差分框架都可以根据其代码特性选择适当的 JVM 优化选项进行测试，以求最大限度的发现编译器的漏洞。

3 JIT 编译器测试框架设计

3.1 系统概述

通过对研究背景和方法的介绍可知,本文旨在设计一种高效的差分模糊测试框架以及合理的 JIT 漏洞检测方法,去发现 JIT 编译器可能存在的潜在漏洞。首先本文已确定要测试的 JDK 版本,主要包括 JDK8 和 JDK11。并已选择测试对象,即参与差分测试的引擎,分别为: HotSpot-jdk8、HotSpot-jdk11、Openj9-jdk8、Openj9-jdk11、Zulu-jdk8、GraalVM-jdk11。通过差分测试系统的交付模块,实现各个引擎的差分测试。如何实现高效的测试流程以及获得正确的差分测试结果,来提升测试速度和测试精度,就是本系统主要完成的创新点和研究方向。

3.1.1 差分框架介绍

首先对设计差分测试流程的简单思路是:测试引擎的选择、测试用例的获取、测试用例的编译、代码特性与 JVM 配置相关性检测、用例程序的执行、监控输出并存储在日志中、分析差异输出向用户抛出差分报告。所以本系统主要分为了以下几个板块: CodeMutation_Module、GetAllFile_Module、Compile_Module、RunClassFile_Module、Summary_Module、CodeAndConfigCorrelation_Module、MySQLConnector_Module 模块。具体的系统设计类图如图 3-1 所示。

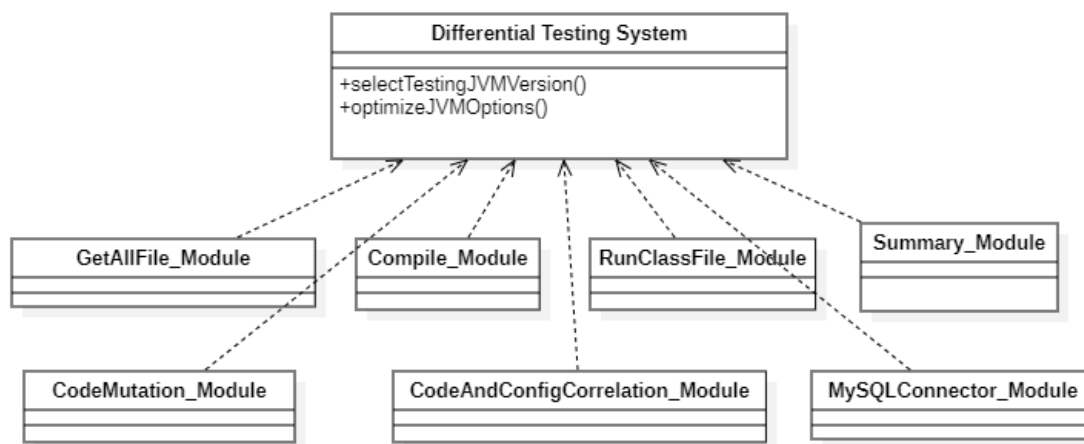
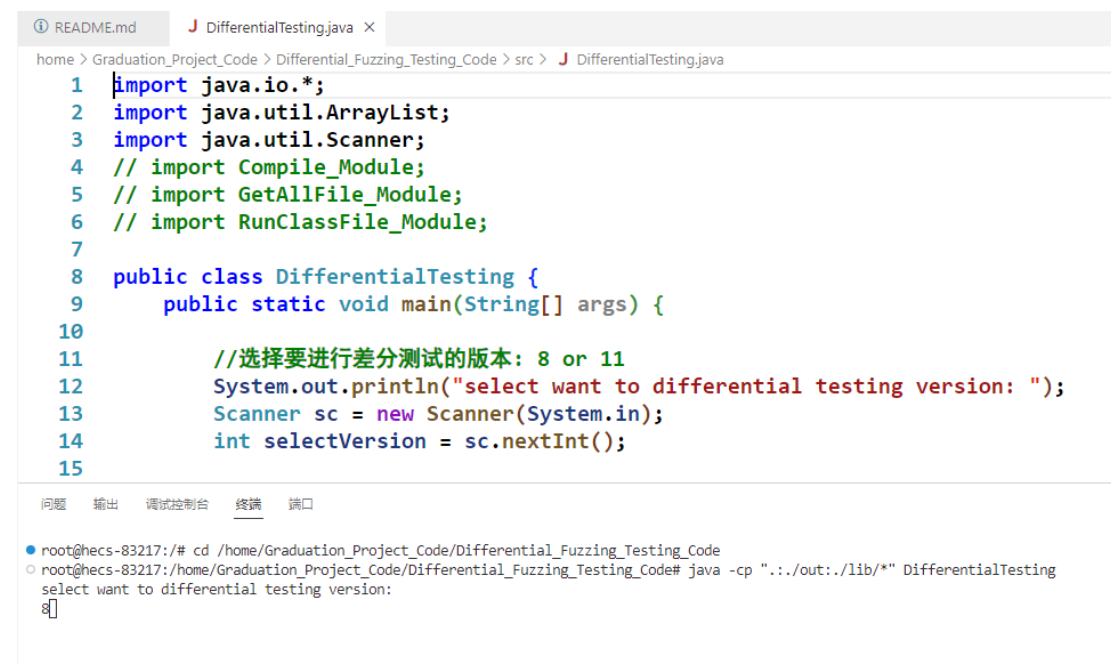


图 3-1 差分模糊测试框架 UML 图

各模块之间相互配合,最终组成差分模糊测试系统。先进行引擎选择。由于

本文测试对象 JDK 版本的多样性，本差分架构设计了一个可由用户选择的差分版本选项，即 jdk8 和 jdk11。用户在运行程序起始时，就可以选择运行哪个版本 jdk，程序会先执行 selectTestingJVMVersion()方法，选择合适的被测版本，界面如图 3-2 所示。版本选择完毕后，即可选择对应的 JVM 进行测试。



```
1 import java.io.*;
2 import java.util.ArrayList;
3 import java.util.Scanner;
4 // import Compile_Module;
5 // import GetAllFile_Module;
6 // import RunClassFile_Module;
7
8 public class DifferentialTesting {
9     public static void main(String[] args) {
10
11         //选择要进行差分测试的版本: 8 or 11
12         System.out.println("select want to differential testing version: ");
13         Scanner sc = new Scanner(System.in);
14         int selectVersion = sc.nextInt();
15     }
16 }
```

问题 输出 调试控制台 终端 窗口

● root@hecs-83217:/# cd /home/Graduation_Project_Code/Differential_Fuzzing_Testing_Code
○ root@hecs-83217:/home/Graduation_Project_Code/Differential_Fuzzing_Testing_Code# java -cp "../out/./lib/*" DifferentialTesting
select want to differential testing version:
8

图 3-2 选择 JVM 版本界面图

(1) 种子变异模块

在编译工作开展之前，有一项工作非常重要，那就是对测试用例进行特征化的变异。类图如图 3-3 所示。该模块主要目的在于，生成代码覆盖率更高的测试用例，以求能更深层次发现到目标系统的漏洞。本系统配置的了多种变异方法，主要有基于正则表达的语法匹配和基于 javaparser 语法解析工具的语法分析方法。在本节不进行详细介绍，将在下节进行变异方法原理分析。

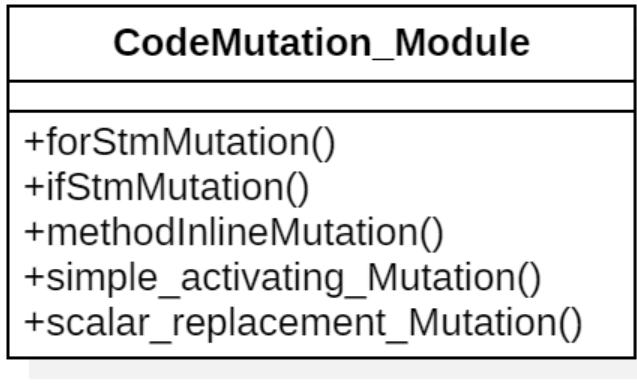


图 3-3 变异模块类图

(2) 文件信息获取模块

接下来，系统需要对将要编译和后续可运行的源代码文件做一次扫描，这一模块称作 `GetAllFile_Module`，即为获取文件信息模块，类图如图 3-4。具体内容是：先将指定测试套件目录下的各个文件路径和文件名称存储在列表中。并且在读取文件过程中识别文件内容，将可直接执行的类（即含有 `public static void main()` 方法的类）进行标记，存储在可执行列表中，在后续的执行模块内直接根据可执行列表进行差异化测试即可。具体使用到的技术也包括正则表达式的语法匹配和基于 `javaparser` 语法解析工具。在 `GetAllFile_Module` 模块中，`getMainAndPackageForClass()` 方法主要负责源代码的类名读取和包路径获取，并判断该源代码程序是否可直接通过 `javac` 执行。

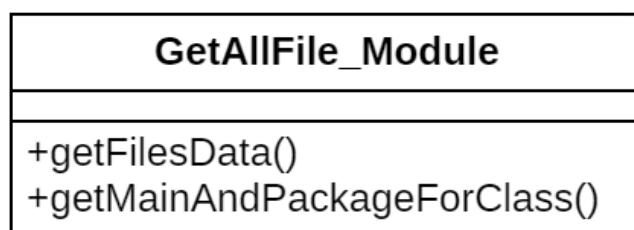


图 3-4 文件获取模块类图

在此简要介绍关于 `javaparser` 语法解析工具对 `GetAllFile_Module` 模块的帮助原理，以解释本模块内源代码内容识别的实现流程。可对以下 `java` 代码 3-1 进行阅读，发现其是一段简单的 `main()` 程序，内含一个 `println()` 方法。

代码 3-1 `JavaParserTest`

```
package com.github.Test;
import java.io.*;

public class JavaParserTest {
    /**
     *获取当前的时间"
     *@param args
     */
    public static void main(String[] args){
        System.out.println(System.currentTimeMillis());
    }
}
```

JavaParserTest 类解析为以下的语法 AST 树：

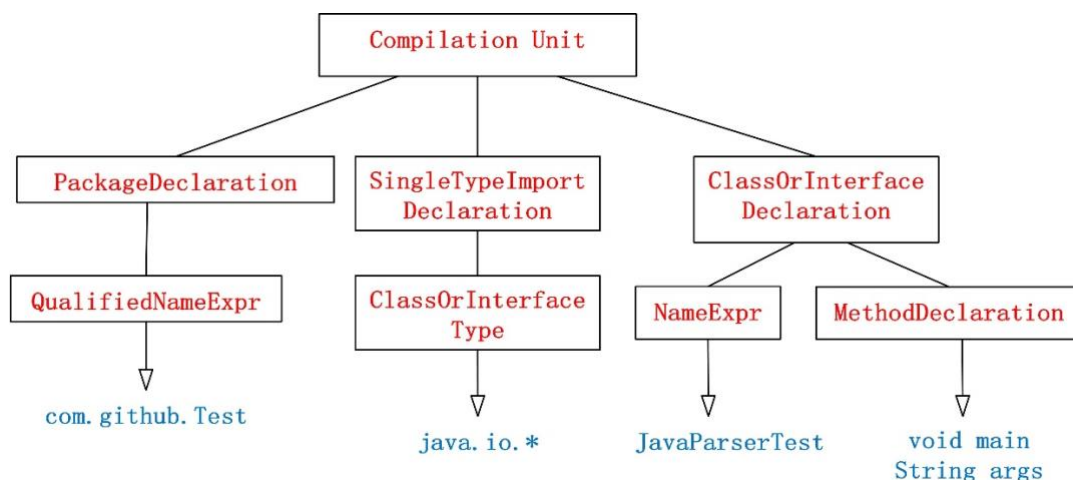


图 3-5 类 JavaParserTest 的 AST 树

由 AST 语法树可知，本类类名为 JavaParserTest，存储在类/接口定义的 NameExpr 属性中。同时本类的方法也存储在 MethodDeclaration 属性中，即本类目前的唯一方法 void main()。对于本类的相关 package 路径和 import 包导入路径，都存储在相应的属性 Expr 当中。所以本文 GetAllFile_Module 模块在探测每个类文件时，选择去读取 ClassOrInterfaceDeclaration 中的 NameExpr 属性来达到读取类名的目的。同时本文通过 ClassOrInterfaceDeclaration 的 MethodDeclaration 属性去探索该类是否包含可运行的主方法，保证测试用例的高效运行，避免去运行那些无法运行的类。当然，main()内部的结构也可以被 javaparser 获取到，此时我们仅需更深层次探索即可，其内部语法树表示如图 3-6。更深层次的 BlockStmt 结构就不再继续分析。

综上所述，可以通过解析 java 代码实现对可执行程序的识别标记。

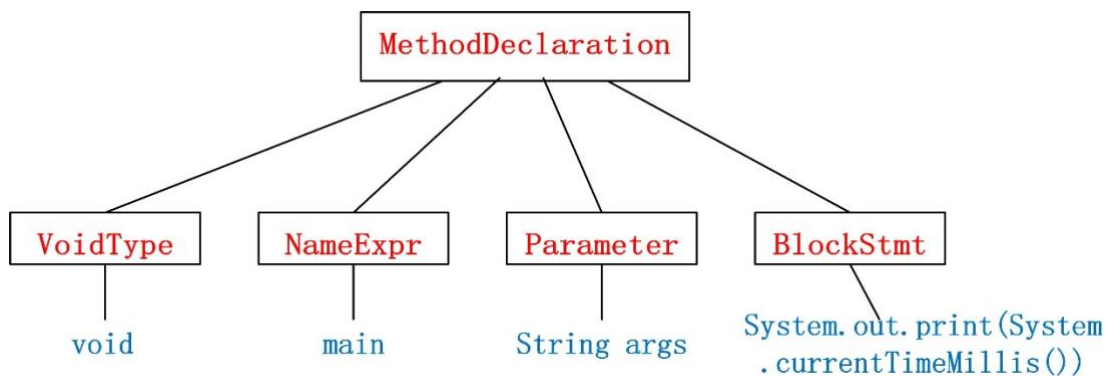


图 3-6 方法 main()的 AST 树

(3) 数据库连接模块

系统选用 mysql-connector-java-5.1.41 版本的 connector 连接 MySQL5。数据库设计为主要的两张表，runcode_jkd8u_list 表和 testcase_jkd8u_list。testcase_jkd8u_list 主要负责存储所有可以进行编译的.java 文件，属性包括 id、filename、canRun(该程序是否可 java 指令执行)、各种编译器编译输出等。testcase_jkd8u_list 的详细表结构如图 3-7。编译模块的日志报告都存储在该表中。

名	类型	长度	不是 null	虚拟	键	注释
▶ id	int	8	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	
fileName	varchar	50	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
canRun	int	2	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
HotSpotCommonOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		HotSpot正常编译输出
HotSpotErrorOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		HotSpot错误编译输出
OpenJ9CommonOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		OpenJ9正常编译输出
OpenJ9ErrorOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		OpenJ9错误编译输出
ZuluCommonOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		Zulu正常编译输出
ZuluErrorOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		Zulu错误编译输出

图 3-7 testcase_jkd8u_list 表结构

同理，runcode_jkd8u_list 存储的是程序进行差分测试的内容。主要用于后期的测试结果进行差分比较，找出有差异输出的编译器。属性包括 id、filename、各 JVM 环境下测试用例输出等。runcode_jkd8u_list 表结构如图 3-8 所示。

名	类型	长度	不是 null	虚拟	键	注释
▶ id	int	8	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	
fileName	varchar	50	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
HotSpotCommonOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		HotSpot正常运行输出
HotSpotErrorOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		HotSpot错误运行输出
OpenJ9CommonOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		OpenJ9正常运行输出
OpenJ9ErrorOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		OpenJ9错误运行输出
ZuluCommonOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		Zulu正常运行输出
ZuluErrorOutput	longtext		<input type="checkbox"/>	<input type="checkbox"/>		Zulu错误运行输出

图 3-8 runcode_jkd8u_list 表结构

(4) 编译模块

Compile_Module 模块，本文一般直接选择 javac 命令，并采用多个 Java 编

译器对用例进行编译。由于需要使用相同规约的不同编译器，所以记录每个 jdk/bin 文件下的 javac 程序，以便切换不同的编译器。编译的正确与错误结果都将存储在对应编译器的 commonOutput 和 errorOutput 中，以供查看失败编译信息。

(5) 代码特性与 JVM 配置相关性匹配模块

即 CodeAndConfigCorrelation_Module 模块。对于一般的差分模糊框架来说，仅仅考虑了如何实现差分功能。而对于本文来说，探究了一种测试用例代码特性与 JVM 优化选项之间的相关性矩阵，系数值范围 value=[0-1]。系数越接近 1 就代表两个维度的相关性越大，越接近 0 就代表相关性越小。相关性代表使用某一测试用例进行测试时，同时采用对应的 JVM 优化选项可能触发 JIT 缺陷的概率。相关性矩阵例图如图 3-9 所示。

	CODE	-XX:LOOPUNROLLLIMIT	XX:THREADSTACKSIZE	-XX:+USEBIASEDLCKING
	FEATURES			
	LOOP ₂₊	a ₁	a ₂	a ₃
	SYNCHRONIZED	b ₁	b ₂	b ₃
	ACCESS			
	INVOCATION	c ₁	c ₂	c ₃
	FIELD ACCESS	d ₁	d ₂	d ₃

图 3-9 代码特性与 JVM 配置相关性矩阵

其中 {a₁,a₂,a₃} 等元素，就代表两个维度的相关性。

(6) 差分运行模块

JVM 存在多个优化选项，本系统可参照相关性矩阵，根据代码特性对每个测试用例定制化的选择优化选项，以求通过优化发现漏洞。确定好相关优化选项后，开始在台 JVM 上差分测试。

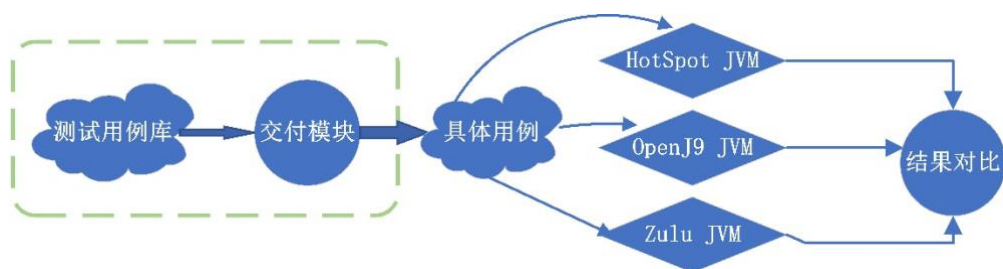


图 3-10 一个测试用例在本系统的运行流程

正如图 3-10 所示，将指定的配置选项载入 java 运行命令当中，等待测试结果进行对比即可。

(7) 结果比较模块

Summary_Module 中，将有差异的结果进行比较输出，存储于差分测试结果日志文件中即可。文件内格式如图 3-11。

```

jdk8u_summary_log.txt X
home > Graduation_Project_Code > Differential_Fuzzing_Testing_Code > LogBackup > Original_Without_Xcomp > jdk8u_summary_log.txt

217
218 类名: MontgomeryMultiplyTest    本代码存在差异输出的编译器如下:
219
220
221 类名: TestEscapeThroughInvoke    本代码存在差异输出的编译器如下:
222 HotSpotErrorOutput ***** OpenJ9ErrorOutput
223 OpenJ9ErrorOutput ***** ZuluErrorOutput
224
225
226 类名: Test6982370    本代码存在差异输出的编译器如下:
227
228
229 类名: TestG1HeapRegionSize    本代码存在差异输出的编译器如下:
230 HotSpotErrorOutput ***** OpenJ9ErrorOutput
231 OpenJ9ErrorOutput ***** ZuluErrorOutput
232
  
```

图 3-11 差分结果分析

3.1.2 用例变异程序介绍

通过 JOpFuzzer^[5]、Classming^[1]等论文研究发现，初始的种子很少能发现目标系统的漏洞，50%以上的漏洞都是通过变异测试用例发现的。而在模糊测试当中，测试用例的生成也分为基于符号执行、基于污点分析和基于遗传变异的三种方式。各种生成用例的方法都是以能覆盖更多的测试执行路径，提高测试效率为目的。所以通过以上可知，普通的测试用例对 JVM 的测试深度有限，提高测试

种子的质量就是测试工作的非常关键一步。

本文的测试套件代码来自于 `github` 开源网站上高 `star` 项目，包括 `Tcases`（一个基于模型的测试用例生成器）和 `Auto-Unit-Test-Case-Generator`（旨在生成具有完全自动化的高代码覆盖率单元测试套件）。其中 `Auto-Unit-Test-Case-Generator` 使用基于搜索的软件测试（`SBST`）作为主要算法框架，来提升测试用例随机字符串组合的能力。并采用 `accutare` 搜索算法，来减少生成器的性能开销。

当测试用例准备完毕，系统就可以开展对用例的变异工作。借助 `javaparser` 语法树的作用，可以轻松地获得一个方法语句块内部是否包括 `loop` 循环、`if` 语句、`switch` 语句、方法调用、逻辑运算等信息。使用 `stmt.isForStmt()` 或 `stmt.isWhileStmt()` 对单层 `loop` 循环变异成多层 `loop` 循环嵌套，使用 `stmt.isIfStmt()` 方法对 `if` 语句进行变异等。

显然，变异时需要考虑新用例的复杂度问题以及覆盖率问题。当内层存在一个非常大的循环时，外层的循环就不应该过于庞大，否则会面临测试性能低下、单个测试用例运行时间太长等问题。所以在此问题的基础上，本文用例变异程序会计算每个循环体复杂度 $O(n)$ 的峰值，在变异时动态的赋予外层迭代循环长度，保证时间复杂度和空间复杂度在可控范围内。

本章截至目前，测试用例以及测试框架已完成部署。

3.2 针对 JIT 编译器漏洞的测试设计思路

本文目标是通过模糊差分测试，研究 JVM 内 JIT 模块的优化安全性及缺陷存在。由于承担着代码优化工作，JIT 编译器相较于 `javac` 命令下的编译器要复杂的多，漏洞也会复杂得多，JIT 内部可分为客户端编译器（C1 编译器）和服务端编译器（`opto` 或 C2 编译器）^[27]。C1 编译器启动速度快，峰值性能较低，主要包括局部优化，如内联；C2 编译器对长时间运行后台的程序启动时间更长，关注全局优化以及基于程序运行时信息。所以对于 JIT 及其相关漏洞，需要有基本的认识 and 了解。

在此基础上，阐述本文的主要研究方法：基于测试代码特性与 JVM 优化选项的二维相关性探索与测试。主要解释如何实现相关性赋值探索和差分测试。

3.2.1 JIT 编译器常见的漏洞

JVM 的安全性不仅依赖于字节码验证器、类加载器等组件，也依赖于 JIT 编译器。JIT 编译器直接处理字节码序列，动态的将字节码编译为本机代码，这给攻击者提供了非常多的漏洞。

(1) 恶意代码注入攻击^[28]：攻击者会选择程序运行时，利用 JIT 的动态特性向文件内注入恶意代码。JIT 将恶意代码与正常代码都编译为本机代码，这样攻击者就可以利用 JIT 的漏洞进行恶意攻击。

(2) 整数溢出攻击：Java 中变量包括包装类基本变量和普通数据类型变量，攻击者可以利用包装类变量的溢出来攻击堆内存区，这是因为大部分包装类基本变量对象都存储在堆中。通过造成堆区的溢出，可以干扰程序的正常执行甚至触发意外行为导致程序崩溃。

(3) 堆区溢出：由于堆区的变量可以长期存储，所以恶意代码对堆区的操作往往会产生很大影响。不仅限于上文提到的包装类变量，大部分对象和方法都可以被攻击而产生堆区溢出。通常 JIT 自动的分配内存给程序，若存在数据未被验证就传入堆区当中，容易造成堆区溢出。

(4) JIT spraying 漏洞^[29]：是一种计算机漏洞类型，主要是通过 JIT 可以动态即时编译的特性，来规避地址空间布局随机化（ASLR）和数据执行保护（DEP）。由于 JIT 编译器不能在无可执行数据环境中运行，所以 JIT 不存在数据执行保护机制。在此基础上，就可以利用漏洞进行 JIT 的堆 spraying，也称堆喷射。攻击者可能会操纵指令的操作地址，对相关地址进行重定向，最终造成 CPU 无法以正常方式执行指令。

3.2.2 测试方法

目前的大部分研究项目，如 Classming 仅关注种子的数据流与控制流；JavaTaiLor 是通过从历史测试用例中选取部分片段，随机插入到种子池当中进行变异，生成新的种子文件。但以上的测试技术大部分关注种子的输入问题，很少考虑针对于某一特定 JVM 优化配置情况下漏洞的测试工作。

通过 JOpFuzzer 论文研究可知，在测试 JVM 时，包含特殊代码特性的测试用例才能更高效地测试出 JIT 编译器问题。更重要的是，61.7%（333/540）的 JIT 缺陷仅在非默认的 JVM 优化配置选项下才会被显式地测试出来^[5]。所以本文计划从以上两个维度来进行研究，充分利用测试代码特性与 JVM 优化选项之间的关联性，才能更高效的测试出 JIT 错误与缺陷。

本文所构建的系统，旨在设计二维矩阵去存储代码特性与 JVM 优化选项之间的相关性，并利用相关性指导测试工作的开展，矩阵数据结构如前文图 3-9。具体工作流程如下：

A.初始化

初始启动系统时，将进行历史编译记录分析与相关性矩阵更新工作。系统数据库中会提前建好名为“codeFeatureAndJVMOptionCorrelation”的表，各属性名分别为（代码特征、JVM 选项、JVM 选项取值，相关性系数、已匹配的种子数量），其中主键为（代码特征、JVM 选项、JVM 选项取值）。矩阵的 MYSQL 表结构如图 3-12 所示。

本文提前获取了 Java 程序所有基础的代码特性，例如 if 语句、方法调用、算术运算符、逻辑运算符、移位运算符、单层循环、多层循环、Array 操作语句、Lambda 表达式、同步访问等。同时收集所有的 JVM 可用选项放入列表中，在舍弃掉与优化无关的选项之后，从列表中选择 70 多个主流优化策略相关的选项，根据 JVM 开发人员的推荐初始值进行矩阵的初始化，即属性（代码特性）与（JVM 选项、JVM 选项取值）进行笛卡尔乘积，生成代码特性与任意一种优化选项搭配的表项。具体来说，JVM 选项若为布尔类型，取值为 0/1；若为整形，取值整形即可。当然，在具有先验知识的前提下，初始化还可以更简单高效，避免了毫无关联的特性与 JVM 选项之间的二维匹配。

对象	codeFeatureAndJVMOptionCorrelati...							
保存	添加字段	插入字段	删除字段	主键	上移	下移		
字段	索引	外键	触发器	选项	注释	SQL 预览		
名	类型	长度	小数	不是 null	虚拟	键	注释	
codeFeature	varchar	40		<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	代码特性	
JVMOption	varchar	40		<input checked="" type="checkbox"/>	<input type="checkbox"/>	2	JVM优化选项	
JVMOptionValue	float			<input checked="" type="checkbox"/>	<input type="checkbox"/>	3	选项值	...
correlation	varchar	40		<input checked="" type="checkbox"/>	<input type="checkbox"/>		相关性系数	
seedNum	int	11		<input checked="" type="checkbox"/>	<input type="checkbox"/>		已匹配的种子数	

图 3-12 相关性矩阵数据库表结构

数据结构完成后，系统从历史触发过 JIT 编译器缺陷的测试用例中进行特性读取和运行指令检查。将读取到的代码特性记为 F_i ，将运作指令中的 JVM 选项配置记为 C_j ，通过公式（3-1）的计算即可更新对应表项的相关性系数 R_k 。

$\text{LimitCount}(F_i, C_j)$ 代表代码特性为 F_i 且历史触发过 JIT 缺陷的测试用例中，有多少个用例运行代码时使用了 JVM 优化选项 C_j ，该函数返回符号条件的用例数量。同样， $\text{Count}(F_i)$ 代表历史触发过 JIT 缺陷的测试用例中特性为 F_i 的代码数量。

$$R_k = \text{GetCorrelation}(F_i, C_j) = \frac{\text{LimitCount}(F_i, C_j)}{\text{Count}(F_i)} \quad (3-1)$$

经过计算，将新的 R_k 更新到数据库对应表项即可。经过先验知识训练和一定数量集的初始准备，目前的相关系数矩阵已经初具雏形，可以用于简单的差分模糊测试工作。

B. 种子变异

如差分框架介绍章节所讲，在测试工作执行前，会根据原始种子的特性进行代码变异。本文实现了较为简单的几种变异算子设计，主要目的是为了增加现有控制流中基本块的数量而不改变原有程序的正常运行流程，所以设计变异语句为 if 语句和 loop 语句。

本文还专门为测试用例变异程序设计了一种算法，旨在保证种子变异迭代次

数受控制的前提下进行充分变异，具体算法如代码 3-2 所示。Seed 是输入的种子；Count 是当前已经变异的次数；Body 是代码块，用于变异代码是嵌套使用。当 $\text{Count} > \text{limitSize}$ 时，就表示已经到达变异次数顶点，此时输出变异体；其他情况下，执行 `getCodeFeature()` 函数获取原始种子的代码特性，选择进行 `if()` 或者 `while()` 语句变异，并继续递归变异函数，直到到达变异次数顶点。

代码 3-2 MutationAlgorithm

```

Input: Body, Seed, Count
Output: Mutation

function MutationAlgorithm
    if (Count > limitSize) then
        return Mutation  $\leftarrow$  Seed
    selection  $\leftarrow$  getCodeFeature(Body, Seed)
    if (selection == 0) then
        expr  $\leftarrow$  "while(limit){Body; --limit;}"
    else if (selection == 1) then
        expr  $\leftarrow$  "if (true) {Body;}"
    mutation  $\leftarrow$  seed.update(expr)
    ++Count;
    return MutationAlgorithm(expr, mutation, Count)

```

C. 参数配置选择

经历了前期的准备工作，参数配置阶段就可使用完备的（代码特征&JVM 优化选项）相关性矩阵进行分析。首先利用 `javaparser` 探测出测试用例的代码特性 F_i ，再针对特性 F_i 进入数据库寻找包含该特性的最高相关性系数 R_k 。最终，定位 R_k 相关性系数的表项，找到最佳优化配置 C_j 和推荐配置值 V_j 即可。

目前较频繁使用的相关性组合有（`loop2+`，`-XX:LoopUnrollLimit=Value`），`-XX:LoopUnrollLimit=Value` 表明 JIT 将展开循环数小于 Value 的循环体。所以在进行 JVM 优化选项配置时，可以解析测试用例循环体的迭代次数，选择合适的参数填入即可。

D.漏洞检测

从相关性矩阵和测试用例角度来看，大部分工作基本已完成，所以准备执行测试用例，进入漏洞检测环节。此模块需要前期的变异种子以及系统提供好的 JVM 优化配置选项，然后执行程序并记录相关运行结果。此模块的测试报告和测试用例需要进行留存，以方便下一次初始化相关性矩阵时，本次测试用例作为“历史触发过 JIT 编译器缺陷的测试用例”训练并更新模型。

通过该差分模糊系统的多次测试，相关性信息也被循环训练，最终能提高该系统的测试效率和稳定性。

4 实验

4.1 实验设置

4.1.1 实验对象

本论文所研究的实验对象为 Java 虚拟机 (JVM)，以及 JVM 中的 JIT 模块。我们将选取一些常见的 JVM 实现作为实验对象，例如 Open JDK、Oracle JDK、Zulu JDK 和 GraalVM JDK 等。

4.1.2 实验环境

硬件方面，本论文的实验全程在 Ubuntu 18.04（内核版本 4.15）操作系统下进行。实验平台包括一台安装有 Linux 操作系统的高性能服务器，主频 3.6GHz 的 Intel i7-7820x 处理器和 64GB 内存。为了保证实验的可重复性，我们将使用 Docker 等工具来构建实验环境。

软件方面，本文的开发语言是 Java，用到的 JDK 版本为 jdk8 和 jdk11，主要使用到的 JVM 有 HotSpot 8.0.332、Openj9 8u342-ga、Zulu 8.60.0.21、HotSpot 11.0.14、Openj9 11.0.16+8 等。开发工具为 VS Code/IDEA。

4.1.3 实验设计

本论文将采用实验研究方法，通过对 JVM 实现进行差分模糊测试来评估其安全性和稳定性。具体实验设计将包括以下步骤：

- ①选取目标 JVM 实现，确定测试对象和测试用例；
- ②构建实验环境，包括安装和配置各版本 JVM 和相关的工具、构建测试环境等；
- ③使用模糊测试工具对 JVM 的内部模块 JIT 实现进行测试，生成测试报告；
- ④分析测试报告，评估 JIT 的漏洞和安全性。

4.1.4 评估指标

模糊测试技术包含很多实验指标，例如：

- **漏洞发现率^[30]**：漏洞发现率是软件测试中用来衡量漏洞检测技术有效性的指标，它是特定技术或工具检测到的已知漏洞的百分比。简单来说，漏洞发现率是一种技术或者工具可以检测到的漏洞数量与被测软件中存在的漏洞总数的比值。例如，某系统存在 50 个已知漏洞，而一个工具可以检测出其中 20 个已知漏洞，那么此工具的漏洞检测率为 40%。高检测率代表着此检测工具能高效的识别漏洞，表明了它识别软件漏洞的能力。
- **漏洞类型^[31]**：在软件测试中，攻击者可以利用系统本身存在的弱点，针对于某一软件漏洞进行攻击。由于软件的复杂性，软件漏洞的种类也层出不穷，每种漏洞都会根据特定方式对系统造成风险。常见的软件漏洞包括：注入漏洞、访问控制漏洞、缓存区（内存）溢出漏洞、加密漏洞等。所以在测试工作中对漏洞进行合理分类，就可以更全面探测系统缺陷。
- **JVM 差异**：本文统计不同 JVM 之间出现的差异。通过分析这些差异，来将潜在漏洞提交给 JVM 开发团队进行缺陷确认。

本论文将对差分模糊测试生成的结果进行分析，包括漏洞发现率、漏洞类型、不同 JVM 差异等。同时引入对比实验，对比实验对象是 Classming 模糊测试技术，记录两者评估指标之间的差异。通过分析这些数据，本文将能够评估 JVM 内部 JIT 编译器的部分漏洞和实现差异，并针对具体漏洞，进一步改进差分模糊测试框架。

4.2 实验结果分析

本文通过使用“基于相关性矩阵的差分模糊测试框架”检验 JIT 模块，在 JVM 选项配置和测试用例代码特性的多种搭配情况下测试出各编译器的一些异常问题。以下从测试用例变异、相关性矩阵更新和差分测试等方面进行实验结果分析。

4.2.1 测试用例变异实验

要对 JVM 进行漏洞检测，就需要一套合适的变异算子和变异框架保证测试用例的质量。本文结合实际测试用例生成场景，将变异算子引入到变异程序当中，主要实现了 loop 代码块与 if 代码块的变异工作。如图 4-1，可观察种子文件变异前后的情况。图 4-1(a)仅包含一层循环，而经过用例变异程序执行后，用例在外部多了一层循环 loop(j)。同时采用 javaparser 解析器对变异语句附近的代码结构进行安全性分析处理，保证在本实验的 1200 多个测试用例中，72.4% (868/1200) 的原始种子可以被充分的安全变异，不用担心变异后数据流发生意外错误（例如，①变异后无法找到变量声明语句；②变异后方法内返回值语句丢失等）。

<pre>public int MutationTest { int result = 0; int[] arr = new int[10]; for (int i=0; i<arr.len();++i) { arr[i] += i; result += arr[i]; } return result; }</pre>	<pre>public int MutationTest { int result = 0; int[] arr = new int[10]; for(int j=0;j<20;j++){ for(int i=0;i< arr.len();++i) { arr[i] += i; result += arr[i]; } } return result; }</pre>
(a) 用例编译前	(b) 用例编译后

图 4-1 测试用例 loop 语句变异对比

通过本文差分模糊测试验证得出，在总共 217 个可执行类文件中，原始种子仅能测出的 22 个差异的输出内容，差异占比约为 10.14%。而经过 loop 语句变异的测试用例，差分测试后出现差异的结果升至 41 个，存在差异用例占总用例的 19%，差分测试效率有了明显的提升。

4.2.2 差分测试实验

通过使用本文特色的相关性矩阵，差分模糊测试框架可以选择合适的 JVM 优化选项执行程序。在执行用例时，可以选择-Xcomp 选项显式地开启 JIT 编译器，增加检测出 JIT 优化错误的几率。

还需要注意的是，JIT 包含两个内置编译器：一个称为客户端编译器（C1 编译器），启动速度快、峰值性能低，用于局部优化。所以若某个用例的函数调用场景较多，可利用此类型用例重点测试 C1 编译器。另一个称为服务器编译器（C2 编译器），更关注全局优化和运行时信息。所以有关堆栈空间管理和垃圾回收机制的相关用例，可重点测试 C2 编译器部分。

根据本实验评估指标的要求，将从软件漏洞类型和漏洞发现率两方面对差分实验结果进行分析：

（1）软件漏洞类型：

通过分析差分报告，本节从 JVM 内存溢出、运行时间差异过大、Thread 进程问题、hashcode 值安全性等方面进行对应实验结果分析，探究各漏洞类型触发场景和原因。

A.内存溢出

在原始种子经过 loop 语句变异的条件下，执行以下测试用例 TestSpecTrapClassUnloading、TestDeoptOOM、CountedLoopProblem 的变异种子可以发现：测试用例在 OpenJ9 JVM 上运行出现了程序内存溢出，而在 Zulu JVM 和 HotSpot JVM 上未出现任何异常。这表明 OpenJ9 的内存分配实现与 HotSpot、Zulu 都有所不同，甚至可能其存在更深层次的危险漏洞。图 4-2 展示了用例 CountedLoopProblem 在 OpenJ9 内存溢出。

```
|
运行.class文件完成序号: 219      类名: CountedLoopProblem
错误输出:
line1: JVMDUMP039I Processing dump event "systhrow", detail "java/lang/OutOfMemoryError" at 2
please wait.
line2: JVMDUMP032I JVM requested System dump using '/home/Graduation_Project_Code/Differentia
core.20230427.014120.29484.0001.dmp' in response to an event
line3: JVMPORT030W /proc/sys/kernel/core_pattern setting "|/usr/share/apport/apport %p %s %c
that the core dump is to be piped to an external program. Attempting to rename either core or
line4:
```

图 4-2 CountedLoopProblem 在 OpenJ9 内存溢出

此处选择 CountedLoopProblem.java 程序进行分析。代码块内存在一个 StringBuilder 对象——str，由于程序在相当大的循环次数内一直迭代执行，所以 str 对象会频繁地调用 str.append()函数。一般来说，new StringBuilder()对象都存储在堆，所以 str 大量的数据存入导致了内存的溢出。代码异常内容如代码块 4-1 所示。而对于 HotSpot 和 Zulu 两个版本，由于 JVM 底层实现和数据安全防护都与 OpenJ9 不同，所以暂未出现内存溢出。

代码 4-1 CountedLoopProblem.java 存在异常代码块

```
try {
    StringBuilder str = new StringBuilder();
    //方法已经 for 语句变异
    for (int newIndex1 = 0; newIndex1 < 20; ++newIndex1)
        for (int i = 0; i < 1000000; ++i) {
            int v = Math.abs(r.nextInt());
            str.append('+').append(v).append('\n');
            x += v;
            //方法已经 for 语句变异
            for (int newIndex2 = 0; newIndex2 < 20; ++newIndex2)
                while (x < 0) x += 1000000000;
            str.append('=').append(x).append('\n');
        }
    if (str.toString().hashCode() != 0xaba94591) {
        throw new Exception("Unexpected result");
    }
} catch (OutOfMemoryError e) {e.printStackTrace();}
```

B. Thread 进程执行顺序问题

进程/线程的同步与异步执行问题一直是系统中需要特别关注的工作。而在 Java 语言运行环境中，异步线程的执行顺序通常无法完全确定。根据 Java 基本规约^[22]，Object 对象都包含 finalize()方法，并且一般在实例被垃圾回收器释放时才会调用 finalize()方法。

由差分测试发现，用例 Test7190310 在三种 JVM 机器上的输出都不相同，输出结果如图 4-3 所示。经过大量测试统计，OpenJ9 的输出主要为图 4-3(a)，表现为先执行两个 Thread()内的 run()方法（第 1-3 行操作），最后再执行 Object 实例化对象的 finalize()方法（第 4 行操作）。而 HotSpot 与 Zulu 的主要输出为图 4-

3(b), 表现为先执行 Object 实例化对象的 finalize()方法 (第 1 行), 后执行线程内的 run()方法 (第 2-4 行操作)。目前三个差异输出都已分别提交到对应 JVM 开发团队 Bug issues 目录, 暂时仅收到 OpenJ9 的 issue 回复, 位于 OpenJ9 Github 仓库#17290 号 issue。OpenJ9 开发团队表示: “OpenJ9 是与 Hotspot 不同的实现, Java 规范中不保证这些操作的顺序。在 OpenJ9 中 ReferenceQueue.remove()方法可以在打印 finalization 消息前后任何时候执行”。由于 HotSpot 和 Zulu 都是先执行 finalize()释放方法, 再打印 Thread 线程 run()方法, 与普通程序运行流程不太相符, 所以本文结束后将继续与 HotSpot 和 Zulu 开发团队关于此 issue 进行沟通交流。

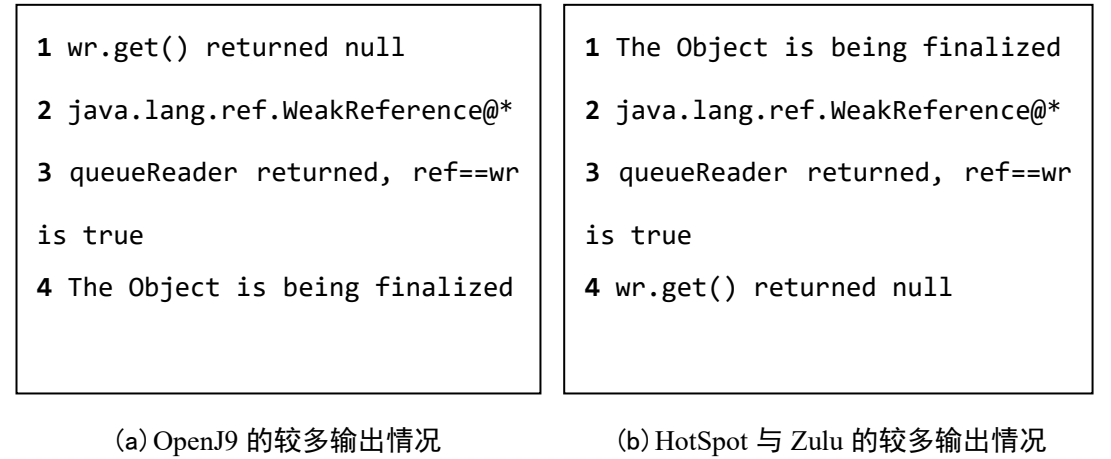


图 4-3 Test7190310 在不同 JVM 上的输出

C.运行时间差异大

程序运行时间是检验 JIT 性能与主机性能的重要参数, 在本文的差分模糊测试工作中出现了同一用例在不同 JVM 上运行时间相差过大的情况。针对这些运行时间差异过大的用例, 本文进行详细分析。通过 for 语句变异的用例 Test6896617 是运行速度差异较大的一个代表情况, 其不同 JVM 上的运行时间统计日志如图 4-4 所示。

HotSpot	<div>运行.class文件完成序号: 171 类名: Test6896617</div> <div>正常输出:</div> <div>line1: Testing different source and destination sizes</div> <div>line2: Testing big char</div> <div>line3: Testing ISO_8859_1\$Encode.encodeArrayLoop() performance</div> <div>line4: size: 256 time: 486</div> <div>line5: Failed 1: ArrayEncoder.encode char[256]</div> <div>line6: Testing ISO_8859_1\$Encode.encode() performance</div> <div>line7: size: 256 time: 336</div> <div>line8: FAILED</div>
---------	--

OpenJ9	运行.class文件完成序号: 171 类名: Test6896617 正常输出: line1: Testing different source and destination sizes line2: Testing big char line3: Testing ISO_8859_1\$Encode.encodeArrayLoop() performance line4: size: 256 time: 4574 line5: Failed 1: ArrayEncoder.encode char[256] line6: Testing ISO_8859_1\$Encode.encode() performance line7: size: 256 time: 1461 line8: FAILED
Zulu	运行.class文件完成序号: 171 类名: Test6896617 正常输出: line1: Testing different source and destination sizes line2: Testing big char line3: Testing ISO_8859_1\$Encode.encodeArrayLoop() performance line4: size: 256 time: 509 line5: Failed 1: ArrayEncoder.encode char[256] line6: Testing ISO_8859_1\$Encode.encode() performance line7: size: 256 time: 316 line8: FAILED
图 4-4 用例 Test6896617 在不同 JVM 上时间统计	

三个结果当中，OpenJ9 的总运行时间最长。OpenJ9 测试 encodeArrayLoop() 模块的耗时接近 HotSpot 的 10 倍，测试 encode() 模块的耗时也是其他 JVM 的 4 倍左右。由此可得，OpenJ9 在多个双层循环的场景下表现不及 HotSpot 等 JVM。

D. Hashcode 值

通过差分测试发现，在 HotSpot 和 Zulu JVM 上执行包含 Object.hashCode() 方法的测试用例 test.java 后，Object 实例化对象 a 的 hash 值 $Hash_{a_now}$ ，与 test.java 文件历史测试中 Object 实例化对象的 a 的 hash 值 $Hash_{a_previous}$ 相等，即 $Hash_{a_now} == Hash_{a_previous}$ 。更通俗来讲，就是对同一个程序中的 Object 类型对象 a，它本次运行程序获得的 hash 值 a_1 与上次运行程序获得的 hash 值 a_2 相同，即 $a_1 == a_2$ 。这是非常不符合程序设计规范的情况，通常 Object 对象 a 会存储在主存堆区，待程序结束执行垃圾回收机制，自动释放实例 a 的资源占用，等到下次运行该程序时再重新为实例 a 分配其他空间，但 HotSpot 和 Zulu 的情况明显违反了这一点。通过大量测试，该情况仅在 HotSpot 和 Zulu JVM 上被触发，OpenJ9 JVM 每次运行时变量的 hash 值会变化。以上差异至少揭示了 HotSpot 和 OpenJ9 在各自的内存管理上是通过不同方式实现的，以及在 JIT 模块的代码优化也存在一些不同实现。

(2) 漏洞发现率：

漏洞发现率，是通过计算本方法可以检测到的漏洞数量与被测软件中存在的漏洞总数的比值^[32]来获得的。本文对三个版本 JVM 的漏洞发现率都进行了统计，旨在验证本文差分模糊测试方法的有效性。

经过 Oracle JDK 官网和部分 CVE 数据库的 Bugs 列表统计，关于 JDK8 的 JIT 漏洞总数多达 138 个。而本文在 JDK8 版本的测试流程中，一共选择了 321 个 jdk8u 测试用例。首先，分析本实验 jdk8u 用例下的所有差分结果后确认，总共用 22 个差异结果输出，它们可归纳为 4 种已确认的潜在漏洞；其次，对 Oracle JDK 官网给出的其他错误报告进行复现测试，42 种已确认的漏洞可以在本文差分模糊测试框架下验证得到。所以本文差分模糊框架对 HotSpot JDK8 JVM 内 JIT 模块的漏洞发现率为 33.3%（46/138）。对其他版本 JVM 也进行相同测试，最终本系统针对三个版本 JVM 漏洞发现率如图 4-5 所示。

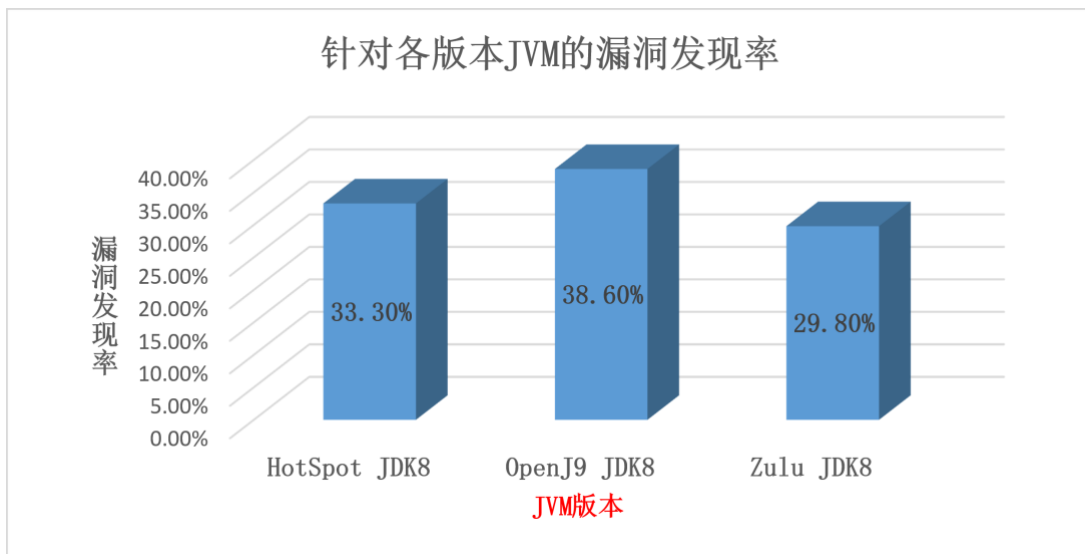


图 4-5 各 JVM 漏洞发现率

4.2.3 对比实验分析

本文的对照实验是“Classming 模糊测试技术”^[1]。通过对比两种测试方法在漏洞检测效率、漏洞类型上的差异，来分析各方法的优劣，最终达到改进本系统

的目的。

首先，简要介绍 Classming 的工作原理：

- Classming 记录种子类文件的活字节码。所谓活字节码，就是程序会执行到的字节码序列。在种子突变层面，唯有对会被执行的代码进行变异，才算是有效变异。
- 它系统地操纵和改变种子内活字节码中的控制流和数据流，以从种子生成语义不同的突变体。
- Classming 用产生的这些突变体对各版本 JVM 进行差分测试，旨在暴露各 JVM 的差异和潜在漏洞。得出实验结果，进行实验分析。

经过流程介绍可以得出，Classming 的主要工作是变异种子测试用例的字节码文件，尤其是改变种子文件的控制流和数据流，以此为基础创建具有不同于语义的突变体。Classming 对照实验最终选择了 8 个基准测试，包括 avrora、batik、eclipse、fop、jython、pmd、sunflow 等内容。测试结果表明，在本实验的 321 个 hotspot jdk8u 测试用例中，13 个测试用例出现了差异输出，同时这些经过变异的测试用例的特征都与数据流/控制流强相关。其中，HotSpot 会强制执行结构化锁定的规则，抛出 IllegalMonitorStateException 异常，另外两个 JVM 不会发生此异常；OpenJ9 不允许监控未初始化的对象，否则会抛出 NullPointerException 异常。

再分析本文所设计的基于“特征-JVM 选项”相关性矩阵的差分模糊测试框架发现，本文框架对用例的变异主要集中在.java 源文件上，而非 Classming 方法在字节码序列层面对种子进行变异。测试结果表明，在本实验的 321 个 hotspot jdk8u 测试用例中，22 个测试用例出现了不同 JVM 的差异输出。而这些有差异的测试用例代码特征不局限于数据流等，都比较分散。因此，相比 Classming，采用此差分测试框架对 JVM 内 JIT 模块的测试覆盖面也更全面，也更会利用执行命令中 JVM 优化选项来针对性的测试 JIT 模块相应漏洞。

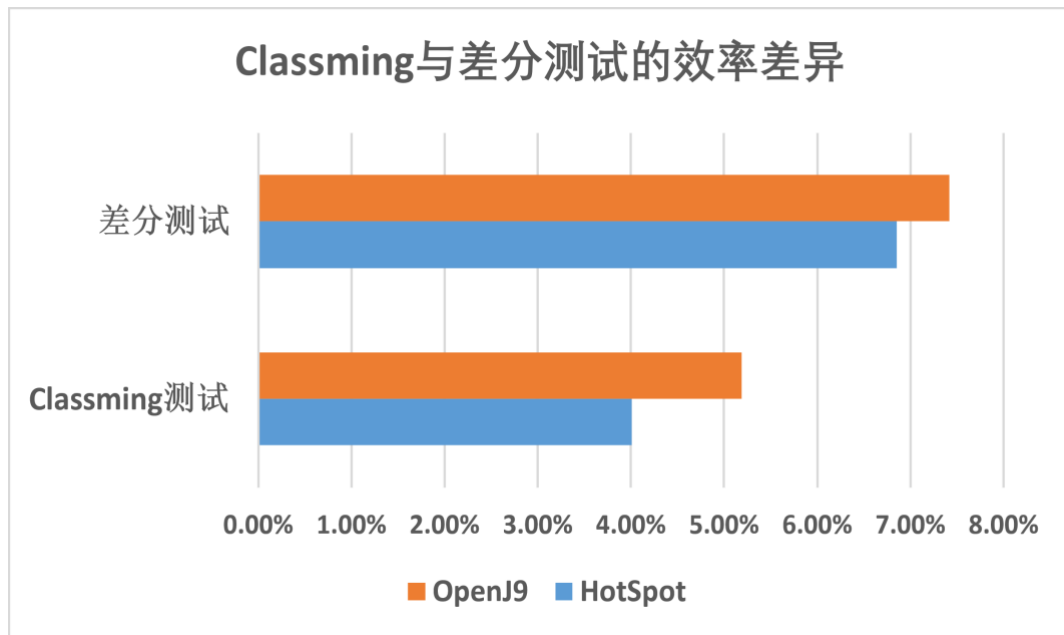


图 4-6 Classming 与差分测试的效率差异

本章截至目前为止，已经对实验设置和差分模糊测试实验中获得的有价值结果进行详细分析，提出了各种漏洞类型和可能的触发原因，达到了实验分析的目的。同时进行指标评估和对比实验，展现了工作的可靠性与高效性，突出了本文所设计差分框架的优势。

5 结论与展望

5.1 研究结论总结

Java 虚拟机（JVM）是目前软件工程领域一种常用的虚拟机，它是 Java 应用程序能正常开发、生产和发行的关键基础部件。然而随着软件复杂度和功能性的不断提升，业界对 JVM 的可靠性和安全性的要求也在快速提升。作为 JVM 中最重要的动态编译模块，JIT（Just-In-Time）编译器负担着越来越多代码优化、程序性能提升方面的工作，同时关于 JIT 的 Bug 数量和严重程度也在逐年上升。为了保证 JIT 编译器能更高效、正确的翻译代码与执行代码优化，本文提出了一种基于差分模糊测试的 JIT 测试框架，它主要创新地包含种子突变程序和“代码特性-JVM 配置”相关性模块，以求最大限度的指导种子变异与更高效的测试出 JIT 的相关潜在漏洞。

本文主要包括以下几个方面的研究工作：

（1）本文通过对 JVM 及 JIT 运行原理进行研究，得到 JIT 优化技术方法和触发 JIT 优化的一般场景。在此基础上，具有针对性、创新性地设计或变异测试用例，旨在更高效、更深层次的测试到 JIT 的相关问题。本文提出了两种变异算子和一个变异算法：两种变异算子主要针对 if 语句块变异和 loop 语句块变异；而变异算法则是通过 javaparser 自动解析初始种子的代码特性，然后根据设置的变异迭代次数进行突变，突变过程中搭配设计好的两种变异算子即可。突变后的种子可以增加测试的执行深度，提升检测效率。

（2）本文对基本的差分模糊测试框架进行学习，分析其优势与不足。最终根据差分模糊测试技术的特点，独创性地设计出基于差分模糊测试技术的 JIT 测试方法，并独立进行差分模糊测试系统的代码实现。与此同时，在差分模糊测试框架基础上引入“测试用例代码特性与 JVM 优化配置”相关性矩阵的思想，旨在通过输入特定的 JVM 优化选项（例如-Xcomp）来更高效、更具有针对性的测试出 JIT 相关缺陷。使用这种相关性矩阵的方法，还有助于通过历史测试数据训练相关性矩阵模型，提升下一次的 JIT 测试效率。

（3）本文获取差分模糊系统得到的差分报告，对存在差异输出的测试用例

进行分析，并确认其是否存在潜在漏洞。最后讨论 JVM 底层实现的安全性和稳定性，以及漏洞发现率、漏洞类型等指标，给出简单建议。

5.2 存在的问题与展望

相较于普通的差分测试框架，本文提出的基于差分模糊测试的 JIT 检测技术测试效率更高、测试稳定性更强。其内含的“测试用例代码特性与 JVM 优化配置”相关性矩阵模块具有一定的创新性，对深度测试 JIT 模块有较大的提升。然而，本方法还是存在一定的改进之处，主要包含以下几个方面：

（1）测试用例的变异是提升测试效率非常关键的一步。由于本文的重点在差分测试方面，所以本文系统框架的测试用例突变程序仅包含面向 if 语句和 loop 语句的突变，相比起专业的种子突变项目略显单调。甚至在某段测试时期，也可以会因为种子测试用例的变异不充分而导致无法测出 JIT 漏洞或缺陷。基于此，本文应该在后续研究中丰富变异算子和变异算法，提升测试用例质量。

（2）由于“测试用例代码特性与 JVM 优化配置”相关性矩阵模块的引入，不可避免的会导致问题的搜索空间爆炸增长。假设从 JVM 配置角度来看，每个布尔选项都有 0/1 两个值，其他整形选项的可选范围更大。那么将 JVM 配置与用例代码特性进行笛卡尔积连接，也会产生非常多的数据库表项，可能会导致测试效率低下。所以在此基础上，本文希望增加相关性矩阵更新算法，当某些占比很小的组合表项长时间没有进行更新，那么就抛弃这种影响不大的“特性——配置”组合，以此减少问题搜索空间的复杂度；同时对整型配置选项，进行区间值限定，一个组合表项就代表该配置下某值的一段区间，如此操作可以也可以减少问题的复杂程度，提升速率。

参考文献

- [1] Chen Y, Su T, Su Z. Deep differential testing of JVM implementations: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)[C], 2019. IEEE.
- [2] Chen C, Cui B, Ma J, et al. A systematic review of fuzzing techniques[J]. Computers & Security, 2018,75:118-137.
- [3] McKeeman W M. Differential testing for software[J]. Digital Technical Journal, 1998,10(1):100-107.
- [4] Chen J, Patra J, Pradel M, et al. A survey of compiler testing[J]. ACM Computing Surveys (CSUR), 2020,53(1):1-36.
- [5] Jia H, Wen M, Xie Z, et al. Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces[J].
- [6] Suganuma T, Ogasawara T, Takeuchi M, et al. Overview of the IBM Java just-in-time compiler[J]. IBM systems Journal, 2000,39(1):175-193.
- [7] Böhme M, Cadar C, Roychoudhury A. Fuzzing: Challenges and reflections[J]. IEEE Software, 2020,38(3):79-86.
- [8] 冯兆文, 刘振慧. 开源软件漏洞安全风险分析[J]. 保密科学技术, 2020(02):27-32.
- [9] 李舟军, 张俊贤, 廖湘科, 等. 软件安全漏洞检测技术[J]. 计算机学报, 2015,38(04):717-732.
- [10] 吴世忠, 郭涛, 董国伟, 等. 软件漏洞分析技术进展[J]. 清华大学学报(自然科学版), 2012,52(10):1309-1319.
- [11] 张雄, 李舟军. 模糊测试技术研究综述[J]. 计算机科学, 2016,43(05):1-8.
- [12] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990,33(12):32-44.
- [13] 李伟明, 张爱芳, 刘建财, 等. 网络协议的自动化模糊测试漏洞挖掘方法[J]. 计算机学报, 2011,34(02):242-255.
- [14] Böhme M, Pham V, Roychoudhury A. Coverage-based greybox fuzzing as markov chain: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security[C], 2016.
- [15] 李唯. 基于动态符号执行和静态分析的Fuzzing测试算法研究[D]. 北京邮电大学, 2020.
- [16] 谢肖飞, 李晓红, 陈翔, 等. 基于符号执行与模糊测试的混合测试方法[J]. 软件学报, 2019,30(10):3071-3089.
- [17] 苗媛. 模糊测试用例的生成方法研究与应用[D]. 内蒙古大学, 2022.
- [18] 高凤娟, 王豫, 司徒凌云, 等. 基于深度学习的混合模糊测试方法[J]. 软件学报, 2021,32(4):988-1005.
- [19] Zhao Y, Wang Z, Chen J, et al. History-driven test program synthesis for JVM testing: Proceedings of the 44th International Conference on Software Engineering[C], 2022.
- [20] Nilizadeh S, Noller Y, Pasareanu C S. Diffuzz: differential fuzzing for side-channel analysis: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)[C], 2019. IEEE.
- [21] Brennan T, Saha S, Bultan T. Jvm fuzzing for jit-induced side-channel detection: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering[C], 2020.
- [22] 周志明. 深入理解 Java 虚拟机: JVM 高级特性与最佳实践[M]. 机械工业出版社, 2020.

-
- [23] Wu M, Lu M, Cui H, et al. JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers: Proceedings of the 45th International Conference on Software Engineering, ser. ICSE[C].
- [24] Stepanian L, Brown A D, Kielstra A, et al. Inlining Java native calls at runtime: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments[C], 2005.
- [25] Würthinger T, Wimmer C, Mössenböck H. Array bounds check elimination for the Java HotSpot™ client compiler: Proceedings of the 5th international symposium on Principles and practice of programming in Java[C], 2007.
- [26] Gulzar M A, Zhu Y, Han X. Perception and practices of differential testing: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)[C], 2019. IEEE.
- [27] Prokopec A, Duboscq G, Leopoldseder D, et al. An optimization-driven incremental inline substitution algorithm for just-in-time compilers: 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)[C], 2019. IEEE.
- [28] 练坤梅, 许静, 田伟, 等. SQL注入漏洞多等级检测方法研究[J]. 计算机科学与探索, 2011,5(05):474-480.
- [29] Chen P, Fang Y, Mao B, et al. JITDefender: A defense against JIT spraying attacks: Future Challenges in Security and Privacy for Academia and Industry: 26th IFIP TC 11 International Information Security Conference, SEC 2011, Lucerne, Switzerland, June 7-9, 2011. Proceedings 26[C], 2011. Springer.
- [30] Antunes N, Vieira M. On the metrics for benchmarking vulnerability detection tools: 2015 45th Annual IEEE/IFIP international conference on dependable systems and networks[C], 2015. IEEE.
- [31] Votipka D, Stevens R, Redmiles E, et al. Hackers vs. testers: A comparison of software vulnerability discovery processes: 2018 IEEE Symposium on Security and Privacy (SP)[C], 2018. IEEE.
- [32] 唐成华, 田吉龙, 王璐, 等. 一种基于软集和多属性综合的软件漏洞发现方法[J]. 计算机科学, 2015,42(05):183-187.