分类号：TP391　　学校代码：10697

密　级：公开　　学　　号：201831956

西北大学
Northwest University

硕士专业学位论文

Dissertation for the Professional Degree of Master

面向嵌入式 JavaScript 引擎的差分模糊测试方法研究

学科名称：电子与通信工程

专业学位类别：工程硕士

作者：姚厚友　　指导老师：牛进平　副教授

汤战勇　教　授

西北大学学位评定委员会

二〇二一年

Research on Differential Fuzzing Testing for Embedded JavaScript Engine

A thesis submitted to

Northwest University

in partial fulfillment of the requirements

for the degree of Master

in Electronic and Communication Engineering

By

Yao Houyou

Supervisor: Niu Jinping Associate Professor

Tang Zhanyong Professor

June 2021

西北大学学位论文知识产权声明书

本人完全了解西北大学关于收集、保存、使用学位论文的规定。学校有权保留并向国家有关部门或机构送交论文的复印件和电子版。本人允许论文被查阅和借阅。本人授权西北大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。同时授权中国科学技术信息研究所等机构将本学位论文收录到《中国学位论文全文数据库》或其它相关数据库。

保密论文待解密后适用本声明。

学位论文作者签名：姚厚友 指导教师签名：王晓

2021 年 6月 8日 2021年 6月 8日

西北大学学位论文独创性声明

本人声明：所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，本论文不包含其他人已经发表或撰写过的研究成果，也不包含为获得西北大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。学位论文作者签名：姚厚友

2021年 6月 8日

摘要

JavaScript 凭借其动态、交互、跨平台等特性，成为嵌入式物联网设备广泛使用的编程语言。为了在物联网设备上解释执行 JavaScript 程序，各类嵌入式 JavaScript 引擎被大量开发。然而，由于编程人员水平参差不齐，对 JavaScript 引擎设计规范理解偏差，导致开发的 JavaScript 引擎存在安全缺陷、功能缺陷和性能缺陷等问题。安全缺陷会使嵌入式设备面临安全风险，功能缺陷会影响 JavaScript 程序的正确运行，嵌入式 JavaScript 引擎的性能缺陷不仅会白白消耗有限的计算资源，还会造成低功耗嵌入式设备的能量浪费。

黑盒差分模糊测试是目前发现 JavaScript 引擎上述缺陷的主要途径，但现有的差分模糊测试方法存在两个方面的问题：一方面，差分模糊测试主要针对安全缺陷和功能缺陷进行检测，而忽略了性能缺陷检测；另一方面，差分模糊测试结果中复杂的测试用例和大量重复的测试结果增加了手动分析测试结果的成本。针对上述问题，本文研究内容如下：

（1）本文提出了一种以性能缺陷检测为导向的差分模糊测试方法，该方法旨在发现更多的 JavaScript 引擎性能缺陷。为了实现该目标，本文对用例生成质量和检测方法设计进行改进。在保证测试用例生成质量方面，首先从开源代码库中提取语法正确且语义相对完整的 JavaScript 函数，然后调用提取的函数并传递参数得到测试用例，最后对测试用例进行抽象语法树级别的变异使其具备性能缺陷检测的能力。在缺陷检测方法设计方面，改进当前的差分模糊测试方法，使其能够捕获测试用例所触发的性能缺陷。

（2）本文提出了基于抽象语法树的测试用例精简方法和基于多维特征的测试结果过滤方法，用于简化测试用例的复杂度，减少测试结果的重复率。测试用例精简使复杂的测试用例变得简单，便于快速理解缺陷。测试结果过滤会将重复的测试结果删除，避免分析重复的测试结果造成人力资源浪费。

（3）设计并实现了 JSDiff 原型系统，并详细介绍了系统模块划分和关键算法设计，同时从多个角度对 JSDiff 系统进行实验评估，JSDiff 系统总共发现 71 个缺陷，其中确认 43 个。实验结果表明，本文提出的方法不仅具备良好的性能缺陷、功能缺陷及安全缺陷的检测能力，而且也能有效地降低测试结果的人力分析成本。

关键词：嵌入式 JavaScript 引擎，差分模糊测试，性能缺陷，缺陷检测

ABSTRACT

With its dynamic, interactive, and cross-platform features, JavaScript has become a programming language widely used in embedded IoT devices. Various embedded JavaScript engines have been developed in large numbers to interpret and execute JavaScript programs on IoT devices. However, due to the varying level of programmers and deviations in understanding JavaScript engine design specifications, the developed JavaScript engine has security defects, functional defects, and performance defects. Security defects will expose embedded devices to security risks, and functional defects will affect JavaScript programs' correct operation. The performance defects of embedded JavaScript engines will consume limited computing resources in vain and cause energy waste in low-power embedded devices.

Black box differential fuzzing testing is currently the primary way to find the defects mentioned above of JavaScript engines, but the existing differential fuzzing testing method has two problems. On the one hand, differential fuzzing testing mainly focuses on security defects and functional defects but ignores performance defects. On the other hand, the complicated test cases and many repeated test results in the differential fuzzing test results increase the cost of manually analyzing the test results. In response to the above problems, the research content of this article is as follows:

(1) This thesis proposes a performance defect detection-oriented differential fuzzing testing method to find more JavaScript engine performance defects. This thesis improves the quality of generated test cases and the detection method design to achieve this goal. In terms of ensuring the quality of generated test cases, we first extract JavaScript functions with correct syntax and relatively complete semantics from the open-source code library. Then we call the extracted functions and pass parameters to get test cases. Finally, we mutate test cases on the level of the abstract syntax tree to make it have the ability to detect performance defects. In terms of defect detection method design, we improve the current differential fuzzing testing method to capture the performance defects triggered by test cases.

(2) This thesis proposes a test case simplification method based on the abstract syntax tree and a test result filter method based on multi-dimensional features, which are used to simplify the complex test cases and reduce the repetition rate of test results. Test case

simplification makes complex test cases simple and facilitates a quick understanding of defects. The test result filtering will delete the repeated test results to avoid wasting human resources caused by analyzing the repeated test results.

(3) We designed and implemented the JSDiff prototype system and introduce the system module division and primary algorithm design in detail. At the same time, we evaluate the JSDiff system from multiple angles. And JSDiff system found 71 defects in total, of which 43 confirmed. The experimental results show that the method proposed in this thesis not only has good defect detection ability on performance defect, functional defect, and safety defect, but also can effectively reduce the cost of manually analyzing test results.

Keywords: Embedded JavaScript engine, Differential fuzzing testing, Performance defect, Defect detection

目录

第一章 引言

1.1 研究背景与意义

随着物联网技术的发展，万物互联已经是必然的趋势。为了将物联网设备顺利接入互联网，极具网络编程优势的 JavaScript 成为嵌入式开发的首选编程语言，因此也出现了许多优秀的嵌入式 JavaScript 引擎。JavaScript 引擎负责解释并执行 JavaScript 语言编写的程序，其功能的正确性决定了 JavaScript 程序是否能正确运行。JavaScript 引擎是执行 JavaScript 程序必不可少的软件，JavaScript 引擎的特殊性引起了学术界对其安全问题的广泛关注[1,2]。

嵌入式 JavaScript 引擎[3]除了需要关注功能正确性和安全性以外，还需要关注引擎性能的高效性。在计算能力强且电源供应便捷的服务器或者 PC 上，软件的性能缺陷可以用硬件优势予以弥补。然而，嵌入式 JavaScript 引擎运行在计算能力较弱且要求低功耗的设备上，使得嵌入式 JavaScript 引擎在实现时使用的算法需要具备高效的特点。与桌面 JavaScript 引擎不同，嵌入式 JavaScript 引擎运行在嵌入式的设备上使其无法使用即时编译技术，没有即时编译技术加持使得嵌入式 JavaScript 引擎的性能问题会更严重。嵌入式 JavaScript 引擎的性能缺陷会使程序的执行时间变长，增加嵌入式设备的能耗，使得嵌入式 JavaScript 引擎的性能缺陷检测变得更迫切。性能缺陷通常具有隐蔽性强，影响时间长，修复缓慢的特点。其主要原因有两点：其一，引擎开发过程中程序员会更关注功能的正确性，性能问题通常没有得到足够的重视；其二，与功能测试不同，性能测试是通过判断其性能是否在可接受范围之内决定测试是否通过，但是否可接受的定义具有较强的主观性。

差分模糊测试是一种软件缺陷自动检测技术，被广泛应用于编译器和解释器的自动化测试。其主要用于检测 JavaScript 引擎的安全缺陷和功能性缺陷，目前为止尚未有研究学者使用差分模糊测试对性能缺陷进行检测。经过对现有工作的分析发现，差分模糊测试难以检测性能缺陷的原因主要有三点。首先，测试性能缺陷的测试用例需要有具备语法正确和语义丰富两个条件。语法正确的测试用例是检测嵌入式 JavaScript 引擎性能缺陷的基本条件，语义丰富的测试用例才有可能触发嵌入式 JavaScript 引擎的性能缺陷。其次，测试用例中语句的执行时间与计时器精度不匹配，语句执行次数太少未能暴露出引擎性能缺陷等问题，使得测试用例即使覆盖了存在性

能问题的缺陷代码也难以使用自动化测试技术检测其性能缺陷。这是由于 JavaScript 除了诸如正则匹配之类的少数操作的执行时间可能会比较长以外，大部操作的执行时间都远小于 1ms，然而计时器的精度大于 1ms 使得计时器无法估量单个操作的执行时间。最后，如何自动客观地指定性能评价指标是具有挑战且有必要解决的难题。正因为这三个问题的存在，使得差分模糊测试没有被用于软件的性能缺陷检测。

针对差分模糊测试无法检测性能缺陷的问题，本文提出了以性能缺陷检测为导向的差分模糊测试方法。首先利用 GitHub 开源仓库作为语料库并自动生成语法正确且语义丰富的测试用例，然后设计抽象语法树级别的测试用例变异方法改变测试用例中语句的执行次数以解决测试用例性能缺陷检测能力不足的问题，最后改进差分模糊测试方法使其具备检测性能缺陷的能力。通过这种方式不仅可以检测嵌入式 JavaScript 引擎的性能缺陷，还能检测嵌入式 JavaScript 引擎的安全缺陷和功能缺陷。

使用差分模糊测试技术测试嵌入式 JavaScript 引擎的缺陷时，随机获取的测试用例能测试出大量的测试结果，这些测试结果需要人工分析并确认其是否触发了引擎的未知缺陷，然而人工分析测试结果的成本非常高。测试结果分析成本高的原因主要来源于两个方面。

一方面，随机获取的测试用例功能复杂，若经过差分模糊测试后触发了可疑缺陷，需要将功能复杂的测试用例简化为程序员容易理解的简单用例，便于确认测试结果是否触发了未知的缺陷。人工精简测试用例的过程非常缓慢且需要专业的测试人员，对大量的测试用例进行手动精简会极大地提高测试成本。为了解决手动精简测试用例效率低且成本高的问题，Mozilla 公司开发了一款基于代码行的测试用例精简工具 Lithium[4]。然而，由于没有充分考虑 JavaScript 语言的特点，在对具有块结构特征的 JavaScript 测试用例进行精简时 Lithium 存在效率低和精简不彻底等问题。

另一方面，随机获取的测试用例会重复触发大量已经发现过的测试结果，大量重复的测试结果增加了手动分析测试结果的成本。为了解决测试结果重复的问题，Groce A[5]等人在对任务文件系统进行测试时采取了手动调整规则的方式实现了测试结果过滤。为了解决手动调整过滤规则效率低的问题，Chen Y[6]等人提出了对触发缺陷的测试用例进行排序的方法，将有限的人力资源集中于分析排名靠前的测试用例触发的测试结果以提高测试结果的分析效率。对测试用例进行排序的方法虽然缓解了测试结果重复的问题，但其仍然没能从根本上解决问题。

针对导致测试结果分析成本高的两个问题，本文分别提出了基于抽象语法树的测

试用例精简方法和基于多维特征的测试结果过滤方法，测试用例精简解决了已有方法存在的测试用例精简效率低、精简不彻底及精简出错的问题，测试结果过滤方法能将重复的测试结果进行精准过滤。通过测试用例精简和测试结果过滤可以极大地降低测试结果的分析成本。

1.2 国内外研究现状

1.2.1 软件安全缺陷检测研究现状

安全缺陷是软件缺陷中最严重的问题之一，尤其是编译器等特殊软件的安全缺陷，因此编译器安全缺陷检测方法也是当前缺陷检测工作中的研究重点。保证复杂编译器系统的安全性是一项具有挑战性的任务，模糊测试是当前检测编译器安全缺陷的主流方法。模糊测试是通过检测软件崩溃，并根据软件崩溃是否可用于安全攻击而决定其是否是安全缺陷。下面将介绍几种最新的利用模糊测试技术检测安全缺陷的方法。

Groß S[7]提出了一种字节码级别的测试用例变异方法 Fuzzilli。通过自定义一种字节码级别的中间语言 FuzzIL，并在此中间代码上设计变异规则对测试用例进行变异以改变程序的数据流和控制流，实现抽象语法树级别难以完成的变异，达到检测 JavaScript 引擎安全缺陷的目的。

Han H S[8]等人是提出了一种语义感知的测试用例生成方法 CodeAlchemist。通过将现有的 JavaScript 代码拆分成较小的代码片段，并利用这些代码片段组装成语法正确且具有语义信息的测试用例，把这些随机组装的测试用例作为测试的输入，利用模糊测试技术对 JavaScript 引擎的安全缺陷进行检测。

Lee S[9]等人提出一种基于机器学习模型的缺陷检测方法 Montage。将 JavaScript 代码的抽象语法树的子树作为训练语言模型的语料库，利用深度学习算法 LSTM 学习 JavaScript 代码抽象语法树的结构特征并利用其实现了测试用例的突变，然后使用训练后的网络模型自动生成测试用例，用于检测 JavaScript 引擎的安全缺陷。

Park S[10]等人提出了一种基于变异的测试用例生成方法 DIE。将已检测出缺陷的测试用例或单元测试的测试用例作为变异种子，在保留种子原有结构和数据类型的前提下在抽象语法树级别对测试用例进行变异，并利用变异后的测试用例检测 JavaScript 引擎安全缺陷。

1.2.2 软件功能缺陷检测研究现状

编译功能的正确性决定了运行在编译器上的程序是否能正确运行，保证编译器功

能的正确性至关重要。但编译器是一个复杂的软件系统，存在转换空间大等特点，如test262[11]等语言规范提供的官方测试套件都难以保证软件的正确性。为了弥补语言规范提供的测试套件不完善的问题，当前主流的解决方案是使用差分测试技术。差分测试在编译器和运行时系统测试方面成绩斐然，弥补了测试套件不完善的问题，差分测试被成功用于 C/C++，Java[12]等语言的编译器或解释器测试中。下面将介绍几种典型的差分测试方法。

Yang X[13]等人提出手动构建测试用例生成模型的缺陷检测方法 CSmith。此方法通过手动构建的随机生成模型探索 C 语言的非常规组合并得到语法正确的 C 代码作为测试用例，然后使用差分测试技术判断引擎的执行结果中是否存在编译器的功能缺陷。

Cummins C[14]等人利用 OpenCL 源码作为训练数据，使用深度学习技术训练一个具有自动生成测试用例能力的语言模型，通过该语言模型可以不间断地随机生成测试用例。借助语言模型不仅实现了测试用例的自动生成还实现了测试用例的变异。最后使用差分测试技术判断测试用例是否触发了软件缺陷，其方法共计发现了 OpenCL 编译器的 67 个缺陷。

1.2.3 软件性能缺陷检测研究现状

算法低效或编码错误导致的性能缺陷通常会导致软件的性能损失[15]，资源浪费，功耗增加等问题。性能测试在所有的应用和系统中都非常重要，尤其在医疗，通讯，汽车，飞行控制，国防等领域。性能缺陷曾经导致了数亿美元的项目被迫停止[16]，因此性能缺陷检测仍然是一个严峻的问题。修复性能缺陷存在修复成本高，可能引入新的缺陷，增加代码复杂性，周期长等特点[17,18]。下面将对几个典型的性能缺陷检测相关的研究展开介绍。

Jin G[19]等人通过对分布在 5 个开源软件套件中的 109 个真实存在的性能缺陷进行分析并总结，利用总结的性能缺陷指导开发人员如何避免和修复性能缺陷，同时指导测试人员如何检测性能缺陷。根据总结的性能缺陷的特征手动设计了检测性能缺陷的规则，手动设计的规则在真实的测试环境中取得了良好的效果。

Nistor A[20]等人针对智能手机上进行大规模输入并检测软件性能缺陷困难的问题，提出了一种在小规模输入的情况下如何预测大规模输入时可能存在的性能缺陷的方法。使用该方法对智能手机上的 5 个应用程序进行测试，实验结果表明此方法能在小规模输入的情况下检测智能手机上的软件性能缺陷。

Nistor A[21]等人针对性能缺陷修复可能存在入侵性的问题，提出了一种静态检测和修复非入侵性性能缺陷的方法，其检测出的性能缺陷更容易被开发人员接受和修复。具体方法是针对一种特殊的性能缺陷进行检测，这种性能缺陷与一个循环和控制条件相关联。一旦条件满足，执行循环中的其他语句都将失去意义且浪费资源，这种缺陷的修复方法也非常简单，只需要在条件满足时跳出循环即可，修复此类缺陷不会引入新的缺陷即不具有入侵性。

1.3 本文研究内容

本文通过对嵌入式 JavaScript 引擎和差分模糊测试技术的深入研究，提出了一种面向嵌入式 JavaScript 引擎的差分模糊测试方法。通过对获取的测试用例进行变异得到可用于测试嵌入式 JavaScript 引擎性能缺陷的测试用例，使用具有性能缺陷检测能力的差分模糊测试方法判断测试用例是否触发了引擎的缺陷，并对差分模糊测试得到的可疑测试结果进行测试用例自动精简和测试结果自动去重。

本文主要针对以下几个方面进行研究：

（1）研究软件的缺陷检测方法

对已有软件测试技术进行研究，分析不同软件测试技术的优缺点。主要针对当前编译器缺陷自动检测方法中主流的差分模糊测试方法进行深入研究，分析从测试用例获取，测试方法设计，到测试结果处理的流程中存在的问题。从编译器的缺陷检测能力和测试结果的处理能力两方面，对比研究差分模糊测试方法和测试结果自动化处理方法。

（2）研究以性能缺陷检测为导向的差分模糊测试方法

通过对嵌入式 JavaScript 引擎和已有缺陷检测方法的研究，本文从开源的 GitHub 仓库中提取语义相对完整的函数得到函数库，调用函数库中的函数并随机传递相应数据类型的参数获得高质量的测试用例，并在抽象语法树上对测试用例进行变异提升测试用例的性能缺陷检测能力。最后，使用改进后的差分模糊测试方法捕获测试用例触发的缺陷。通过测试用例生成，测试用例变异和改进后的差分模糊测试方法，本文方法不仅能检测嵌入式 JavaScript 引擎的性能缺陷，还能检测嵌入式 JavaScript 引擎的功能缺陷和安全缺陷。

（3）研究以高精度为导向的用例精简与结果过滤方法

差分模糊测试方法检测出的可能触发引擎缺陷的测试结果中存在测试用例复杂

和测试结果大量重复两个问题，复杂的测试用例和大量重复的测试结果会极大地增加手动分析测试结果的成本。为此，提出了基于抽象语法树的测试用例精简方法，实现准确、快速、彻底地测试用例自动精简，便于快速理解测试结果。同时，提出了基于多维特征的测试结果过滤方法实现测试结果的自动过滤，避免分析重复测试结果造成人力资源浪费。通过测试用例精简和测试结果过滤可以极大地降低测试结果的分析成本。

（4）设计并实现 JSDiff 原型系统

基于面向嵌入式 JavaScript 引擎的差分模糊测试方法设计并实现了 JSDiff 原型系统。使用 JSDiff 原型系统对嵌入式 JavaScript 引擎进行测试，并与相关缺陷检测方法进行比较，评估本文方法的缺陷检测能力。同时设计实验并验证测试用例精简与测试结果过滤方法的有效性。

1.4 本文组织结构

本文将划分为五个章节对嵌入式 JavaScript 引擎的差分模糊测试方法展开介绍。详细的组织结构如下：

第一章介绍了嵌入式 JavaScript 引擎并引出嵌入式 JavaScript 引擎缺陷检测的重要性，通过嵌入式 JavaScript 引擎测试重要性及其测试存在的问题阐述本文的背景与意义。然后介绍了安全缺陷、功能缺陷和性能缺陷检测的国内外研究现状。最后介绍了研究内容和组织结构。

第二章描述了本文相关的理论与技术。首先介绍了本文的测试对象，常用的软件测试方法和原理，以及不同软件测试方法的应用场景。最后介绍了差分模糊测试结果中测试用例复杂和测试结果重复的相关解决方案。

第三章阐述了本文设计的以性能缺陷检测为导向的差分模糊测试方法。具体分为测试用例生成和差分模糊测试两个方面，前者介绍了测试用例生成方法，该方法生成的测试用例具有性能缺陷检测的能力，后者介绍了判断测试用例是否触发引擎缺陷的具体实现。

第四章介绍了本文设计的以高精度为导向的用例精简与结果过滤方法。具体包括测试用例精简和测试结果过滤两部分，测试用例精简主要是删除触发缺陷的复杂测试用例中的无关代码，测试结果过滤则是在精简用例的基础之上过滤重复的测试结果。

第五章介绍了本文设计的 JSDiff 原型系统的实现和针对原型系统的实验评估。

JSDiff 原型系统实现部分将从系统模块设计，系统界面两个方面展开介绍。实验结果评估部分将对本文提出的差分模糊测试方法和测试用例精简与测试结果过滤方法进行评估和分析。

第二章 相关理论与技术

本章介绍了相关的理论与技术。首先介绍了嵌入式 JavaScript 引擎及其测试。其次，介绍了常用的软件测试方法及其使用场景。最后，介绍了目前针对差分模糊测试结果存在测试用例复杂和测试结果大量重复问题所提出的解决方案。

2.1 JavaScript 及嵌入式 JavaScript 引擎简介

JavaScript 凭借其使用简单，交互，跨平台等特性在 Web 程序设计方面独占鳌头，网络编程的优势及物联网设备接入互联网的需求让其成为在嵌入式设备上广泛使用的编程语言。本节将从 JavaScript 语言，嵌入式 JavaScript 引擎及其测试三个方面展开介绍。

2.1.1 JavaScript 简介

JavaScript 连续八年被评为最大开源社区 Stack Overflow 上最受程序员欢迎的编程语言， 2020 年就有 69.7%的专业开发人员使用 JavaScript[22]。在最大的代码托管平台 GitHub 上，JavaScript 也是最受欢迎的编程语言[23]。JavaScript 的语言规范是 ECMAScript-262，简称为 ES 规范，在 2015 年以前 ES 规范的版本更新时间不确定，其命名方式是以 ES 加版本号的方式进行的。如 ES4 表示 ECMAScript-262 规范的第 4 版。从 2015 年开始每年都会定期发布一个新的正式版本，与此同时，ES 规范的版本号还使用了 ES 加年份的方式命名方法。2015 年是 ES 规范发布的第 6 个版本，这个版本也存在 ES2015 和 ES6 两种名称。以此类推，2016 年发布的 ES 规范也称为 ES2016 或 ES7。目前正式发布的 ES 规范已经更新到了 ES2020。

2.1.2 嵌入式 JavaScript 引擎

随着物联网的发展，嵌入式开发的需求激增，为了充分利用 JavaScript 开发社区的优势，嵌入式设备上使用的编程语言逐渐实现了从 C/C++到 JavaScript 的转换。JavaScript 语言能完美契合于嵌入式设备上的应用开发[24]。首先，JavaScript 与其运行的硬件平台无关，使用 JavaScript 进行开发很容易实现应用的跨平台，减少嵌入式应用的开发成本。其次，物联网设备接入互联网是一种必然的趋势，而 JavaScript 是网络编程的首选语言，在嵌入式设备上使用 JavaScript 进行网络编程具有非常大的优势。再次，嵌入式设备要求低功耗，基于事件驱动的编程能有效降低功耗，而基于事件驱

动编程的 JavaScript 满足了嵌入式软件开发的要求。最后，庞大的 JavaScript 开发社区缓解了嵌入式开发人员紧缺的问题，同时也极大的降低了嵌入式开发的门槛。

嵌入式设备是为了实现特定功能而对硬件进行灵活裁剪的小型计算机，相比于传统的计算机，嵌入式设备具有成本低，计算能力弱，内存和闪存容量小，要求低功耗等特点。为了让 JavaScript 能在嵌入式设备上运行，需要在嵌入式设备上配备相应的 JavaScript 解释器或 JavaScript 编译器，简称为 JavaScript 引擎。嵌入设备闪存容量小的特点要求运行在嵌入式设备上的 JavaScript 引擎的体积紧凑，JavaScript 引擎运行时对象需要进行特殊的优化才能确保引擎能在嵌入式设备上稳定地运行。供电条件的限制使得嵌入式 JavaScript 引擎需要具备低功耗的特点。嵌入式设备资源受限的特点使得传统的运行在 Web 浏览器上的桌面 JavaScript 引擎无法适用于嵌入式设备，由此也诞生了许多非常优秀的嵌入式 JavaScript 引擎。例如，三星开源的 JerryScript[25]，它是一款能运行在内存小于 64K 的设备上的超轻量级 JavaScript 引擎，JerryScript 不仅被用于三星的物联网平台 IOT.js 和华为开源的鸿蒙系统，还被 Fitbit 公司成功的应用在其智能手表 Ionic 上。XS[26]是一款用于嵌入式 JavaScript 应用开发平台 Moddable 上的 JavaScript 引擎，被广泛用于索尼的相机和电子书阅读器，惠普的打印机，以及东芝和三星的手机等产品上[27]。Duktape[28]是一款专注于可移植性，代码和数据内存占用小的嵌入式 JavaScript 引擎，被应用于 Zabbix 等软件中。

2.1.3 嵌入式 JavaScript 引擎测试

由于嵌入式设备被广泛应用于生活的每个角落，因此嵌入式设备的安全性问题一直是学术研究的热点话题[29,30]。嵌入式设备资源限制，嵌入式操作系统通常缺乏像桌面操作系统完善的安全机制[31]，与运行在桌面操作系统上的软件相比，嵌入式设备上的软件安全漏洞更容易被利用。嵌入式 JavaScript 引擎是运行在嵌入式设备上的核心组件，其安全性对整个嵌入式设备的安全性起着至关重要的作用。

功能正确的嵌入式 JavaScript 引擎是保证 JavaScript 程序正常运行的前提，因此保证嵌入式 JavaScript 引擎功能的正确性至关重要。一方面，JavaScript 引擎的功能错误会导致 JavaScript 程序运行错误。另一方面，对于 JavaScript 开发人员来说，JavaScript 引擎通常是可信任的软件，功能异常的 JavaScript 引擎会增加 JavaScript 开发人员调试和维护代码的负担。在功能错误的 JavaScript 引擎上调试的代码移植到功能正常的编译器上会导致执行结果错误，这种编译器错误会影响 JavaScript 程序的可移植性。在保证 JavaScript 引擎功能正确性方面，当前主流的解决方案是使用 ECMAScript-262

的官方测试套件，但手动编写的测试套件也存在测试覆盖率不完整的问题。

嵌入式 JavaScript 引擎运行在计算能力弱和低功耗的嵌入式设备上，通常要求采用的算法具备高效性且不会占用太多的内存空间。存在性能缺陷的嵌入式 JavaScript 引擎会进行大量的复杂计算，不仅浪费计算能力不足的嵌入式设备的计算资源，还增加嵌入式设备的功耗。关于嵌入式 JavaScript 引擎性能缺陷检测的相关研究较少，而且到目前为止尚未有学者使用差分测试对嵌入式 JavaScript 引擎的性能缺陷进行检测。

2.2 差分模糊测试方法

在保证编译器的安全性和功能正确性方面，当前主流的缺陷检测方法是模糊和差分测试。本节将对本文使用到的差分测试和模糊测试方法展开介绍。

2.2.1 模糊测试方法

模糊测试[32,33]是软件安全保护中常用的测试技术，无论是黑盒测试[34,35]，灰盒测试[36]，还是白盒测试[37]都大量使用了该技术。模糊测试是一种通过操作系统的安全机制自动检测软件缺陷的方法。其原理是在程序执行过程中，操作系统的安全监测模块会监测程序执行过程中是否存在不安全的操作，如果存在不安全操作，操作系统会终止程序的执行并返回特殊的信号。通过捕获进程执行后返回的特殊信号可以判断是否触发了软件的漏洞。因为模糊测试是利用操作系统的安全机制实现软件测试的，因此模糊测试通常只用于测试软件的安全性缺陷。

2.2.2 差分测试方法

针对当前基于断言测试编写的语言规范配套的官方测试套件难以保证软件功能正确性的问题，McKeeman W M[37]等人提出了差分测试。差分测试[38-40]被广泛应用于大型软件系统的测试。使用差分测试的前提条件是存在两个或两个以上可以对比的测试对象，若执行结果不同则可能触发了软件系统的缺陷。差分测试被成功的用于编译器等大型软件系统的缺陷检测，例如，专注于测试 C 编译器缺陷的 CSmith，基于深度学习的编译器自动化测试方法 DeepSmith。他们使用差分测试对编译器进行测试的原理都是一样的，按照同一个规范实现的编译器具有相同的功能，给功能相同的编译器提供相同的输入，若其中某个编译器的输出与其余编译器的输出不同，则这些编译器中一定存在功能错误的编译器。以 C 编译器为例，严格按照 C99 规范实现的 C 编译器虽然会根据其应用场景的不同采用不同的算法，但他们的最终实现的功能应该是

一致的，因此差分测试被成功的应用到 C 编译器的缺陷检测中[39,41]。

目前差分测试主要专注于检测编译器或解释器的功能性缺陷。差分测试最重要的研究内容是如何获取具有检测软件缺陷能力的测试用例。在编译器测试中，基于模型的测试用例生成是一种比较常见的测试用例自动获取的手段。下面将对两种基于模型的测试用例生成方法展开介绍。

Yang X[13]等人利用 C 语言的语法规则定义了一个 C 语言语法规则子集的随机生成模型实现测试用例的自动生成，根据 C 语言的语法规则自动生成测试用例可以避免测试用例出现语法错误，通过复杂的动态分析保证生成的测试用例中不存在变量未定义的情况，这样生成的测试用例能轻松通过 C 编译器的前端，并专注于测试编译器更复杂的中端。在此方法基础上开发了自动测试工具 CSmith 并用于 C 编译器的测试，CSmith 在缺陷检测方面取得了优异的成绩，并为后续的编译器自动测试提供了指导性的作用。

Cummins C[14]等人提出了一种快速，高效，低工作量的随机测试用例生成方法，利用深度学习自动构造人类手写代码的概率模型代替手动定义语法规则，极大的降低了设计测试用例生成模型的构建成本。将手写的代码作为训练数据输入到深度学习模型中，训练一个具有推断编程语言语法和语义以及常见结构和模式能力的语言模型。通过将随机程序生成的问题转化为构建语言生成模型的问题，极大的简化了测试用例生成的过程。使得测试用例生成只与训练语言模型的语料库有关，而与测试用例自动生成模型的开发人员的能力水平和开发成本无关。使用深度学习算法自动生成测试用例的方法降低了构建语言模型的成本且具有容易实现跨语言的优点，经过 1000 个小时的测试，此方法共发现了 67 个开源商业编译器的缺陷。

2.3 测试结果处理方法

差分模糊测试过程中，使用大量随机测试用例进行自动化测试后会存在触发缺陷的测试用例复杂和重复触发大量相同缺陷两个问题。针对这两个问题，有学者提出了删除测试用例中无关代码的测试用例精简算法和缓解测试结果重复的测试结果过滤方法。本节将对其详细展开介绍。

2.3.1 测试用例精简方法

为了触发缺陷而获取到的测试用例通常功能复杂且代码比较长，导致触发缺陷的测试用例包含大量的与触发缺陷无关的代码。正确理解缺陷需要先理解测试用例，将

复杂的测试用例简化为简单测试用例并保证简单用例与复杂测试用例所触发的缺陷相同是缺陷检测的重要步骤，通用的手段是采用手动删除测试用例中无关代码的方式实现测试用例精简。于是有人提出了测试用例自动精简方法，用于自动化地将测试用例中的无关代码删除并得到能触发原缺陷的简单用例。

Zeller[42]等人提出的一种通用的测试用例精简算法。由于可以按需求对待精简的输入进行字符，代码行，标签等形式的拆分，因此精简的对象可以是字符串，代码，和超文本标记语言（HTML）等任意形式的输入。在精简过程中，当精简后测试用例的执行结果与精简前的测试用例执行结果相同，则认为精简保留了原有的属性。其精简算法 ddmin 包含以下三个步骤。第一步，列表 L 会被等分为 N 份，对于列表中的每一份 U，只保留 U 并判断只保留 U 时是否保留了原有的属性。若仍然保留了原有的属性 U，则删除 L 中 U 的补集，并执行第一步。第二步，只保留列表 L 中 U 的补集并判断是否保留了原有的属性。若仍然保留了原有的属性，则删除 U 并重新执行第一步。第三步，把剩余的每一份分成 2 份。即把列表 L 从 N 份拆分为 2N 份，并把拆分得更细的 2N 作为 L 并重复第一步。当 L 中的每一份都没法按照指定的规则继续拆分时，则 L 中剩余的元素就作为精简的结果，精简结束。

Lithium[4]是 Mozilla 公司开源的一种基于代码行的测试用例精简工具。Mozilla 一直都积极的对该项目进行维护，而且该公司的模糊测试项目组用此工具精简了数百个触发了浏览器崩溃的测试用例。在保证测试用例精简后所触发的可疑缺陷与精简前所触发的可疑缺陷一致方面，Lithium 采取的策略是根据引擎执行测试用例后的输出或异常信息中是否包含关键信息作为判断测试结果是否改变的判断条件，即测试用例是否可精简的判断条件。其精简算法的实现是 ddmin 算法的修改版本，Lithium 默认以代码行为单位，使用二分法删除测试用例中与触发可疑缺陷无关的代码从而实现测试用例精简。考虑到 JavaScript 测试用例中的语句不是以行为单位的情况，其允许将 JavaScript 测试用例视为字符序列并以字符为单位对测试用例进行精简，但以字符为单位也意味着会带来更大的性能开销。

Lithium 的精简算法介绍将以图 1 中的测试用例为例，假定测试用例的预期输出为 3 时，以代码行为单位，使用 Lithium 对测试用例进行精简。第一轮精简：由于代码的总行数为 6，而 6 大于 2 的二次方小于 2 的三次方，因此测试用例精简的第一轮会以 4 为初始块大小。首先尝试删除测试用例的 3-6 行，删除后测试用例的输出不包含 3，此次删除无效。由于 1-2 行未达到此轮精简的块大小限制 4，因此不再尝试对

1-2 行进行删除。第二轮精简：块大小缩减为上一轮块大小的二分之一，即第二轮精简的块大小为 2。第二轮精简过程中代码块的删除顺序是 5-6 行、3-4 行、1-2 行，由于删除 5-6 行后输出中仍然包含 3，因此 5-6 行删除有效。删除 3-4 行或 1-2 行后都不再输出 3，因此均不能删除。同上，第三轮精简：代码块大小为 1，代码块的精简顺序分别是删除代码的 4 行、3 行、2 行、1 行，此轮精简只删除了测试用例的第 4 行。由于代码块大小目前已经是最小单位 1，所以精简结束，并且只保留了原测试用例中无法再进行删除的 1-3 行代码，实现了测试用例的自动化精简。Lithium 的精简算法擅长精简更长的测试用例，为了方便演示，本文使用 6 行代码演示其精简算法。

```
1. var a = 1;
2. var b = 2
3. print(a + b);
4. print(a * b)
5. print(a - b);
6. print(a / b);
```

图 1 测试用例示例

2.3.2 测试结果过滤方法

在差分模糊测试中，随机输入的测试用例会重复触发大量已经发现过的缺陷，重复产生的测试结果会影响测试人员从测试结果中寻找未知缺陷的效率。因此，将测试结果中重复的结果过滤非常重要，而测试结果过滤的任务通常是由人手动完成的[6]。目前在解决测试结果重复问题方面的研究比较少[6,43]，下面将对几种较新的研究成果展开介绍：

（1）基于排序的测试结果过滤方法

Yang Chen[6]等人以测试用例相似度越低则测试结果重复的可能性越低为思路，提出了使用机器学习算法对触发了编译器缺陷的测试用例进行排序的方法，给重复概率低的测试结果分配更高的优先级，集中主要精力分析优先级高的测试结果，达到缓解测试结果重复造成的测试结果分析成本高的目的。

（2）基于调用栈和执行路径的崩溃测试结果去重方法

导致崩溃的测试用例的唯一性可以由目标线程的调用堆栈和导致崩溃的指令地址来确定[44]。如果两个触发崩溃的不同的测试用例的调用栈相同，则这两个测试用例很可能触发了相同的崩溃，只需要保留其中的一个进行人工分析。相反，如果它们导致目标程序相同位置的崩溃，但使用不同的堆栈跟踪，它们很可能对应两个不同的崩溃，则两个测试用例都需要分析。

（3）基于覆盖率的崩溃测试结果去重方法

AFL[45]是一个知名的以覆盖率为指导的模糊测试器，通过代码插桩收集被测程序运行时的路径信息。如果触发崩溃的测试用例找到了待测程序的一条全新的路径或者已经发现崩溃的测试用例中不存在与当前测试用例完全相同的执行路径时，则认为当前测试用例触发的崩溃是唯一的。AFL 的崩溃测试结果过滤方法正是利用其基于执行路径的模糊测试方法。通过生成独特的诱导崩溃的测试用例有助于减少冗余的模糊测试结果，为人工分析节省时间和精力。

2.4 本章小结

本章介绍了相关理论与技术，首先介绍了 JavaScript 编程语言的相关背景知识，嵌入式 JavaScript 引擎及其测试相关内容。接下来，介绍了模糊测试和差分测试两种当前主流的黑盒缺陷检测方法，并对两种缺陷检测方法的原理进行了详细的描述。最后，描述了当前差分模糊测试存在的测试用例包含大量无关代码和测试结果冗余两大问题，并分别展示了几种已有的解决方法。

第三章 以性能缺陷检测为导向的差分模糊测试方法研究

自动化测试中保证测试方法有效性的手段主要有两个，其一是测试用例生成质量，其二是判断是否是存在缺陷的缺陷检测方法。高质量的测试用例是触发缺陷的关键，缺陷检测方法决定了能否检测出缺陷。本章将针对嵌入式 JavaScript 引擎的性能缺陷检测的问题，提出一种以性能缺陷为导向的测试用例生成方法和具有性能缺陷检测能力的差分模糊测试方法。本文的方法不仅具有检测性能缺陷的能力，还具有检测功能缺陷和崩溃的能力。

3.1 方法概述

本章提出了以性能缺陷检测导向的差分模糊测试方法，其主要目标是通过具有导向性的测试用例生成并设计相应的差分模糊测试方法检测嵌入式 JavaScript 引擎的性能缺陷。测试用例的质量决定了测试系统是否具有缺陷检测的能力，于是本文提出了语法正确且语义丰富的测试用例生成方法，以及为了提高测试用例检测嵌入式 JavaScript 引擎性能缺陷能力而设计的测试用例变异方法。检测嵌入式 JavaScript 引擎的性能缺陷不仅需要具有性能缺陷检测能力的测试用例，还需要自动识别性能缺陷的缺陷检测方法。为此，提出了以性能缺陷检测为导向的差分模糊测试方法，该方法不仅能检测嵌入式 JavaScript 引擎的性能缺陷，还能检测嵌入式 JavaScript 引擎的功能缺陷和崩溃。以性能缺陷检测为导向的差分模糊测试方法流程图如图 2 所示。

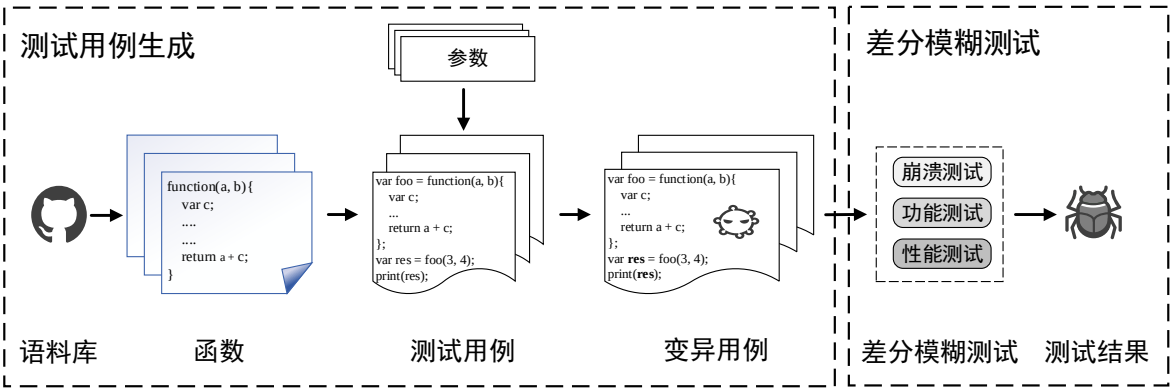

图 2 以性能缺陷检测为导向的差分模糊测试方法流程图

根据功能的不同，以性能缺陷检测为导向的差分模糊测试方法将划分为测试用例生成和差分模糊测试两个核心部分：

（1）测试用例生成

为了确保测试用例来源广泛和测试用例质量，本文首先从知名代码托管平台GitHub上收集排名靠前的开源JavaScript代码仓库，称之为语料库。然后，提取语料库中的函数，以保证测试用例具有完整的语义信息并扩大语料库的利用率。接下来，对这些函数进行调用并传递相应数据类型的参数保证函数中丰富的语义信息被充分利用。最后，为了进一步提高测试用例检测嵌入式 JavaScript 引擎性能缺陷的能力，还设计了测试用例变异方法对测试用例进行变异。

（2）差分模糊测试

自动生成的测试用例除了具有检测嵌入式JavaScript引擎性能缺陷的能力外，还具有检测功能缺陷和崩溃的能力。为了充分利用测试用例的价值，本文改进了差分模糊测试方法实现了嵌入式JavaScript引擎的崩溃测试，功能测试和性能测试。

3.2 以性能缺陷检测为导向的测试用例生成方法

高质量的测试用例是测试的前提，为此本文广泛收集开源代码库中评价较高的JavaScript代码仓库，如GitHub上的start排名靠前的JavaScript代码仓库。通过这种方式获取的语料库具有语义丰富，API 覆盖率广等优点。为了充分利用语料库中的JavaScript 代码并保证测试用例语义的完整性，从 JavaScript 代码中提取出功能较为完整的函数。然后对提取出来的函数进行调用并传递相应数据类型的参数，保证函数体尽可能多的被执行，以提高测试用例的代码覆盖率。调用并传递参数后的测试用例仍然存在性能缺陷检测能力不足的问题，本文还针对此问题提出了测试用例变异方法以提高测试用例的性能缺陷检测能力。

3.2.1 函数提取

语料库中JavaScript代码功能丰富，但使用整个项目或单个文件作为测试单元时没有充分地利用有限的 JavaScript 代码资源。这种利用不充分是由于 JavaScript 代码中执行顺序靠前的代码抛出异常后，后续的代码将不再执行导致的。为此，本文从语料库中提取代码长度适中且功能相对完整的函数，既保证测试用例具有相对完整的功能还能充分利用语料库语义丰富的优点。在提取函数时，采用正则表达式匹配的方式提取JavaScript代码段中的函数块，提取的过程中需要尽可能地保证函数块的功能完整性。然后，使用 JavaScript 前端编译器对函数进行处理，使 JavaScript 函数具有语法正确，不包含注释和格式统一的特点。接下来，计算函数的哈希值并比较不同函数的哈希值是否相同实现函数的过滤，避免使用这种完全重复的函数进行测试造成的计

算资源浪费。函数中不包含注释和格式统一的特点能保证基于哈希值比较的去重方法的去重效果。这种提取函数的方法保证了测试用例的语法正确性，这样做是因为编译器前端相对简单，编译器前端的缺陷仅占编译器缺陷的 3%[7]，因此本文将专注于检测编译器中端的缺陷。除了保证语法正确之外，提取函数时还会过滤测试用例中 ES6 以上规范定义的功能，其原因是本文测试的绝大多数嵌入式 JavaScript 引擎都不支持 ES6 以上的规范，测试用例中 ES6 以上规范定义的功能只会增加手动分析测试结果的成本。

3.2.2 测试用例生成

若函数只定义而不调用则引擎只会检查其语法正确性，而不会检查其语义的正确性，这样的测试用例只能检查 JavaScript 引擎的前端错误。为了尽可能多的发现引擎中端的缺陷，需要对函数进行调用并传递相应数据类型的随机参数。参数传递的过程中会使用简单的类型推导方法推断出最有可能的参数类型，并随机选择相应数据类型的值作为函数的参数[46]。传递正确的参数类型可以缓解程序提前终止造成的函数利用不充分的情况发生，随机值作为参数增加了触发缺陷的可能性。图 3 展示了测试用例生成过程的示例，测试用例生成是在原有函数定义的基础上添加函数调用，并把函数的执行结果打印输出。

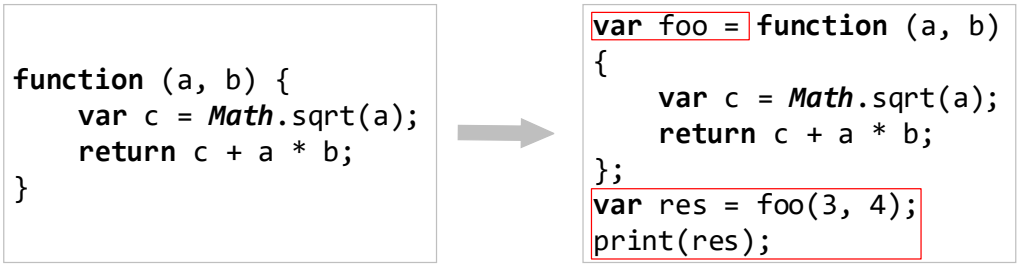

图 3 测试用例生成示例

3.2.3 测试用例变异

经过函数提取和函数调用后得到的测试用例具备测试引擎功能缺陷和崩溃的能力，但是测试用例触发引擎性能缺陷的能力有限，需要对测试用例变异以适用于检测嵌入式 JavaScript 引擎的性能缺陷。

测试用例性能缺陷检测能力不足的问题主要来自于两个方面，一方面，计时器精度与 JavaScript 操作的执行时间不匹配使性能测试变得困难。大部分 JavaScript 操作的执行时间都远小于 1 毫秒，而程序计时器的精度几乎都在毫秒级别。例如，JavaScript 中计时器的精度在不同的 JavaScript 引擎中实现不一致，目前精度较高的 JavaScript

计时器是 time.js[47]，其精度也只能达到毫秒级别。另一方面，JavaScript 引擎在执行简单的操作时可能不会暴露出潜在的性能问题，但随着操作次数的不断增加，引擎的执行效率会越来越低，当达到一定程度时这种性能缺陷就会表现出来。例如，在测试数组扩容算法是否高效时，需要测试用例中存在扩容操作且扩容的频率要足够高。

为了解决测试用例性能缺陷检测能力不足的问题，提出通过改变测试用例中语句执行次数的方式对测试用例进行变异的方法，从而提升测试用例的性能缺陷检测能力。一方面，提出了增加某一条或多条语句的执行次数，降低计时器精度不足的影响，并将隐藏的性能缺陷放大并暴露出来。同时，改变语句的执行次数还可能会改变测试用例的语义信息，实现测试用例的变异。通常本文采用的方式是在原语句的外层增加循环结构，这样可以保证在测试用例体积不膨胀的情况下增加语句的执行频率。由于嵌入式 JavaScript 引擎运行在资源受限的设备上，所以嵌入式 JavaScript 引擎都没有采用即时编译技术，因此这种通过循环增加语句执行次数的变异方法的不会受到即时编译技术对高频率代码进行优化编译的影响。图 4 展示了测试用例变异的示例，示例中增加了测试用例中第三行代码的执行次数。变异后的测试用例就具备了检测引擎实现 Array.prototype.push 方法时是否存在严重性能缺陷的能力。

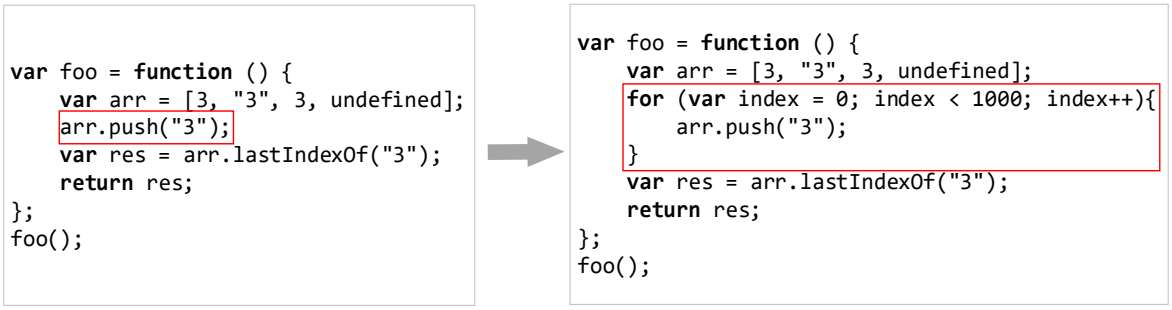

图 4 测试用例变异示意图

无区别的对测试用例中的每条语句都分别添加循环并利用其进行测试可能会导致大量完全等价的测试用例被反复执行。因为原始测试用例中的部分分支可能没有被执行，对此分支中的语句添加 for 循环没有意义且浪费计算资源。为了避免这种无效变异造成计算资源的浪费，本文还采用了插桩的方式对测试用例中未执行的语句进行检测并删除，只保留测试用例中能被执行的语句。删除未执行语句后的测试用例将作为性能变异的种子用例。

另一方面，嵌入式 JavaScript 引擎的性能问题几乎都与循环相关，本文还针对测试用例中的循环语句进行了变异。变异主要针对循环语句中的循环遍历顺序，循环次数，循环步长以及索引数据类型等进行变异。例如，根据循环语句正序和逆序执行所

涉及的编译器分支不同本文会将循环的执行顺序从正序改为逆序，逆序改为正序。JavaScript 弱类型的特点使不同数据类型之间的可以轻松实现数据类型转换，因此本文还支持对循环语句的索引类型进行变异。

3.2.4 性能变异算法

针对测试用例的性能变异是通过删除测试用例中未被执行的语句的执行次数，并给测试用例中能够被执行到的语句添加循环达到改变单条语句执行次数的目的，每次变异都确保测试用例中只有一条语句的执行次数被改变。每为测试用例中的一条语句添加一次循环就可以得到一个测试用例。详细的算法如下：

算法 1 测试用例变异算法

输入：testcase，字符串表示的删除了未执行语句的 JavaScript 代码段

输出：mutatedTestcaseList，变异的测试用例集合

```
1:  mutatedTestcaseList = []
2:  testcaseAST = generateAST(testcase)
3:  queue = [testcaseAST]
4:  while len(queue) > 0 do
5:      node = queue.pop()
6:      for key, value in node.items() do
7:          if type(value) == list and key == "body" then
8:              childNodeList = value
9:              for index in range(len(childNodeList)) do
10:                 queue.add(childNodeList[index])
11:                 mutatedChildNode = deepcopy(childNodeList)
12:                 mutatedChildNode = addLoopStructure(mutatedChildNode, index)
13:                 node[key] = mutatedChildNode
14:                 mutatedTestcase = generateJSCode(testcaseAST)
15:                 mutatedTestcaseList.add(mutatedTestcase)
16:                 node[key] = childNodeList
17:             end for
18:         else if type(value) == dict then
19:             queue.add(value)
20:         end if
21:     end for
22: end loop
23: return mutatedTestcaseList
```

算法分析如下：

1-3 行：初始化存储变异测试用例的数组 mutatedTestcaseList，将测试用例转化成抽象语法树，并将抽象语法树的根节点添加到待变异的节点队列中。

4-6 行：采用广度优先遍历算法遍历抽象语法树中的每一层节点，遍历的顺序是由外层到内存逐层遍历，直到当前节点中不存在子节点为止。

7-17 行：查找当前节点的子节点是否可以添加循环且不会导致语法错误，即当前节点的子节点是语句类型。在抽象语法树上分别对每条语句添加循环实现变异，每一次变异都生成一个变异测试用例并且在获取变异测试用例后将抽象语法树恢复为未变异前的测试用例的语法树状态，保证每个变异的测试用例只修改原测试用例中一条语句的执行次数。

18-20 行：如果当前节点只是一个普通的节点，则将其加入待变异的节点队列中。如果其不是节点类型，结束对当前节点的变异。

3.3 具备性能缺陷检测能力的差分模糊测试方法

上一节介绍了测试用例生成方法得到的用于测试嵌入式 JavaScript 引擎缺陷的变异用例。本节将介绍使用变异用例在不同嵌入式 JavaScript 引擎上的执行结果进行对比的缺陷检测方法——差分模糊测试。差分模糊测试可以分为模糊测试和差分测试。本文中的模糊测试主要用于检测软件是否出现崩溃；差分测试主要用于检测引擎的功能缺陷，同时对差分测试进行改进使其适用于检测引擎的性能缺陷。

3.3.1 模糊测试

表 1 常见的 POSIX 信号[48]

码值	信号	描述
1	SIGHUP	挂起
2	SIGINT	终端中断信号
3	SIGQUIT	终端退出信号
4	SIGILL	非法指令
5	SIGTRAP	跟踪/断点陷阱
8	SIGFPE	错误的算术运算
11	SIGSEGV	无效内存访问

JavaScript 引擎作为 JavaScript 程序运行的载体，保证嵌入式 JavaScript 引擎的鲁棒性对嵌入式设备的安全稳定运行至关重要。本文的模糊测试是通过捕获 JavaScript 引擎在 POSIX[49]兼容的操作系统上执行测试用例后返回的 POSIX 信号[50]来实现测试

的。当从引擎执行测试用例的进程中捕获到 POSIX 信号时，表明引擎执行此测试的进程被操作系统中断了，并且认为引擎出现了崩溃，常见的 POSIX 信号如表 1 所示。例如，信号 SIGSEGV 通常代表无效内存访问或者发生了段错误，这通常是一种很危险的操作，并且这种缺陷可能会被黑客利用并进行安全攻击。

3.3.2 差分测试

模糊测试只能检测嵌入式 JavaScript 引擎执行过程中是否出现了崩溃。嵌入式 JavaScript 引擎在执行过程中没有被操作系统中断，即使引擎实现的功能不符合 JavaScript 语言规范的定义也无法被模糊测试方法检测出来。为了测试嵌入式 JavaScript 引擎功能的正确性，主流的自动化缺陷检测方法是差分测试，但差分测试仍然未被用于检测性能缺陷。如果引擎的执行结果正确，即使引擎实现此功能时采用的算法低效或编码错误等问题导致了性能缺陷也无法被差分测试方法检测出来。为此本文提出了使用差分测试检测引擎性能缺陷的思路。根据测试重点的不同，将差分测试分为功能测试和性能测试两个模块。

功能测试：测试嵌入式 JavaScript 引擎实现的功能与 ECMAScript-262 规范定义的功能是否一致，即测试引擎功能的正确性。待测试的嵌入式 JavaScript 引擎都遵循 ECMAScript-262 规范，规范详细定义了相应的功能应该达到的目标和实现的关键步骤。这些引擎实现同一个功能采用的算法可能不同，但其实现此功能最终应该达到的目标必须是一致的。差分测试正是基于这一特点对嵌入式 JavaScript 引擎进行测试的。判断引擎实现的功能是否正确需要一个标准结果与其对比，若某个引擎的执行结果与标准结果不符，则这个引擎没有正确实现此功能。待测试的引擎是不可靠的，不能指定某个引擎的执行结果作为标准的结果。差分测试方法采用了所有引擎对执行结果进行投票的方式，选择最有可能的正确执行结果作为标准结果。若某个引擎执行结果与标准结果不符则认为此测试用例可能触发了该引擎的缺陷，再经过人工介入的方式可以精确判断测试用例是否真正触发了引擎的缺陷。测试用例中可能会存在生成随机值，或者获取当前时间戳等操作，执行这样的测试用例可能会导致引擎之间的执行结果都不一致。因此，在进行投票的时候需要设置一个阈值，若引擎投票选出的标准结果的票数与待测引擎的总数之比小于这个阈值，则认为当前选出的标准结果不可靠并认为当前测试用例无效。

性能测试：当功能相同的引擎执行同一个测试用例时得到的执行结果相同，并且某个引擎的执行时间远高于其他引擎，则表明该引擎可能存在性能缺陷。经过模糊测

试和功能测试后，需要对既没有出现崩溃，执行结果也正确的引擎进行进一步分析，以检测其隐藏的性能缺陷。执行结果正确的引擎中，同样需要选择一个性能正常的引擎作为对照，本文选择了执行速度最快的引擎的执行时间作为标准时间。引擎执行测试用例的时间会受到计时器精度，CPU 调度和引擎实现算法等诸多因素的影响，所有引擎的执行时间都不相同。为了消除上述因素造成的影响，本文会先计算引擎的执行时间与标准时间的比值，记为性能因子，并将性能因子与一个阈值进行对比。若性能因子大于阈值，则认为引擎存在严重的性能缺陷，否则认为这个引擎的执行时间在可接受范围内。此处的阈值可以根据对引擎性能要求的严格程度对其进行调整，如果性能要求高则可以降低阈值，本文选择 10 为阈值是为了发现引擎的严重性能缺陷。

由于差分测试主要用于检测编译器的功能缺陷，尚未找到使用差分测试检测编译器性能缺陷的相关研究工作。为了将测试功能缺陷的差分测试应用于检测引擎的性能缺陷，需要将测试功能缺陷的差分测试的方法进行改进。功能测试时，使用投票的方式选择标准结果，而性能测试时每个引擎的执行时间几乎不可能相同，因此本文选择执行速度最快的引擎的执行时间作为标准时间。这样选择是默认了执行结果都相同的引擎实现的功能相同且正确，当引擎的功能正确则执行速度越快越好，因此选择了执行速度最快的引擎的执行时间作为标准时间。在判断是否存在性能缺陷时，没有直接比较引擎的执行时间，而是通过比较性能因子与阈值的大小关系，缓解了执行时间偏差带来的性能缺陷判断困难的问题。

3.3.3 方法实现

前面介绍了差分模糊测试方法的设计思路，接下来将介绍差分模糊测试的方法实现。图 5 展示了 JavaScript 引擎执行测试用例及其可能的输出类型，引擎执行测试用例后的输出类型分为 4 类：超时，崩溃，脚本异常和运行通过。下面将对差分模糊测试的具体方法展开介绍：

（1）以 JavaScript 代码段作为测试用例，并使用多个嵌入式 JavaScript 引擎执行测试用例，同时捕获引擎执行测试用例后的返回码，标准输出，标准错误输出以及执行时间等信息作为引擎的执行结果。为了避免无限循环等无意义的测试用例导致的计算资源浪费，还需要限制引擎执行测试用例的时间。后续将利用引擎执行测试用例后的状态信息判断引擎是否存在缺陷。

（2）崩溃测试，判断引擎执行测试用例的进程是否被操作系统中断，即是否捕获到 POSIX 信号。若引擎执行的正常流程被中断且执行未超过限制的时间则认为引

擎出现了崩溃。记录此测试用例及其在各个引擎上的执行结果并标记此测试用例发现了引擎的崩溃。

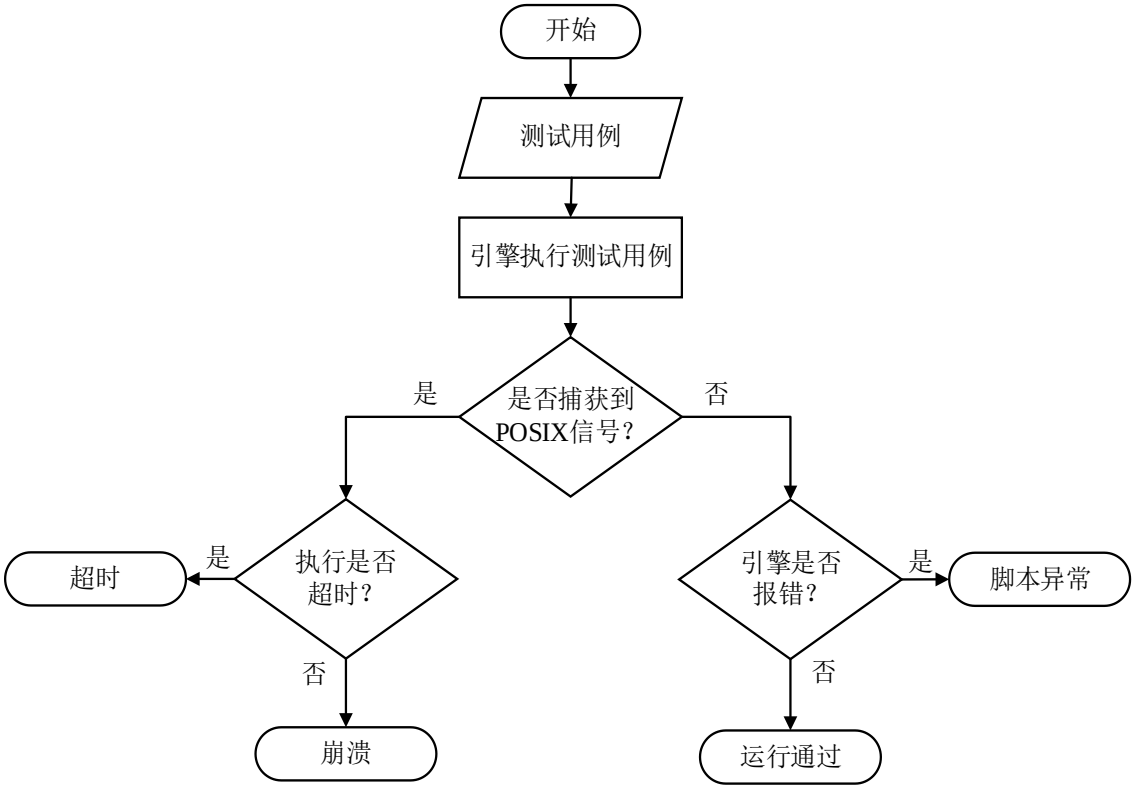

图 5 JavaScript 引擎执行测试用例流程图

（3）功能测试，当引擎执行测试用例的进程未被中断，则引擎有两种可能的输出类型，即抛出测试用例中的脚本异常或测试用例执行通过并打印输出。若某个引擎的输出类型与大多数引擎的输出类型不同时，则这个引擎可能存在功能缺陷。当大部分引擎都运行通过且打印输出相同的值，且某个引擎的输出与大多数引擎的输出值不同，则这个引擎也会被标记存在功能缺陷。

（4）性能测试，若大部分引擎的输出类型都一致且未被标记为崩溃或功能缺陷，则比较这些功能正常的大多数引擎的执行时间，根据执行时间判断功能正确的引擎是否存在性能缺陷。首先，统计这些输出类型一致的引擎中执行速度最快的引擎的执行时间作为标准时间，并计算这些输出类型一致的引擎的执行时间与标准时间的比值，即性能因子。若性能因子大于预设的阈值，则认为引擎存在的性能缺陷，否则不存在性能缺陷。

3.3.4 差分模糊测试算法

前面介绍了差分模糊测试方法利用引擎执行结果判断引擎是否存在缺陷的思路，其主要利用了引擎执行测试用例后的返回码，标准输出，标准错误输出，以及引擎的

执行时间判断引擎是否存在缺陷。下面将介绍其详细的算法实现：

算法 2 差分模糊测试算法

输入：harnessResult, 记录了测试用例和各个引擎执行测试用例后的状态信息

输出：suspiciousResults，记录存在可疑缺陷的引擎及其缺陷类型的数组

```
 1 performanceFactor = 10
 2 ratio = 2 / 3
 3 majority = getMajority(harnessResult)
 4 testbedNum = len(harnessResult.outputs)
 5 suspiciousResults = []
 6 for output in harnessResult.outputs do
 7       if output.class == "crash" then
 8             suspiciousResults.add(["crash", output.testbed])
 9       else if majority.majorityClassSize / testbedNum < ratio then
10             continue
11       else if majority.majorityClass != output.class then
12             if majority.majorityClass == "pass" or majority.majorityClass == "script_error" then
13                   suspiciousResults.add(["Functional issue", output.testbed])
14             end if
15       else if majority.majorityClass == "pass" and output.class == "pass" and
16             output.stdout != majority.majorityStdout and
17             majority.majorityStdoutSize / majority.majorityClassSize >= ratio then
18                   suspiciousResults.add(["Functional issue", output.testbed])
19       end if
20 end for
21 if not isReliable (harnessResult) then
22       return suspiciousResults
23 normalOutputs = getNormalOutputs()
24 minDuration = getMinDuration(normalOutputs)
25 for output in normalOutputs then
26       if output.duration / minDuration >= performanceFactor then
27             suspiciousResults.add(["Performance issue", output.testbed])
28       end if
29 end for
30 return suspiciousResults
```

算法分析如下：

1-5 行：performanceFactor 是判断是否触发引擎性能缺陷的阈值，其含义是如果

引擎的执行时间与最快引擎的执行时间比值大于这个值则存在性能缺陷，否则不存在性能缺陷。ratio 是判断投票方式选出的标准结果是否有效的阈值，当得到标准结果的引擎数量与总的待测引擎的数量的比值小于 ratio 的值时，则认为这个标准结果是不可信的。majority 是记录大多数引擎的输出类型，执行通过时的输出，以及相应输出类型和相应输出对应的引擎数量。输出类型分为四类：崩溃，超时，抛出异常和执行通过。testbedNum 表示待测引擎的总数，suspiciousResults 用于存储差分模糊测试发现的可疑缺陷的引擎及其缺陷类型。

6-8 行：遍历所有引擎执行测试用例后的状态信息，如果引擎的输出类型为崩溃，则直接标记为崩溃并记录崩溃的引擎。

9-10 行：如果输出类型为标准输出类型的引擎数量与待测引擎数量的比值小于阈值 ratio，则当前测试用例的可能有不同输出并且放弃使用差分测试技术对此测试用例的执行结果进行功能测试。

11-19 行：使用差分测试技术判断引擎是否存在功能缺陷。其中，11-14 行查找引擎输出类型与投票选出的标准输出类型不同的引擎并标记为功能缺陷。此处并没有将所有与标准输出类型不同的引擎都标记为功能缺陷，只把标准输出类型是执行通过或脚本异常且引擎输出类型与标准输出类型不同的引擎标记为功能缺陷。其原因如下，大多数引擎执行超时通常是测试用例中存在复杂的循环或递归调用引起的，这种测试用例通常没有触发有意义的缺陷。而触发了大部分引擎崩溃的测试用例在测试中没有出现。15-19 行则是判断测试用例执行通过的引擎中是否存在打印输出的结果不正确的情况，同样，这里也需要投票选出的标准的打印输出信息并判断此次投票是否有效。

21-22 行：判断投票选出的标准输出类型或标准输出是否可信，若不可信，则当前测试用例无需进行性能测试。

23-24 行：normalOutputs 为既没有崩溃也没有功能缺陷的引擎及其执行测试用例后的状态信息。minDuration 记录了未发现缺陷的引擎中执行测试用例最快的引擎的执行时间，并作为标准执行时间。

25-30 行：遍历所有未发现缺陷的引擎的执行结果，如果引擎的执行时间与基准时间之比大于 performanceFactor，则此引擎被标记为存在可疑的性能缺陷。

3.4 本章小结

本章主要介绍了以性能缺陷检测为导向的差分模糊测试方法。差分模糊测试方法

最重要的是测试用例生成方法和差分模糊测试方法的设计两大模块。本章围绕了这两个主要模块分别展开介绍，首先介绍了测试用例生成，测试用例生成阶段从 GitHub 上获取排名靠前的优质 JavaScript 开源仓库并提取仓库中语法正确且功能相对完整的函数以保证测试用例的质量，对提取到的函数进行函数调用并通过类型推导随机传递相应数据类型的参数以获得测试用例，并针对测试用例难以满足性能测试需求的问题，提出了测试用例变异方法并设计相应的性能变异算法。最后介绍了差分模糊测试方法的设计，在差分测试阶段，使用模糊测试技术检测引擎的崩溃，使用差分测试检测引擎的功能缺陷，同时对差分测试进行改进，使其适用于检测嵌入式 JavaScript 引擎的性能缺陷。通过本文提出的测试用例生成方法和差分模糊测试方法可以检测嵌入式 JavaScript 引擎的性能缺陷，安全缺陷和功能缺陷。

第四章 以高精度为导向的用例精简与结果过滤方法研究

上一章提出了以性能缺陷为导向的差分模糊测试方法，设计了用于嵌入式 JavaScript 引擎缺陷检测的测试用例自动生成方法和相应的差分模糊测试方法，实现了嵌入式 JavaScript 引擎缺陷的自动检测。大量的差分模糊测结果需要手动分析并确认其是否触发了引擎的未知缺陷，然而测试结果中复杂的测试用例和大量重复的测试结果极大地增加了手动分析测试结果的成本。本章将对差分模糊测试结果中存在的两个问题提出自动化的解决方案。

4.1 方法概述

持续地进行缺陷检测会触发大量可疑的测试结果，这些测试结果主要有两个特点，其一，触发缺陷的测试用例会包含大量与触发缺陷无关的代码且功能复杂。无关代码会严重干扰测试人员正确理解软件缺陷，手动删除功能复杂的测试用例中的无关代码速度慢且会消耗大量的人力资源。其二，持续生成的测试用例会反复触发已经发现过的引擎缺陷或者误报，大量重复的测试结果给人工分析测试结果并判断其是否触发引擎的未知缺陷带来了困难。以上两点极大的增加了测试结果的分析成本，限制了差分模糊测试的广泛使用。

针对导致差分模糊测试结果分析成本高的测试用例复杂和测试结果大量重复这两个问题，本章分别提出了基于抽象语法树的测试用例精简方法和基于多维特征的测试结果过滤方法。基于抽象语法树的测试用例精简方法不仅便于测试人员分析测试结果，还能提高测试结果过滤的力度。图 6 展示了测试用例精简与测试结果过滤方法的流程图。

根据核心任务的不同，将测试用例精简与测试结果过滤方法分为测试用例精简和测试结果过滤两部分：

（1）测试用例精简方法

获取触发了引擎可疑缺陷的测试用例，以下简称可疑用例。为了提高测试用例精简的速度和精简的力度，本文会在测试用例的抽象语法树上按代码块，分层对测试用例进行精简，同时保证精简前后的测试用例能触发相同的测试结果。

（2）测试结果过滤方法

精简后的测试用例通常只有简短的几行代码，对精简用例的变量名进行规范化后使用基于哈希值的比较的去重方法删除完全重复的测试用例及其测试结果。进一步提取规范化后测试用例执行结果中的关键信息，并使用基于多维特征的测试结果过滤方法对提取到的关键特征进行判断，实现测试结果的过滤。

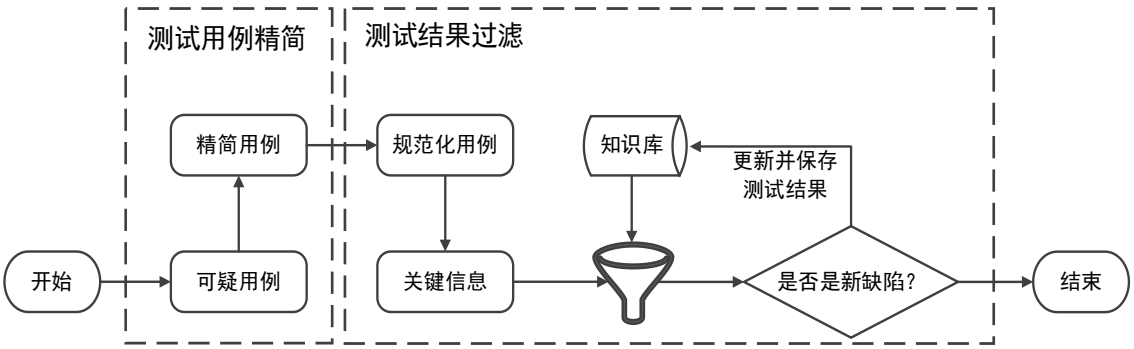

图 6 测试用例精简与测试结果过滤方法流程图

4.2 基于抽象语法树的测试用例精简方法

上一章中的差分模糊测试会触发大量崩溃，执行结果不一致和性能差距过大的测试结果，这些测试结果都需要经过人工确认，并将有意义的测试结果总结为一个缺陷检测报告反馈给引擎的开发人员。人工分析触发此测试结果的原因之前，首先需要从很长的测试用例中提取只与触发此测试结果有关的代码，保证手动分析测试结果的准确率和效率。本文提出了一种基于抽象语法树的测试用例精简方法，能删除测试用例中与触发测试结果无关的代码，并得到能触发原测试结果的简单测试用例。

4.2.1 测试用例精简的作用

随机获取的用于测试嵌入式 JavaScript 引擎的测试用例功能复杂，且包含大量与触发缺陷无关的代码。如何快速，准确地从功能复杂的测试用例中获取能触发原测试结果的简单有效代码段是一个需要解决的难题，这个过程称为测试用例精简。一方面，测试用例功能复杂，难以理解。为了充分理解测试结果并发现待测试引擎的缺陷，需要人工的删除测试用例中与触发测试结果无关的代码，使测试用例变得简单易懂。测试用例精简在测试结果分析过程中是一个必不可少的环节，而人工精简测试用例是一个非常耗时且乏味的工作。为了解决人工分析测试结果成本高的问题，本文提出了基于抽象语法树的测试用例自动精简算法，实现了快速，高效，准确的测试用例自动精简。另一方面，测试用例中存在大量功能差异很大的测试用例，但是这些测试用例经过精简和规范化变量名等操作后可以简化为相似或完全相同的测试用例，有利于后续

的测试结果去重。如图 7 所示，两个复杂的测试用例在经过精简后都只保留了 3 行简单代码，说明两个复杂的测试用例测试出了同一个测试结果，可以直接过滤其中一个测试用例的执行结果。

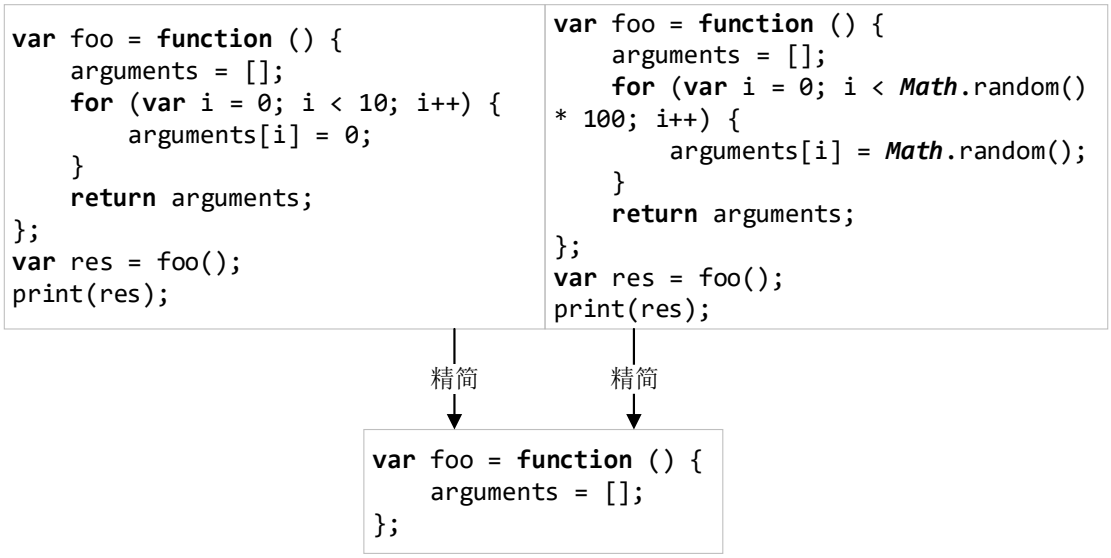

图 7 测试用例精简对测试结果过滤的效果展示

4.2.2 测试用例精简方法

测试用例精简通常采用的方式是删除测试用例中与触发测试结果无关的代码行，保留测试用例中能触发原测试结果的 JavaScript 代码作为精简用例。在测试用例精简过程中存在两大问题，其一，如何高效，彻底地删除测试用例中与触发缺陷无关的代码。其二，如何保证测试用例精简的正确性，即测试用例精简前后所触发的缺陷是同一个。针对如何高效，彻底地删除无关代码的问题，本文提出了抽象语法树级别的测试用例精简算法。针对如何保证测试用例精简正确性的问题，本文提出了具有多维信息的精简判断条件。

（1）抽象语法树级别的测试用例精简算法

针对如何高效准确地删除测试用例中的无关代码的问题，本文提出了基于抽象语法树的算法对测试用例进行精简。抽象语法树是对 JavaScript 源码的语法结构的抽象，基于测试用例的抽象语法树可以对测试用例进行精准操作，以便快速，彻底的精简测试用例。在抽象语法树上精简测试用例能有效避免尝试精简测试用例时出现语法错误的问题，提高精简的速度。这是由于精简前的测试用例语法正确，精简后的测试用例若存在语法错误，则测试结果一定改变，再使用精简判断条件判断测试用例是否可精简没有必要且浪费时间。

在精简测试用例时采用分层精简，逐层递进的方式，同一层节点使用从后向前逐

个节点依次尝试进行删除的精简策略。此方法使先执行的语句后进行精简，后执行的语句先进行精简，保证测试用例精简的效率。图 8 展示了测试用例中的语句在抽象语法树中的层次划分图，图中的 L 表示当前语句所在的层数，如 L-1 表示当前语句在抽象语法树中的第一层。

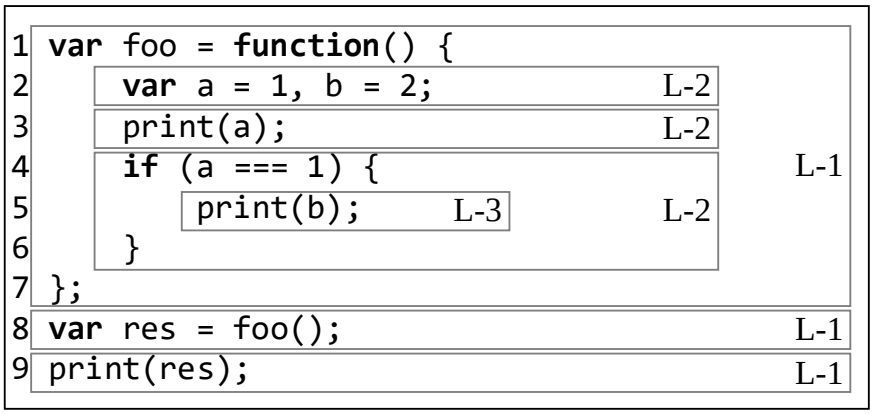

图 8 测试用例的语句在抽象语法树中的层次划分图

除了对测试用例中的语句进行精简，基于抽象语法树还可以轻松实现 JavaScript 代码中的参数和变量声明的精简。测试用例中的参数和变量名的精简如图 9 所示，假设精简后的测试用例预期输出是 1。当第 5 行代码无法被精简时会尝试精简其子节点，即精简实参，若 JavaScript 中的函数调用时没有提供实参，系统会默认传递参数 undefined，因此第 5 行的两个实参都可以被精简。在精简第 2 行中的语句时，由于删除变量 c 的声明后测试用例不再输出 1，因此第 2 行变量声明语句不能删除，但是变量声明语句声明了多个变量，此时会对变量进行精简，且删除变量 d 后不会影响预期的输出。与实参的精简类似，第 1 行中函数的形参也会被精简。

```
1. var foo = function(a,b) {
2.     var c = 1, d = function(){};
3.     print(c);
4. };
5. foo(1,2);
```

图 9 测试用例中参数和变量名的精简示例

（2）具有多维信息的精简判断条件

在精简测试用例时，应该保证精简后的测试用例仍然能触发原测试用例所触发的测试结果，这里的测试结果可能是缺陷也可能是误报。在尝试删除测试用例中的代码后，若精简前后测试结果没有改变，则认为删除的代码块与触发当前测试结果无关并进行删除，否则，恢复删除的代码。判断测试用例精简前后的测试结果是否改变的准

确性决定了精简后的测试用例是否正确。为了保证测试用例精简的准确性，本文设置了如表 2 所示的 6 个判断测试用例精简是否有效的条件因子。

表 2 中的适用缺陷类型表示判断条件适合于崩溃，功能缺陷和性能缺陷中的哪一种或哪几种。本文将以触发性能缺陷的测试用例精简为例进行介绍，假设一个测试用例触发了引擎 A 和引擎 B 两个引擎的性能缺陷，精简后的测试用例有且只能触发 A 和 B 的性能缺陷。若待测引擎中存在执行原始测试用例后抛出异常的引擎 C，则精简后引擎 C 必须抛出异常且关键的异常信息在精简前后没有发生变化。由于是对触发了性能缺陷的测试用例进行精简，因此存在性能缺陷的引擎 A 和引擎 B 在精简前后的性能因子的值的误差必须在合理误差范围内。

表 2 测试用例是否改变的条件因子

序号	适用缺陷类型	条件因子
1	所有缺陷	可疑的引擎数量相同。
2	所有缺陷	可疑的引擎名字一一对应。
3	所有缺陷	测试结果的缺陷类型相同。
4	所有缺陷	引擎的输出类型相同。
5	所有缺陷	引擎的关键异常信息一致
6	性能缺陷	精简前后存在性能缺陷的引擎的性能因子差距在合理误差范围内。

附注 1：输出类型分为四种，超时，崩溃，脚本异常和运行通过。

附注 2：性能因子，即引擎的执行时间与标准时间的比值，详细的定义见 3.3.2 中的性能测试。

4.2.3 测试用例精简算法

测试用例精简方法是一种在 JavaScript 的抽象语法树上尝试删除测试用例中与触发缺陷无关代码的自动简化技术，通过在语法树上删除无关节点可以保证测试用例精简过程中不引入语法错误，提高测试用例精简的效率。在尝试精简无关代码的过程中可能会触发原测试用例无法触发的新缺陷，这些新缺陷会被保存到测试结果中。详细的测试用例精简算法如下：

算法 3 测试用例精简

输入：testcase，字符串表示的 JavaScript 代码段

输出：simplifiedTestcase，newBugList，精简后的测试用例和触发了新缺陷的测试用例的集合

```
1 rootNode= generateAST(testcase)
2 newBugList = []
3 traverse(rootNode)
```

```
4 simplifiedTestcase = generateJSCode(rootNode)
5 traverse(node)
6       for key,value in node.items() do
7           if type(value) == list then
8               newBugSubset = deleteChildNode (value, key)
9               newBugList.concate(newBugSubset)
10          else if type(value) == dict then
11              traverse(value)
12          end if
13      end for
```

算法分析如下：

1-4 行：rootNode 是根据测试用例生成的 JavaScript 的 AST 树的根节点，newBugList 用于存储测试用例精简过程中触发了新缺陷的测试用例。第 3 行遍历语法树中的每一个节点。simplifiedTestcase 则是精简后的测试用例。

5-7 行：寻找当前节点 node 的子节点中可能删除的节点，当其子节点的数据类型是数组类型时，数组中的节点可能是语句，数组元素，参数，变量声明等的抽象语法树表示，可以尝试对其进行精简。

8-9 行：对同一层节点进行精简，并将精简过程中触发的新缺陷添加到 newBugList 中。在对同一层节点进行精简时，采用了逐个节点逆序进行精简的策略，优先精简执行顺序靠后或者未被执行的语句，提高测试用例精简的效率。精简过程中，如果尝试精简后的测试用例触发了与原缺陷不同的测试结果，则触发了新的可疑缺陷。

10-11 行：判断当前节点是否可能存在子节点，若当前节点可能存在子节点则递归遍历其子节点。

上面展示了一轮测试用例精简的过程，在少数情况下，一轮测试用例精简无法删除测试用例中的所有无关代码，需要反复进行多轮精简，直到无法精简为止。

4.3 基于多维特征的测试结果过滤方法

源源不断的测试用例输入到不同引擎后经过差分模糊测试会得到大量的测试结果，而这些测试结果最终都需要经过人工对比才能最终确定其是否是未知缺陷。人工参与测试结果分析时会受到这些重复测试结果的严重影响，极大的提高了发现未知缺陷的人工成本。然而测试结果复杂多样且难以理解，测试结果是否重复还需要人的主

观判断，精准去除测试结果中的重复测试结果非常困难[6]。尽可能的屏蔽重复测试结果对快速发现软件缺陷的影响是本节关注的主要内容。

4.3.1 测试结果过滤的作用

在嵌入式 JavaScript 引擎缺陷检测中，随机获取的测试用例触发的测试结果中存在大量重复测试结果，手动过滤重复测试结果会消耗大量的人工成本。这些重复的测试结果主要来源于以下三个方面。其一，当某个常用的 API 存在缺陷时，这个缺陷会被反复的触发。其二，不同引擎实现的功能差异大，这些差异会被差分模糊测试方法频繁地检测出来。功能差异大主要有两个原因，JavaScript 语言规范更新速度快，规范的不同版本对同一个 API 的定义不同，使得嵌入式 JavaScript 引擎支持的规范版本各不相同；嵌入式 JavaScript 引擎不用于 Web 浏览器环境等客观原因使得 JavaScript 未实现规范定义的所有功能；其三，为了满足调试等需求，引擎还会实现 ECMAScript-262 规范未规定的功能，如 print 函数。这些非规范的功能没有具体的规范定义，其实现的功能也不完全相同，而且还会触发大量误报。采用人工过滤测试结果的成本高昂限制了差分模糊测试的广泛应用。因此，为嵌入式 JavaScript 引擎的差分模糊测试设计一种提高未知缺陷在测试结果中的比例的测试结果自动过滤方法是非常必要且迫切的任务。

4.3.2 测试结果过滤的方法

为了解决差分模糊测试面临的测试结果分析成本高的问题，本文提出了通过去除重复测试结果实现测试结果过滤的思路。从理论上分析，如果去除重复测试结果的方法足够高效，就能极大缓解差分模糊测试结果分析成本高的问题。即使测试结果中存在大量误报，这种人工成本高的问题也可以改善，这是因为误报是一个相对有限的集合，而重复测试结果和缺陷是一个无限的集合，随着测试的不断进行，误报逐渐被发现并且被过滤，误报的数量会越来越少。理想情况下，后期的测试中就不再存在误报和重复测试结果。

图 10 展示了本文提出的基于多维特征的测试结果过滤方法，为了降低重复测试结果对引擎缺陷检测的影响，本文从测试用例及其执行结果中提取触发异常的 API，引擎的名字以及引擎抛出的异常信息等关键信息作为测试结果去重的依据。从而达到降低人工分析测试结果的成本，提高差分模糊测试系统效率的目的。下面将分别以可能存在缺陷的可疑引擎是否抛出异常的两个例子展示本文测试结果过滤时的特征提

取方法。

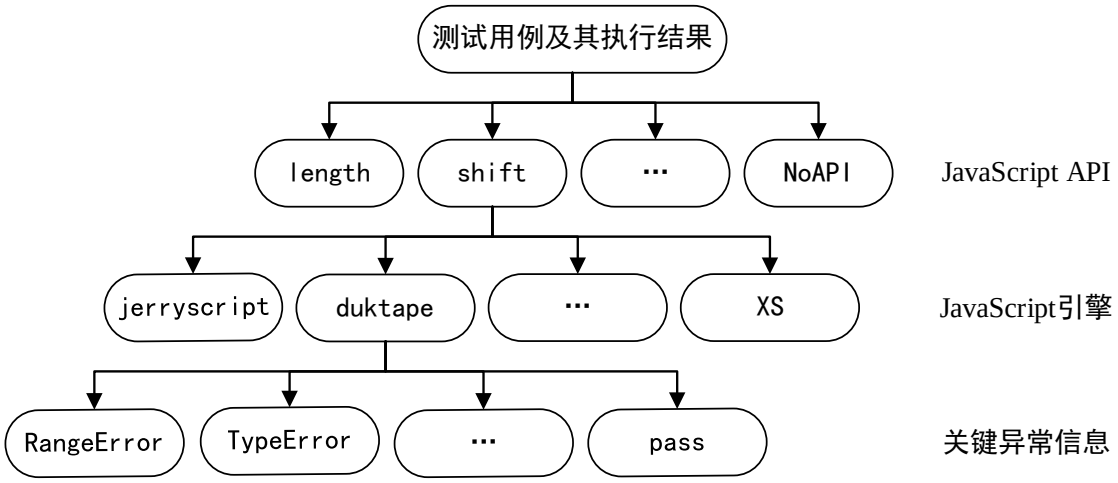

图 10 基于多维特征的测试结果过滤方法

（1）可疑引擎抛出异常：

图 11 展示了引擎 MuJS 执行的测试用例及其抛出的异常信息，支持 ES5.1 规范的 MuJS 引擎抛出 TypeError 异常，待测试引擎中的其余引擎都执行通过，由此 MuJS 被差分测试判定为存在可疑缺陷。从这些信息中可以获得三点重要信息，其一，MuJS 引擎抛出异常且可能存在缺陷，其余引擎均执行通过。其二，异常信息中可以捕获到 MuJS 抛出的异常位置在测试用例的第 2 行，结合测试用例可以获取到可能存在缺陷的 API 是 Object.keys()。其三，异常信息中非常有价值的一个关键异常信息是异常类型及其描述，即 TypeError: not an object。通过这三个重要信息标识了调用 Object.keys() 时传递的参数若不是 Object 对象时 MuJS 会抛出异常，以及只有 MuJS 抛出异常且其余引擎均执行通过这种现象。

```
1. var foo = function () {
2.     var c = Object.keys(NaN);
3. };
4. foo();
```

（a）测试用例

```
TypeError: not an object
     at Object.keys (native)
     at /tmp/javascriptTestcase_ost_r2go.js:2
     at /tmp/javascriptTestcase_ost_r2go.js:4
```

（b）MuJS 引擎抛出的异常信息

图 11 可疑引擎抛出异常的案例

（2）可疑引擎未抛出异常：

执行图 12(a)中的测试用例时，JerryScript 引擎没有抛出异常，但其他的 JavaScript 引擎都抛出了 RangeError，图 12（b）展示了其他抛出异常的引擎及其关键的异常信息。从各个引擎的执行结果与他们抛出的异常信息可以提取到三个关键信息。其一，除了 JerryScript 引擎没有抛出异常外其他引擎都抛出了异常并且 JerryScript 可能存在

缺陷。其二，结合其他引擎的异常信息描述中的测试用例异常位置与测试用例可以分析出可能存在异常的 API 是 Number.prototype.toPrecision。其三，除 JerryScript 以外的其余 5 个功能一致且抛出异常的引擎的关键异常信息。使用这三个关键信息，即可标识这种给 Number.prototype.toPrecision 传递参数越界而 JerryScript 未抛异常的现象。

```
1. var foo = function (b, c) {
2.     var d = b.toPrecision(c);
3. };
4. var p1 = 59246;
5. var p2 = function () {};
6. foo(p1, p2);
```

```
XS:       RangeError: invalid precision
Duktape:  RangeError: number outside range
Hermes:   RangeError: toPrecision argument must
          be between 1 and 100
QuickJS:  RangeError: invalid number of digits
MuJS:     RangeError: precision 0 out of range
```

(a) 测试用例　　（b） 抛出异常的引擎及其关键异常信息

图 12 可疑引擎未抛出异常的案例

在获取关键的异常信息时，会采用正则表达式的方法提取。为此，需要一个通用的正则表达式提取异常信息中的关键异常信息，但提取到的关键异常信息中包含测试用例的变量名等相关信息，若这些信息不进行标准化会严重影响测试结果的过滤效果。例如，提取到的关键信息 ReferenceError: c is not defined 中包含了与测试用例相关的变量名 c，相似的测试用例中提取到的关键信息是 ReferenceError: d is not defined。虽然异常信息完全不同，但是他们表达的含义是完全相同的，因此变量名 c 必须规范化。异常信息的描述非常多样化且每个引擎的描述都不同，为每个引擎的每种异常信息描述都写一个正则表达式实现关键信息标准化需要的人工成本高且效率低。为了快速实现关键信息的标准化，在对测试用例进行命名规范化的同时给变量名以特殊标识，在提取到关键信息时直接对特殊标识进行识别并替换即可实现标准化。

4.3.3 详细的实现流程

前面讲述了测试结果过滤的作用和过滤的方法，测试结果过滤方法部分主要介绍了基于多维特征的测试结果过滤思路。接下来将介绍从精简用例到测试结果过滤的详细处理流程。图 13 详细描述了测试结果过滤方法流程图。

（1）使用 4.2 节中描述的测试用例精简方法对原始用例进行精简后得到精简用例作为测试结果过滤的输入。

（2）规范化用例是对精简用例的变量命名统一化和格式进行规范化，使得规范化的测试用例可以使用基于哈希值的克隆检测方法去除重复测试结果，实现简单去除重复测试结果的目的。变量名统一化时，使用有标识性的名字作为变量名，便于后续步骤中对提取的关键异常信息进行标准化。

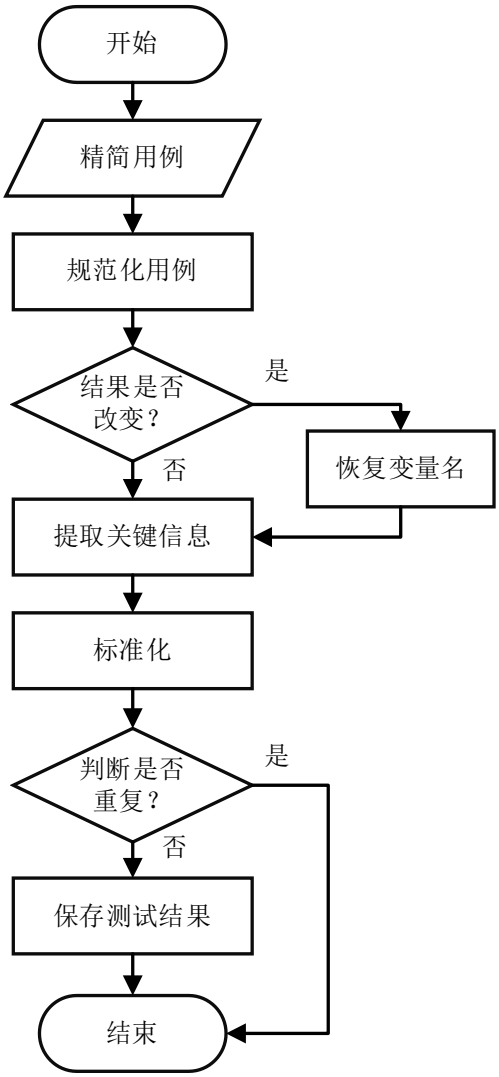

图 13 测试结果过滤方法流程图

（3）部分缺陷的触发与变量名有关系，规范化后的测试用例可能已经无法触发缺陷了，需要判断其测试结果是否改变。若测试结果发生改变，则恢复规范化前的测试用例。

（4）从所有引擎执行规范化后的测试用例的执行结果中提取引擎名称，异常的 API 以及关键的异常信息等内容，同时对异常信息进行标准化即可到用于测试结果过滤的关键特征。

（5）使用到这些关键特征对测试结果进行过滤，对比执行测试用例后引擎的异常状态，出错的 API 以及引擎抛出的异常信息等关键信息判断触发缺陷的测试结果是否重复。若当前测试结果是重复发现的测试结果，则丢弃当前测试结果。否则记录当前测试结果以便手动确认其是否触发了未知缺陷。

4.4 本章小结

本章主要介绍了差分模糊测试结果自动化处理方法。框架主要分为测试用例精简和测试结果过滤两个模块。针对随机生成的测试用例进行差分模糊测试存在测试用例包含大量无关代码和测试结果大量重复的两个问题，分别提出了基于抽象语法树的测试用例精简方法和基于多维特征的测试结果过滤方法实现测试结果的自动化处理，前者测试用例精简不仅降低了人工分析测试结果的难度和时间成本也利于后续的测试结果过滤，后者测试结果过滤去除了重复的测试结果，提高了未知缺陷在差分模糊测试结果中的比例，降低了人工发现缺陷的单位成本。

第五章 原型系统设计与实验评估

通过对嵌入式 JavaScript 引擎和差分模糊测试方法的深入分析，本文引入了嵌入式 JavaScript 引擎的差分模糊测试方法。前两章分别描述了嵌入式 JavaScript 引擎的测试思路和用例精简与结果过滤方法。本章将实现嵌入式 JavaScript 引擎的差分模糊测试原型系统 JSDiff，并使用原型系统进行实验评估，通过实验对比评估本文所提方法的有效性。

5.1 JSDiff 原型系统设计与实现

JSDiff 原型系统设计与实现将分为系统模块和系统界面两个部分。系统模块部分描述了 JSDiff 方法的系统流程和 JSDiff 原型系统的详细模块划分。系统界面部分对本文设计的原型系统的界面及其功能进行了介绍。

5.1.1 系统实现与模块设计

（1）JSDiff 方法的系统流程

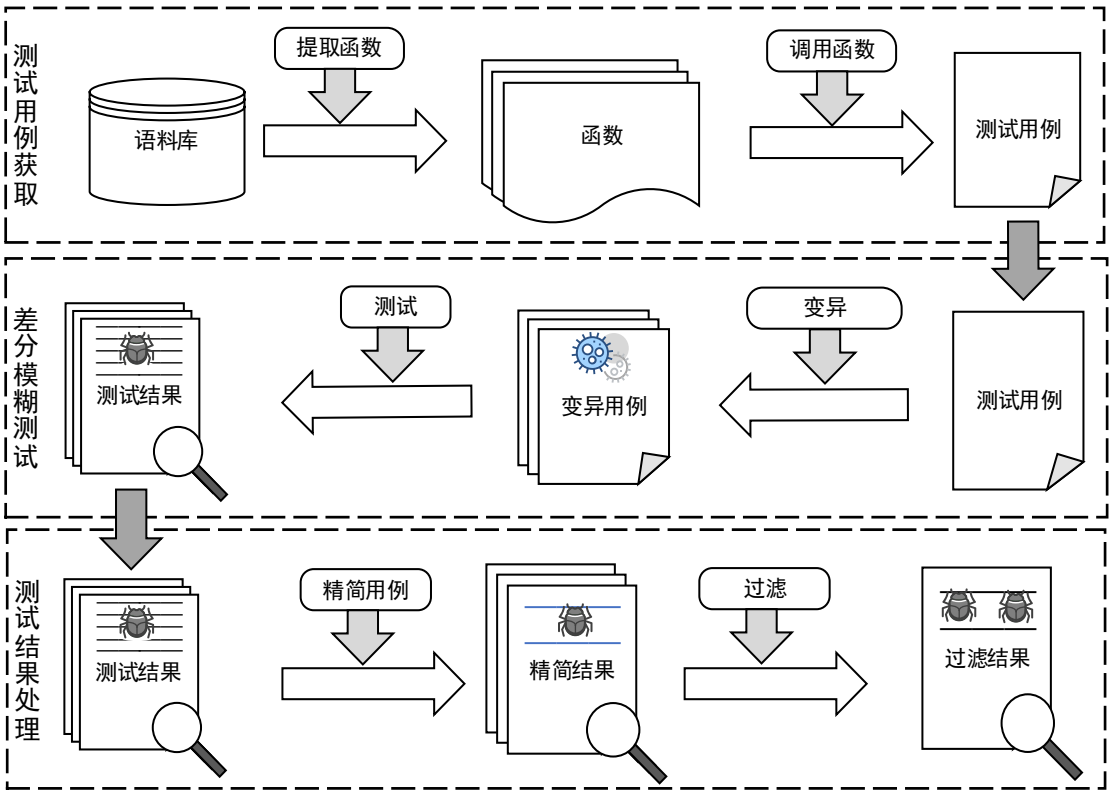

图 14 JSDiff 方法的系统流程图

JSDiff 方法的系统流程如图 14 所示。系统流程主要分为三步，首先从开源的

JavaScript 仓库中获取高质量的测试用例以确保测试用例的缺陷检测能力。然后，针对差分模糊测试检测嵌入式 JavaScript 引擎的缺陷时表现出检测能力不足的问题，设计了测试用例变异方法和和相应的差分模糊测试方法。最后，针对差分模糊测试结果分析成本高的问题提出了测试用例精简和测试结果过滤方法，降低差分模糊测试结果的分析成本。

（2）JSDiff 原型系统模块划分

图 15 展示了 JSDiff 原型系统模块划分图，根据功能的不同本文将系统划分为四个模块，分别是测试用例获取模块，差分模糊测试模块，测试结果处理模块和测试结果分析模块。测试用例获取模块负责从开源仓库中获取可以执行的测试用例并进行变异。差分模糊测试模块会执行输入的测试用例并判断测试用例是否触发了引擎的缺陷。测试结果处理模块会对差分模糊测试后触发缺陷的测试结果进行自动化处理。测试结果分析模块的任务是存储和分析测试结果。

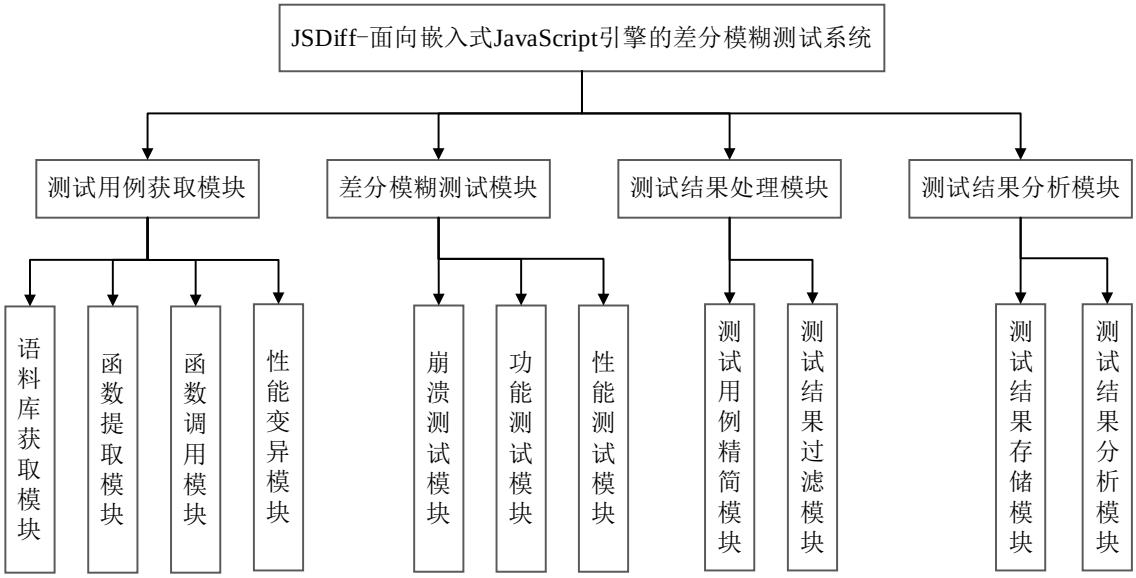

图 15 JSDiff 原型系统模块划分图

5.1.2 系统界面设计

（1）系统主界面

进入 JSDiff 系统后，首先进入系统的测试界面，如图 16 所示。在配置文件中输入存储数据的数据库地址，用户及其密码后按下回车键，待测引擎区域会出现待测引擎的详细信息，根据测试的需要可以适当增加和删除待测引擎的数量。完成上述步骤后，点击开始测试，测试进度区域会显示当前测试的进度，已运行的时间和测试用例总数等信息。

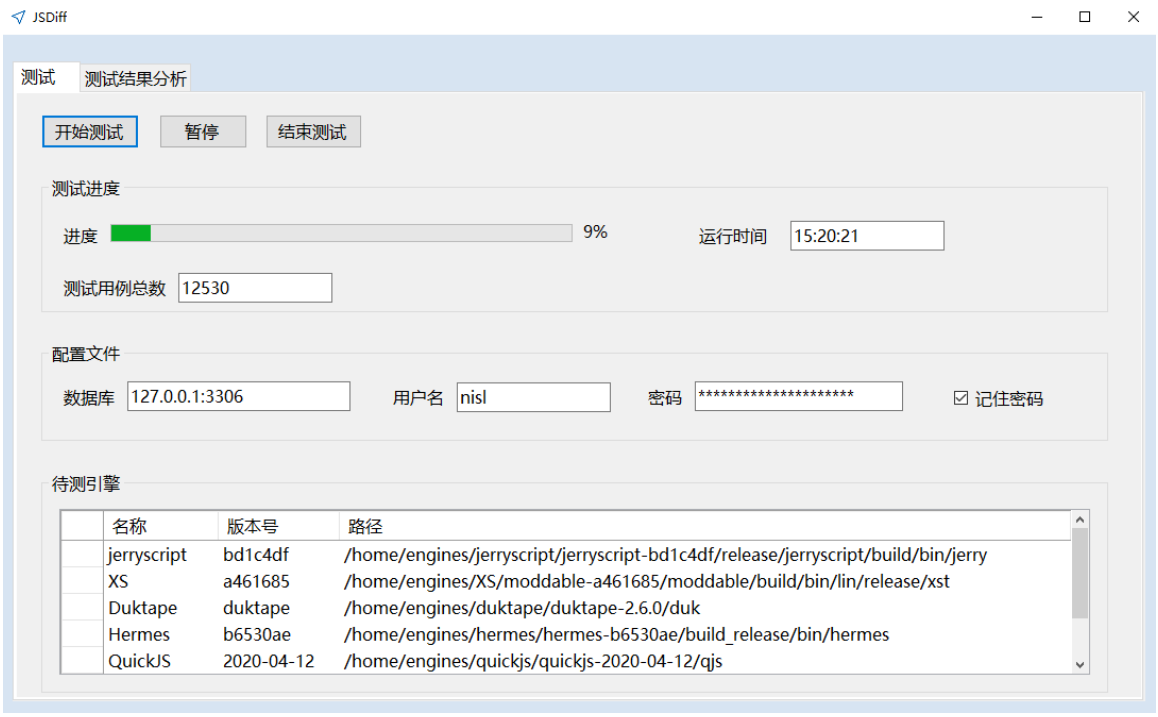

图 16 JSDiff 系统主界面

（2）测试结果分析界面

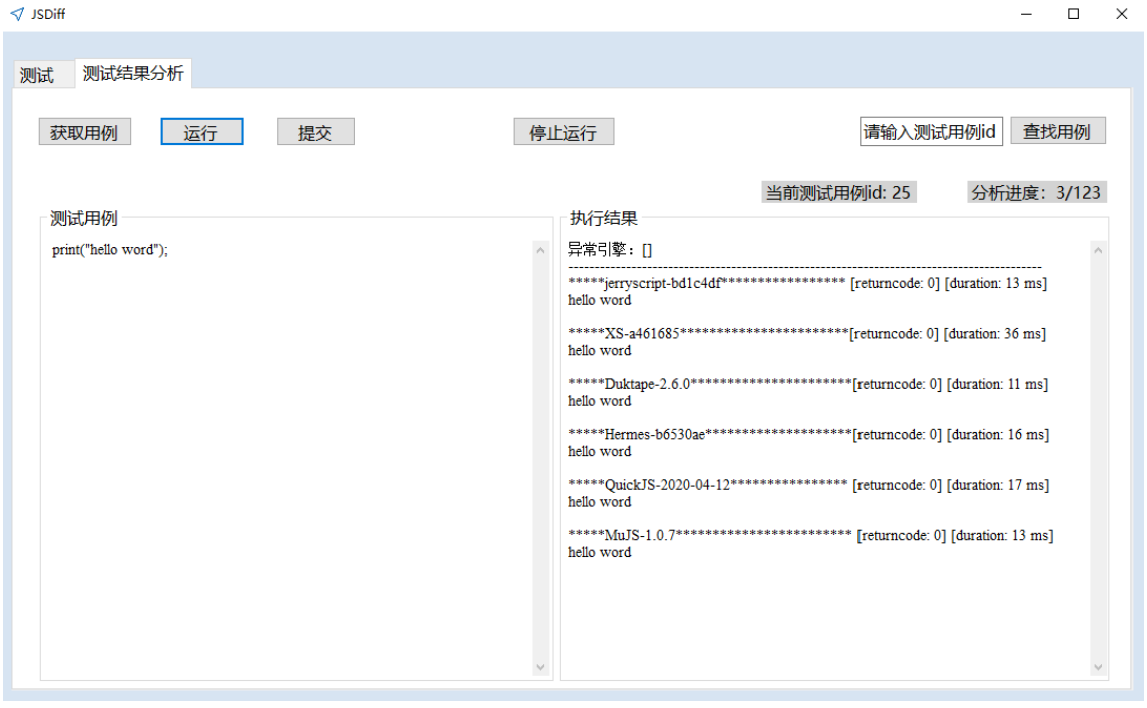

图 17 测试结果分析界面

当测试开始后，打开图 17 所示的测试结果分析界面分析测试结果。在测试结果分析界面中点击获取用例按钮，从数据库中读取触发了引擎缺陷的测试用例并在测试用例区域展示，同时还会读取各个引擎执行测试用例后的相关信息及异常的引擎名称并在执行结果区域显示。除此之外，获取测试用例时还会显示测试用例在数据库中的

唯一标识——测试用例 id，以及当前分析进度。在对测试用例区域的代码修改后点击运行按钮，各个引擎会重新执行测试用例区域的代码并在执行结果区域打印其执行结果。当前测试结果分析完成后点击提交按钮，系统会保存测试用例区域的代码并清空测试用例区域和执行结果区域的文本。当测试用例执行时间很长时，可以点击停止运行按钮停止引擎执行测试用例。最后，还可以根据测试用例 id 查找指定的测试用例，便于核查前期分析过的测试结果。

5.2 实验设计

5.2.1 实验环境和实验步骤

本文的实验环境如下：

操作系统：ubuntu 18.04，CPU：Intel(R) Core(TM) i9-9940X CPU @ 3.30GHz，软件环境：Python 3.7.6。

具体的实验步骤设计如下：

（1）JSDiff 有效性评估：使用本文设计的 JSDiff 系统对开源的嵌入式 JavaScript 引擎进行测试，并从本文提交的缺陷报告状态和不同缺陷类型的缺陷数量两个角度说明本文所提方法的有效性。

（2）测试用例生成效果评估：将 JSDiff 与相关的 5 种缺陷检测方法进行对比，在相同时间内比较本文方法生成的测试用例与相关方法生成的测试用例的缺陷检测能力。

（3）测试用例精简与测试结果过滤效果评估：首先将本文提出的测试用例精简方法与 Lithium 的测试用例精简方法进行对比，从测试用例精简的力度和精简速度两个方面评价测试用例精简算法的高效性，以测试用例精简的准确率评价测试用例精简判断条件的可靠性。然后，在不同数据集上验证测试结果过滤方法的有效性和可用性。

5.2.2 测试引擎与对比方法

（1）测试引擎

本文使用 6 个开源的轻量级的 JavaScript 引擎对 JSDiff 系统的有效性进行评估，如表 3 所示。为了验证 JSDiff 系统对嵌入式 JavaScript 引擎的缺陷检测能力，本文选择了 4 个嵌入式 JavaScript 引擎，其中包含 3 个运行在资源受限设备上的 JavaScript 引擎和 1 个运行在 Android 设备上的 JavaScript 引擎。为了证明本文方法对轻量级的 JavaScript 也有效，本文还增加了 2 个知名的轻量级 JavaScript 引擎进行测试。

表 3 待测 JavaScript 引擎

引擎名称	支持 ES 规范的版本	说明
JerryScript[25]	ES5.1 ES6(+)	最初由三星于 2014 年启动，目前由 OpenJS 基金会负责监督和维护。它是一款超轻量级的运行在资源严格受限的嵌入式 JavaScript 引擎。其不仅是三星的物联网平台 IOT.js 的一部分，还被应用于华为开源的鸿蒙系统等大型软件系统中。
XS[26]	ES2020	XS 以围绕诸如 ESP8266 或 ESP32 之类的微控制器构建的嵌入式平台为目标，其在设计时更关注如何控制引擎本身的体积和运行时内存。目前由 Moddable 对其进行维护，并作为 Moddable SDK 的一部分。
Duktape[28]	ES5.0/ES5.1 ES6(+)	具有可嵌入，可移植，以及代码和数据占用空间小的特点，能在 160kB 闪存和 64kB RAM 的平台上运行。其被应用于 Zabbix 和 low.js 等软件中。
Hermes[51]	ES6(-)	由著名的软件公司 Facebook 开源的运行在可移动设备上的 JavaScript 引擎，其专门针对运行在 Android 设备上的 React Native 应用的快速启动进行了优化。
QuickJS[52]	ES2020	一款轻量级，可嵌入的 JavaScript 引擎，其具有执行速度快，支持最新的 ES 规范等特点。
MuJS[53]	ES5.0	由著名公司 Artifex Software 开发和维护的免费开源软件，其专注于小尺寸，正确性和简单性。在实现引擎时其选择了可移植性较强的 C 语言，保证引擎的可移植性。

附注 1：(+) 表示引擎实现了规范的小部分功能。

附注 2：(-) 表示引擎实现了规范的大部分功能，但有少部分功能未实现。

（2）对比方法

本文选择了 Fuzzilli[7]，CodeAlchemist[8]，DeepSmith[14]， Montage[9]，DIE[10]等 5 个测试用例生成相关的缺陷检测方法与本文的缺陷检测方法进行对比，验证本文缺陷检测方法的有效性。

为了验证本文的测试用例精简算法的有效性以及本文提出的精简判断条件的准

确性，本文选择了 Lithium 进行比较。Lithium 的测试用例精简方法，主要用于精简触发崩溃的测试用例。其精简算法采用了按行精简，并使用二分法的策略对测试用例进行精简，在判断精简后的引擎是否仍然存在缺陷时，通过判断引擎输出中是否包含给定的关键信息而决定是否进行精简。

5.3 差分模糊测试效果评估

缺陷检测系统的目标是对真实的软件系统进行缺陷检测。首先，通过分析 JSDiff 系统所检测出的引擎缺陷数量及状态，说明本文提出的缺陷检测方法能有效检测出嵌入式 JavaScript 引擎的缺陷。其次，还统计分析本文方法检测出的缺陷类型及数量，证明本文方法对不同类型缺陷的检测能力。最后，对比分析本文的缺陷检测方法与 5 个相关方法在固定时间内检测出的缺陷数量，说明本文方法能快速高效的检测出嵌入式 JavaScript 引擎的缺陷。

5.3.1 缺陷检测结果及其状态

表 4 引擎缺陷及其状态

引擎名称	报告数量	已接受		讨论中	拒绝接受
		已修复	未修复		
MuJS	25	7	2	4	12
JerryScript	13	11	0	1	1
QuickJS	12	9	0	1	2
Hermes	9	3	5	0	1
XS	7	2	2	1	2
Duktape	5	0	2	0	3
总计	71	32	11	7	21

表 4 统计了本文方法检测出的不同引擎不同状态的缺陷数量，本文对 4 个嵌入式 JavaScript 引擎和 2 个轻量级 JavaScript 引擎进行测试。报告数量表示本文方法发现并给开发人员提交的缺陷报告的数量。本文将缺陷分为 4 种状态，已修复表示开发人员接受了本文提交的缺陷检测报告并且在接下来的版本中对此缺陷进行了修复；未修复表示开发人员认同提交的缺陷报告但还未对其进行修复；讨论中表示引擎的开发人员正在对缺陷报告进行讨论；拒绝接受表示引擎开发人员不准备对此类缺陷进行修复。

从表 4 可以看出，本文方法发现 6 个引擎共计 71 个缺陷，且待测的 6 个引擎都检测出了缺陷。其中有 43 个缺陷是开发人员明确表示其是一个有效的缺陷且对其进行了修复或即将对其进行修复，43 个已接受的缺陷中有 32 个缺陷已经被修复，11 个缺陷即将被修复。7 个缺陷由于时间因素等原因还没有被开发人员确认或拒绝。以及 21 个被开发人员拒绝的缺陷。

被拒绝的 21 个缺陷报告中，拒绝接受的主要原因是这些引擎在实现 ECMAScript-262 规范时，没有实现规范规定的所有功能，尤其是规范中为了保证 JavaScript 语言兼容性的附录 B 中定义的功能。图 18 展示了一个被开发者拒绝的缺陷的测试用例[54]。按照 ECMAScript-262 规范的定义，测试用例第 3 行定义的函数 r 可以在其定义所在代码块以外进行正常访问，而 XS 在执行测试用例第 5 行代码时抛出了 ReferenceError。XS 拒绝接受此缺陷给出的解释是此功能是附录 B 中定义的功能，其只选择性的实现了附录 B 中的部分功能，此功能不在其实现的范围内。

从上述结论中可以看出，本文方法检测出了所有待测引擎的缺陷，提交的 71 个缺陷报告中有 43 个缺陷得到了开发人员的修复或确认，说明了本文提出的方法在真实的环境中检测 JavaScript 引擎缺陷的能力，并且检测出来的缺陷是有价值的。

```
1. var foo = function () {
2.     {
3.         function r() {}
4.     }
5.     print(r);
6. };
7. foo();
```

图 18 开发者拒绝的缺陷的测试用例示例

5.3.2 不同缺陷类型的对比

图 19 展示了本文方法检测出的不同引擎的不同类型的缺陷数量。本文将引擎的缺陷分为崩溃、功能缺陷和性能缺陷。崩溃是引擎执行测试用例时造成引擎自身的出现了崩溃，这种缺陷可能被黑客利用并进行安全攻击。功能缺陷是引擎没有按照 JavaScript 语言规范实现其功能，引擎表现为执行结果不正确。性能缺陷则是引擎实现了相关功能，但算法不够高效或编码错误导致的引擎执行效率低的缺陷。

JSDiff 系统发现了 6 个引擎 3 种类型的缺陷，发现最多的缺陷类型是功能缺陷，共计 50 个。其次是性能缺陷，共计 16 个。发现数量最少的缺陷类型是崩溃，共计 5 个。发现崩溃的数量比较少的原因是由于收集的测试用例来自于 GitHub 仓库的手写

代码，手写代码通常都是比较规范的代码且本文没有针对崩溃测试专门设计测试用例变异算法，而畸形的测试用例通常会更容易触发软件的崩溃。

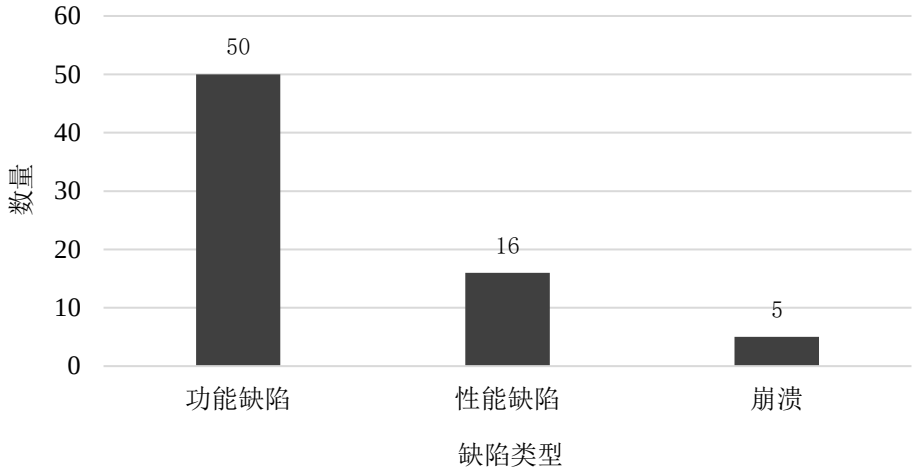

图 19 引擎缺陷及其缺陷类型

5.3.3 与相关缺陷检测方法的对比

为了评估本文所提缺陷检测方法 JSDiff 的缺陷检测能力，本文选择了 5 种较新的缺陷检测方法，对 3 个 JavaScript 引擎进行为期 72 小时的测试。这 3 个 JavaScript 引擎中包含 2 个嵌入式 JavaScript 引擎和 1 个轻量级 JavaScript 引擎。通过手动分析差分模糊测试结果，得到了如表 5 所示的缺陷检测结果。

表 5 不同缺陷检测方法检测出的缺陷数量

缺陷检测方法	功能缺陷	性能缺陷	崩溃	总计
JSDiff	7	3	0	10
DIE	6	0	0	6
Montage	6	0	0	6
DeepSmith	3	0	0	3
Fuzzilli	2	0	0	2
CodeAlchemist	2	0	0	2

从表 5 中可以看出所有的方法都检测出了引擎的缺陷，其中本文方法 JSDiff 检测的缺陷数量最多，缺陷数量为 10 个。从不同缺陷检测方法发现的缺陷类型及数量可以看出，3 类缺陷中最容易检测的缺陷是功能缺陷，并且所有缺陷检测方法都检测出了引擎的功能缺陷。然而，这些缺陷检测方法中只有本文的方法检出了性能缺陷，这是因为本文方法生成的测试用例专门针性能测试进行了变异，而其余方法生成的测

试用例没有进行相应的性能变异。这也说明了本文针对测试用例进行性能变异是有必要的。对于崩溃，所有的方法都未能发现引擎的崩溃，即使是擅长于检测安全缺陷的方法也未能发现崩溃，这是因为引擎崩溃相比于功能缺陷更难触发，测试引擎崩溃通常需要更长的周期。

5.3.4 测试用例生成质量评估

为了解决测试用例有限的问题，本文尝试使用深度学习算法生成测试用例实现持续测试。在评估深度学习算法生成的测试用例的质量时，本文从测试用例的语法通过率，函数覆盖率，分支覆盖率和语句覆盖率四个角度对生成的测试用例进行评估。测试用例的语法通过率衡量深度学习算法学习 JavaScript 语法结构信息的能力，覆盖率则衡量测试用例的缺陷检测能力，因为覆盖率越高的测试用例触发引擎缺陷的可能性会更高。生成测试用例的深度学习算法分别使用了长短期记忆网络（LSTM）和擅长机器翻译的 Transformer，训练数据是本文从 GitHub 仓库中提取的函数。

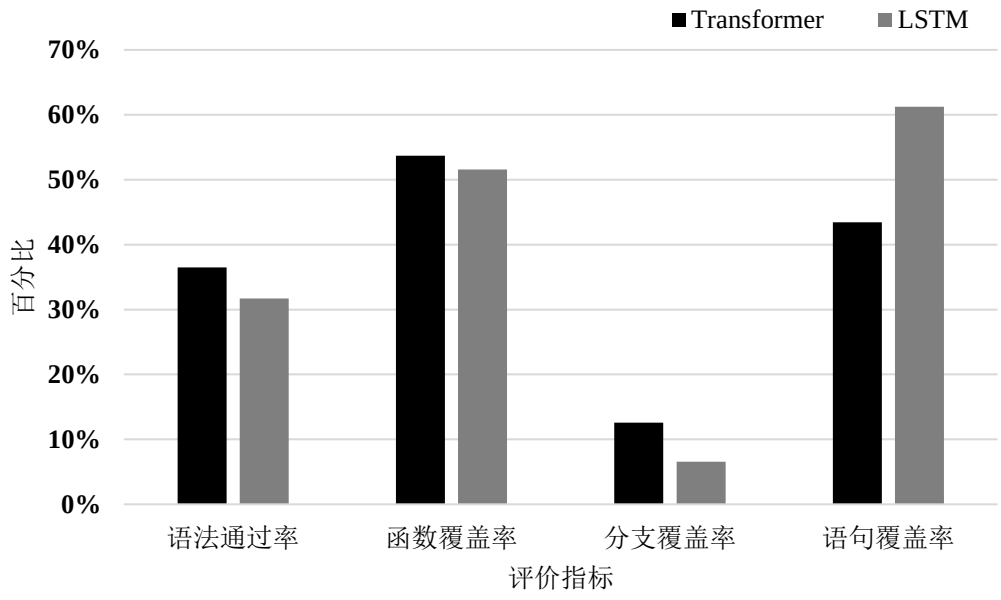

图 20 深度学习算法生成的测试用例质量

深度学习算法生成的测试用例质量如图 20 所示。从图中可以看出 Transformer 生成的测试用例的语法通过率更高，说明 Transformer 学习 JavaScript 语法结构信息的能力更强。在覆盖率方面，Transformer 生成的测试用例在函数覆盖率和分支覆盖率方面都优于 LSTM 生成的测试用例，LSTM 生成的测试用例在语句覆盖率方面则高于 Transformer，这是由于 Transformer 生成的测试用例更复杂，处于条件分支中的语句更多。综合上述四个测试用例的评价指标可以看出，Transformer 生成的测试用例质量

比 LSTM 生成的测试用例质量更高。

5.4 用例精简与结果过滤效果评估

本节分别对测试用例精简和测试过滤效果进行评估。在评估测试用例精简方法时，选择基于代码行的测试用例精简方法 Lithium 与本文方法进行对比，以精简的正确率，精简的力度和精简的速度三个指标对精简算法和精简判断条件进行评估。在评价测试结果的过滤效果时，选择了 5 个不同数据集对本文的测试结果过滤效果进行评估，说明测试结果过滤方法不仅对本文方法有效，对其他的测试用例生成方法也同样有效。

5.4.1 测试用例精简效果评估

通过与基于代码行的测试用例精简方法 Lithium 进行对比，一方面，验证本文提出的判断测试用例精简是否有效的精简判断条件的可靠性，另一方面，验证测试用例精简算法的高效性和精简的力度。另外，还对比了删除同一层节点时采用不同的精简策略的平均精简时间。评估测试用例精简效果时，使用了 966 个触发了可疑缺陷的测试用例作为测试集。判断测试用例精简是否正确时，以 50%的抽样率从测试集中进行随机抽样并手动进行检查，判断测试用例精简是否正确，手动判断的标准是精简前后触发的缺陷或误报相同。

表 6 不同测试用例精简方法的精简效果

精简方法	原始用例平均 token 数	精简用例平均剩余 token 数	平均精简率	平均精简时间（s）	准确率
Lithium-output	410	216	47.3%	14.7	87.3%
Lithium-multi	410	111	72.9%	41.2	99.5%
JSDiff-binary	410	71	82.7%	39.7	99.5%
JSDiff-reverse	410	71	82.7%	38.1	99.5%

表 6 中的 Lithium-output 表示的是使用 Lithium 的基于代码行的精简算法和根据输出中是否包含给定的关键信息而决定是否进行精简的精简判断条件。Lithium-multi 表示的是使用 Lithium 的基于代码行的精简算法和本文提出的具有多维信息的精简判断条件。JSDiff-reverse 表示的是使用本文的基于抽象语法树的测试用例精简算法，在同一层节点的精简过程中，使用逐个节点逆序进行精简的策略，精简判断条件使用的是本文提出的具有多维信息的精简判断条件。与 JSDiff-reverse 的不同之处在于，JSDiff-binary 在精简同一层节点时采用 Lithium 中提出的二分精简算法。

从表 6 观察到，从 Lithium-output 和 Lithium-multi 两种测试用例精简方法的精简准确率可以看出，当精简算法相同，精简判断条件不同时，使用 Lithium 的精简判断条件进行精简的准确率只有 87.3%，而具有多维信息的精简判断条件则能将测试用例精简的准确率提升至 99.5%。由此说明 Lithium 只根据输出信息中是否包含给定的关键信息决定精简是否正确是不可靠的。Lithium 提出的精简判断条件不可靠的原因是其精简判断条件主要适用于精简触发引擎崩溃的测试用例，没有针对触发功能缺陷和性能缺陷的测试用例精简问题提出有效的精简判断条件。通过对比本文的精简判断条件与 Lithium 使用的精简判断条件，证明了本文提出的具有多维信息的精简判断条件可以提高测试用例精简的正确性。

当精简判断条件一致时，比较不同精简算法精简的精简速度和精简力度。表 6 对比了 Lithium-multi，JSDiff-reverse，JSDiff-binary 三种精简方法的平均精简率和平均精简时间。在精简速度方面，基于抽象语法树并在精简同一层节点时使用逐个节点逆序进行精简的策略进行精简的平均精简速度最快，其平均精简时间为 38.1（s）。同一层节点使用二分的策略进行精简的速度稍微慢，其主要原因是二分的策略更擅长于精简可精简程度较高的测试用例。在精简力度方面，基于抽象语法树的精简方法（JSDiff-reverse 或 JSDiff-binary）的精简力度最大，其在基于代码行的精简方法（Lithium-multi）的基础上提升了 36%。基于代码行对测试用例进行精简的精简速度最慢，且精简力度最小。由此说明，基于抽象语法树的测试用例精简方法不仅可以提升测试用例精简的速度和精简的彻底程度，而且在抽象语法树的同一层节点使用逐个节点逆序精简的策略进行精简时不仅精简速度最快且精简最彻底。

通过上述实验可以看出，一方面，本文提出的具有多维信息的精简判断条件能更准确地判断测试用例精简前后的测试结果是否改变，提高了测试用例精简的正确性。另一方面，本文提出的基于抽象语法树的测试用例精简方法能更快，更彻底地对测试用例进行精简，而且在对同一层节点进行精简时使用逐个节点逆序精简的策略能更快速实现测试用例的精简。

5.4.2 测试结果过滤效果评估

为了评估本文提出的测试结果过滤方法的有效性，将使用不同缺陷检测方法触发的测试结果作为测试集。由于本文提出的基于多维特征的测试结果过滤方法主要针对部分引擎抛出异常而部分引擎不抛出异常的测试结果进行过滤，而且这类测试结果占据本文全部测试结果的 80%，因此本文用于评估测试结果过滤方法有效性的数据集

中只包含部分引擎抛出异常而部分引擎未抛出异常的测试结果。

本文使用了 5 种不同的数据集对测试结果过滤效果进行评估，实验结果如图 21 所示。5 个数据集各有 6000 个样本，使用基于多维特征的测试结果过滤方法对不同数据集进行过滤。从图 21 可以看出，5 个数据集的过滤比例都大于 85%，其不仅说明了本文的测试结果过滤方法对本文生成的测试用例和相关缺陷检测方法生成的测试用例触发的测试结果都有效，还说明了对 5 种缺陷检测方法生成的测试用例触发的测试结果进行过滤的必要性。5 种缺陷检测方法生成的测试用例触发的测试结果重率都高的主要原因来源于两方面，首先，不同 JavaScript 引擎实现的功能差异较大，差分模糊测试触发引擎误报的可能性高。其次，生成的大量随机测试用例会反复触发相同的误报或缺陷。

通过在不同数据集上对测试结果过滤方法进行评估，实验表明，本文提出的测试结果过滤方法不仅能过滤本文方法生成的测试用例触发的重复测试结果，对其余方法生成的测试用例触发的重复测试结果也有非常可观的过滤效果。

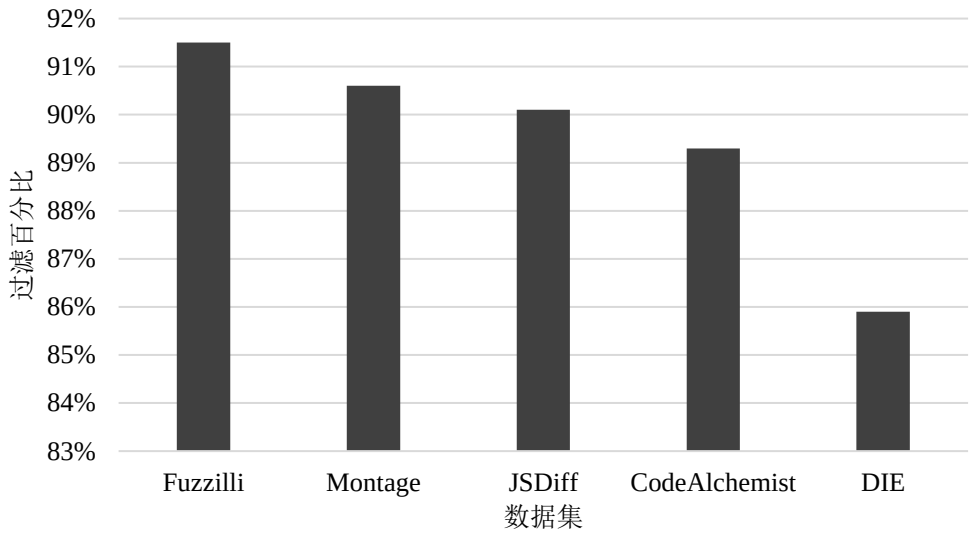

图 21 不同数据集的过滤效果

5.5 案例分析

为了展示本文提出的缺陷检测方法的有效性，本节将对本文的缺陷检测方法检测出的三种不同类型的缺陷案例分别进行剖析，结合实际例子介绍本文缺陷检测方法检测缺陷的过程并介绍本文方法能检测相应缺陷的原因。

5.5.1 性能缺陷案例分析

图 22（a）展示了一个同时检测出两个引擎性能缺陷[55,56]的测试用例，这两个引

擎分别是运行在 Android 设备上的引擎 Hermes 的 3826084 版本以及运行在资源受限设备上的引擎 XS 的 a461685 版本。图 22（a）中触发性能缺陷的测试用例在精简后只有 8 行代码，测试用例中 1-6 行定义了一个函数 foo，函数中的第 2 行创建一个长度为 arrLen 的数组，函数中的第 3-5 行逆序给数组的每一个元素赋值为 0。

图 22（b）展示了使用待测的 6 个 JavaScript 引擎执行测试用例所花费的时间。在执行这个测试用例时，执行速度最快的引擎是 QuickJS，只用了 29 毫秒的时间。执行速度非常慢的两个引擎分别是 XS 和 Hermes，其执行时间分别是 QuickJS 执行时间的 431 倍和 237 倍，因此，XS 和 Hermes 两个引擎被判定为存在严重性能缺陷。

为了节约内存空间，测试用例中的第 2 行创建数组时 JavaScript 引擎通常都不会一次性分配内存空间，而是使用动态数组，即只有数组中需要存放元素时才分配足够存放数组元素大小的内存块，若已分配的内存不足以存放数组中的所有元素时会对数组容量进行扩容。触发这两个引擎的性能缺陷需要满足的必要条件之一是测试用例中的第 7 行的变量 p 要足够大，才足以让引擎在执行的过程中不断地扩容，只有不断地进行扩容才能检测数组扩容时所采用的算法是否高效。触发 Hermes 的性能缺陷还需要满足的必要条件是逆序给数组赋值，其原因是 Hermes 正序和逆序给数组扩容时采取的策略是不同的，且逆序扩容采用的算法效率更低效。

在函数调用并传递参数的过程中，经过类型推断方法推断出参数值是数值类型并传递一个数值类型的变量 p，若测试用例中的变量 p 的初始值较小，测试用例将无法触发引擎的性能缺陷。由于测试用例中存在 while 循环，触发了对循环控制变量进行变异的条件，当 p 值变异为 50000 时触发了引擎 Hermes 和引擎 XS 的性能缺陷。这个测试用例触发的缺陷也证明了本文的类型推断和针对循环控制条件变异方法的重要性和有效性。

```
1. var foo = function(arrLen) {
2.     var arr = new Array(arrLen);
3.     while (arrLen--){
4.         arr[arrLen] = 0;
5.     }
6. };
7. var p = 50000;
8. foo(p);
```

（a）测试用例

```
JerryScript:        33 ms
XS:              12501 ms
Duktape:            78 ms
Hermes:           6863 ms
QuickJS:            29 ms
MuJS:               93 ms
```

（b）各引擎的执行时间

图 22 性能缺陷案例

5.5.2 功能缺陷案例分析

如图 23 所示的测用例触发了引擎 Duktape 的 2.6.0 版本的功能缺陷[57]，执行此测

试用例时除了 Duktape 的输出结果是 0 以外，其余引擎的输出结果都是 1。测试用例中的 1-4 行定义了一个函数 foo，第 5-6 行调用函数 foo 并打印函数的返回值。函数中的第 2 行定义了一个变量 a 并赋值为 4，第 3 行代码中的符号“>>>”表示无符号右移操作，如 4 >>> 2 是将 4 的二进制数向右移动 2 位，其结果应该是 1。

在执行测试用例的第 3 行代码时，引擎 Duktape 先执行“>>>”右边括号中的赋值操作 a = 2，随后第二行中定义的变量 a 的值变为 2，此时运算符“>>>”右边的值是 2，“>>>”左边的值共享了其右边赋值操作对变量 a 的修改结果，即“>>>”左边的值也被修改为 2，最后第三行代码简化为计算 2 >>> 2 的值，因此 Duktape 执行测试用例后的输出为 0。然而，正确的执行顺序应该是将“>>>”左边变量的初始值 4 拷贝一份存入寄存器，再执行操作 a = 2，此时对变量 a 的修改结果不会影响运算符“>>>”左边已经存入寄存器的值，最终第 3 行演变为计算 4 >>> 2 的执行结果，因此正确的输出结果应该为 1。此测试用例中，无符号右移操作中加入其他复杂的操作，严重影响了程序的可读性，此编码方式不是常规的编码，不符合良好的编码规范。此缺陷的发现也印证了本文的测试用例生成方法具有功能复杂，语义丰富的特点。

```
1. var foo = function () {
2.     var a = 4;
3.     return a >>> (a = 2);
4. };
5. var res = foo();
6. print(res);
```

图 23 功能缺陷案例

5.5.3 崩溃案例分析

图 24 展示了触发了 JerryScript 引擎崩溃[58]的测试用例[59]的部分代码。JerryScript 引擎执行测试用例时，JSDiff 捕获到了操作系统发出的 SIGSEGV 信号，因此判定引擎崩溃了。此缺陷是安全相关的缺陷，因此该缺陷在提交后的 2 个小时内被修复了。

触发此缺陷的测试用例是一个拥有 4502 行的 JavaScript 代码段，使用本文基于抽象语法树的测试用例精简方法成功地将其精简为只有 484 行的测试用例，为测试人员分析此缺陷节约了大量的时间。触发崩溃的原因是测试用例的函数 foo 中定义了超过 128 个函数或变量，JerryScript 在将含有 128 个变量或函数的测试用例编译成字节码的过程中，由于编码错误导致了访问越界的问题。本文方法能检测出此缺陷的原因是由于测试用例是 GitHub 开源仓库中提取的函数组装而成的，开源仓库中的 JavaScript 代码的使用范围不仅限于嵌入式设备，而此测试用例中的函数来源于复杂

的 Web 应用程序中的 JavaScript 代码片段，其定义了超过 128 个函数或变量，因此检测出了 JerryScript 内部数组访问越界导致的崩溃。

```
  1. var foo = function () {
  2.     var hookCallback;
  3.     function hooks() {
                    …
                    …
482.     proto.parsingFlags = parsingFlags;
483. };
484. var res = foo();
```

图 24 崩溃案例

5.6 本章小结

本章首先介绍了根据本文所提方法实现的原型系统，并描述了原型系统的系统模块设计与实现和系统界面。然后，根据实现的原型系统对本文提出的方法进行多维度的试验评估，实验评估主要围绕差分模糊测试和用例精简与结果过滤进行。在对差分模糊测试效果进行评估时，使用 6 个 JavaScript 引擎进行测试，共计发现 71 个缺陷，其中确认 43 个。在固定的时间内与相关的缺陷检测方法进行对比，本文提出的方法不仅具有高效检测嵌入式 JavaScript 引擎性能缺陷的能力，还具有检测嵌入式 JavaScript 引擎功能缺陷和崩溃的能力。在对测试用例精简效果与结果过滤效果进行评估时，实验结果表明本文提出的测试用例精简方法能更快速且更准确地对测试用例进行精简，与此同时，本文提出的测试结果过滤方法不仅能有效过滤本文方法触发的重复测试结果，对相关方法生成的测试用例触发的重复测试结果也具有良好的过滤效果。

总结与展望

总结

随着物联网发展的需要，JavaScript 语言凭借网络编程的优势逐渐成为嵌入式设备的主流编程语言，因此社区也开发了许多优秀的嵌入式 JavaScript 引擎。开发人员水平参差不齐等因素会导致嵌入式 JavaScript 引擎存在安全缺陷、功能缺陷及性能缺陷等问题。引擎的安全缺陷会使嵌入式设备面临安全风险，引擎的功能缺陷会影响 JavaScript 程序的正确运行，引擎的性能缺陷不仅会白白增加计算资源匮乏的嵌入式设备的计算量，还会严重增加低功耗嵌入式设备的能耗。本文通过对现有缺陷检测方法进行分析，提出了改进的差分模糊技术并对嵌入式 JavaScript 引擎的性能缺陷进行检测的方法。本文提出的方法在对嵌入式 JavaScript 引擎进行测试时不仅在性能缺陷检测方面效果可观，在安全缺陷检测和功能缺陷检测方面也取得了良好的效果。除了缺陷检测，本文还针对差分模糊测试结果分析成本高的问题提出了测试用例精简与测试结果过滤方法。在面向嵌入式 JavaScript 引擎的差分模糊测试方法基础上实现了 JSDiff 原型系统，并使用该系统对本文方法的有效性进行评估。

本文的主要研究内容总结如下：

（1）研究软件的缺陷检测方法，对目前的缺陷检测方法进行研究，分析已有缺陷检测方法的优缺点。深入分析现有的差分模糊测试方法从测试用例生成，差分模糊测试，到测试结果处理过程中存在的问题，并总结了现有差分模糊测试方法无法检测性能缺陷的主要原因。

（2）研究以性能缺陷检测为导向的差分模糊测试方法，通过分析嵌入式 JavaScript 引擎的特点和现有方法存在的不足，设计了面向嵌入式 JavaScript 引擎的缺陷检测方法。本文从现有的开源代码库中获取语法正确且语义相对完整的函数，对提取的函数进行调用并传递相应数据类型的参数，同时还针对测试用例性能缺陷检测能力不足的问题提出了测试用例变异方法。最后，改进现有的差分模糊测试方法，使其不仅可以检测嵌入式 JavaScript 引擎的功能缺陷和安全缺陷，还能检测嵌入式 JavaScript 引擎的性能缺陷。

（3）研究以高精度为导向的用例精简与结果过滤方法，分析了差分模糊测试结

果分析成本高的原因，并针对导致差分模糊测试结果分析成本高的测试用例复杂和测试结果重复率高的两个问题，分别提出了基于抽象语法树的测试用例精简方法和基于多维特征的测试结果过滤方法。通过测试用例精简和测试结果过滤降低分析未知缺陷的单位成本。

（4）设计并实现 JSDiff 原型系统，并使用 JSDiff 原型系统进行实验评估。选择多个嵌入式 JavaScript 引擎和多个缺陷检测方法进行对比实验，验证本文缺陷检测方法的缺陷检测能力。同时，还选择了不同测试用例精简方法和不同数据集分别对本文的测试用例精简方法和测试结果过滤方法进行对比分析，通过实验说明测试用例精简方法和测试结果过滤方法的有效性。

展望

本文提出的面向嵌入式 JavaScript 引擎的差分模糊测试方法虽然可以检测出嵌入式 JavaScript 引擎的缺陷，但目前仍然存在可提升的空间，具体总结为以下三个方面。

（1）目前测试用例生成主要来源于 GitHub 开源项目中的 JavaScript 代码，这种方式获取的测试用例虽然具有语法正确和语义丰富的特点，但有限的测试用例难以维持长期的测试。后续会尝试使用更先进的深度学习算法有指导性地生成高质量测试用例实现持续测试。

（2）在进行性能测试时增加单条语句的执行次数虽然能暴露出引擎的性能缺陷，但这种方式也增加了测试时的计算量，后续工作中将有针对性的对容易存在性能缺陷的操作进行变异，减少计算资源的消耗。

（3）在对测试结果进行过滤时，本文的过滤方法主要针对测试结果中数量最多的待测引擎执行测试用例后会抛出异常这一类测试结果进行过滤，而所有引擎都执行通过且没有报错的这种测试结果，目前只能依靠测试用例精简和变量名替换的方式进行过滤，但过滤的力度有限。后续将尝试结合引擎内部的执行路径，内存使用情况等信息对测试结果进行过滤。

参考文献

[1] 王越, 孙亮, 王轶骏, 薛质. 一种针对 JavaScript 引擎 JIT 编译器的模糊测试方法[J]. 通信技术, 2021, 54(01):175-180.

[2] Hodován R, Kiss Á. Fuzzing javascript engine apis[C]. International Conference on Integrated Formal Methods. Springer, Cham, 2016: 425-438.

[3] 余启洋. 嵌入式 JavaScript 引擎并行化研究与设计[D]. 电子科技大学, 2013.

[4] Lithium - Line-based testcase reducer [DB/OL]. https://github.com/MozillaSecurity/Lithium

[5] Groce A, Holzmann G, Joshi R. Randomized differential testing as a prelude to formal verification[C]. 29th International Conference on Software Engineering (ICSE'07). IEEE, 2007: 621-631.

[6] Chen Y, Groce A, Zhang C, et al. Taming compiler fuzzers[C]. Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. 2013: 197-208.

[7] Groß S. Fuzzil: Coverage guided fuzzing for javascript engines[D]. Master's thesis, TU Braunschweig, 2018.

[8] Han H S, Oh D H, Cha S K. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines[C]. NDSS. 2019.

[9] Lee S, Han H S, Cha S K, et al. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer[C]. 29th USENIX Security Symposium. 2020: 2613-2630.

[10] Park S, Xu W, Yun I, et al. Fuzzing JavaScript Engines with Aspect-preserving Mutation[C]. 2020 IEEE Symposium on Security and Privacy (S&P). IEEE, 2020: 1629-1642.

[11] Test262 - Official ECMAScript Conformance Test Suite [DB/OL]. https://github.com/tc39/test262

[12] Chen Y, Su T, Sun C, et al. Coverage-directed differential testing of JVM implementations[C]. proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2016: 85-99.

[13] Yang X, Chen Y, Eide E, et al. Finding and understanding bugs in C compilers[C]. Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. 2011: 283-294.

[14] Cummins C, Petoumenos P, Murray A, et al. Compiler fuzzing through deep learning[C]. Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2018: 95-105.

[15] 赵芊. 物联网操作系统中性能及安全缺陷检测工具的研究和实现[D]. 北京邮电大学, 2016.

[16] Perl S E, Weihl W E. Performance assertion checking[C]. Proceedings of the fourteenth ACM symposium on Operating systems principles. 1993: 134-145.

[17] Subraya B M, Subrahmanya S V. Object driven performance testing of Web applications[C]. Proceedings First Asia-Pacific Conference on Quality Software. IEEE, 2000: 17-26.

[18] Zaman S, Adams B, Hassan A E. Security versus performance bugs: a case study on firefox[C].

Proceedings of the 8th working conference on mining software repositories. 2011: 93-102.

[19] Jin G, Song L, Shi X, et al. Understanding and detecting real-world performance bugs[J]. ACM SIGPLAN Notices, 2012, 47(6): 77-88.

[20] Nistor A, Ravindranath L. Suncat: Helping developers understand and predict performance problems in smartphone applications[C]. Proceedings of the 2014 International Symposium on Software Testing and Analysis. 2014: 282-292.

[21] Nistor A, Chang P C, Radoi C, et al. Caramel: Detecting and fixing performance problems that have non-intrusive fixes[C]. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE, 2015, 1: 902-912.

[22] Stack Overflow developer survey results [EB/OL]. https://insights.stackoverflow.com/survey/2020

[23] The 2020 state of the octoverse [EB/OL]. https://octoverse.github.com/

[24] Jaimini U , Dhaniwala M . JavaScript empowered Internet of Things[C]. International Conference on Computing for Sustainable Global Development. IEEE, 2016: 2373-2377.

[25] JerryScript - Ultra-lightweight JavaScript engine for the Internet of Things [DB/OL]. https://github.com/jerryscript-project/jerryscript

[26] XS - An JavaScript engine running on resource-constrained devices [DB/OL]. https://github.com/Moddable-OpenSource/moddable

[27] Grunert K . Overview of JavaScript Engines for Resource-Constrained Microcontrollers[C]. 2020 5th International Conference on Smart and Sustainable Technologies (SpliTech). 2020: 1-7.

[28] Duktape - An embeddable Javascript engine focus on portability and compact footprint [EB/OL]. https://duktape.org/

[29] Ukil A , Sen J , Koilakonda S . Embedded security for Internet of Things[C]. Emerging Trends & Applications in Computer Science. IEEE, 2011: 1-6.

[30] Antonakakis M, April T, Bailey M, et al. Understanding the mirai botnet[C]. 26th USENIX Security Symposium. 2017: 1093-1110.

[31] Muench M, Stijohann J, Kargl F, et al. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices[C]. NDSS. 2018.

[32] Guo T, Zhang P, Wang X, et al. Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation[C]. 2013 Second International Conference on Informatics & Applications (ICIA). IEEE, 2013: 212-215.

[33] Holler C, Herzig K, Zeller A. Fuzzing with code fragments[C]. 21st USENIX Security Symposium. 2012: 445-458.

[34] 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.

[35] Jones D. Trinity: A Linux system call fuzz tester[EB/OL]. http://codemonkey.org.uk/projects/trinity/.

[36] Jääskelä E. Genetic algorithm in code coverage guided fuzz testing[D]. University of Oulu, 2016.

[37] Godefroid P, Kiezun A, Levin M Y. Grammar-based whitebox fuzzing[C]. Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2008: 206-

215.

[38] McKeeman W M. Differential testing for software[J]. Digital Technical Journal, 1998, 10(1): 100-107.

[39] Sheridan F. Practical testing of a C99 compiler using output comparison[J]. Software: Practice and Experience, 2007, 37(14): 1475-1488.

[40] Evans R B, Savoia A. Differential testing: a new approach to change detection[C]. The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers. 2007: 549-552.

[41] Le V, Afshari M, Su Z. Compiler validation via equivalence modulo inputs[J]. ACM SIGPLAN Notices, 2014, 49(6): 216-226.

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

[43] Liang H, Pei X, Jia X, et al. Fuzzing: State of the art[J]. IEEE Transactions on Reliability, 2018, 67(3): 1199-1218.

[44] Research Insights Volume 9—Modern Security Vulnerability Discovery [EB/OL]. https://www.nccgroup.com/jp/our-research/research-insights-vol-9-modern-security-vulnerability-discovery/

[45] AFL - A coverage-guided fuzzer [EB/OL]. Available: http://lcamtuf. coredump.cx/afl/

[46] 曹帅. 基于类型推断的 JavaScript 引擎模糊测试方法研究[D]. 西北大学, 2020.

[47] Time.js - High-precision JavaScript timer [EB/OL]. https://github.com/eligrey/timer.js

[48] Overview of signals [EB/OL]. https://man7.org/linux/man-pages/man7/signal.7.html

[49] POSIX - The Portable Operating System Interface [EB/OL]. https://en.wikipedia.org/wiki/POSIX

[50] Signals - A limited form of inter-process communication [EB/OL]. https://en.wikipedia.org/wiki/Signal_(IPC)

[51] Hermes - A small and lightweight JavaScript engine optimized for running React Native on Android [DB/OL]. https://github.com/facebook/hermes/

[52] QuickJS - A small and embeddable Javascript engine [EB/OL]. https://bellard.org/quickjs/

[53] MuJS - A lightweight Javascript interpreter designed for embedding in other software [EB/OL]. https://mujs.com/index.html

[54] Bug rejected by the developer [DB/OL]. https://github.com/Moddable-OpenSource/moddable/issues/539

[55] Performance Bug of Hermes [DB/OL]. https://github.com/facebook/hermes/issues/134

[56] Performance Bug of XS [DB/OL]. https://github.com/Moddable-OpenSource/moddable/issues/607

[57] Functional Bug of Duktape [DB/OL] https://github.com/svaarala/duktape/issues/2392

[58] Crash of JerryScript [DB/OL]. https://github.com/jerryscript-project/jerryscript/issues/4532

[59] Test case that crashed JerryScript [DB/OL]. https://github.com/jerryscript-project/jerryscript/files/5864797/testcase.txt

攻读硕士学位期间取得的科研成果

1. 申请（授权）专利

[1] 房鼎益, 曹帅, 叶贵鑫, 田洋, 姚厚友等. 一种基于类型推断的具有引导性的测试用例变异方法: 中国, 202010200651.0[P]. 2020-03-20.

[2] 叶贵鑫, 瞿兴, 姚厚友等. 一种基于代码精简与误报过滤的编译器模糊测试方法: 中国, 202110510418.7 [P]. 2021-05-11. (受理)

2. 参与科研项目及科研获奖

[1] 国家自然科学基金，《基于大型开源仓库的软件源代码漏洞深度检测与修复方法研究》，编号 61972314

[2] 陕西省重点研发计划，《弱感知信号条件下多目标行为跨场景深度识别与认证方法研究》，编号 2020KWZ-013

致谢

首先感谢汤战勇，叶贵鑫，房鼎益，牛进平四位老师的悉心指导，让我得以顺利完成学业。汤老师不仅传授了勤奋踏实的科研态度，还帮助我提高了语言表达能力。叶贵鑫学长兼老师在科研方面帮助我提出更严谨和周密的解决方案，其积极上进的精神不断地激励着我。房老师认真严谨的工作态度是我的方向标，无论是本科教学时的认真负责，还是科研时的严肃谨慎都指引着我研究生期间的学习和科研。牛老师和蔼可亲，每次寻求帮助时她总是不厌其烦地帮助我解决问题，乐于助人的良好品质是我学习的榜样。

感谢 NISL 所有同学营造的严于律己，互帮互助，积极上进的学习氛围。感谢曹帅，田洋，瞿兴，王媛，弋雯，李豪斌在研究生期间的工作中给予的帮助与支持。感谢李豪斌在实验结果分析部分给予的帮助。感谢研三的所有同学在生活上给予的帮助，积极活泼的生活态度促使我积极工作，健康生活。感谢舍友金博，孟一和赵珂研究生三年的理解与包容，论文写作过程中的积极分享，他们的理解与包容让我得以安心完成每一项科研任务。

感谢父母在我求学路上的默默支持与无私奉献，遇到困难的时候总是他们在不断地鼓励我。无论是高中毕业后远赴他乡求学，还是攻读硕士，他们都坚定的支持着我的决定。感谢家人在生活上给予的鼓励和支持，他们总是在我遇到困难的时候开导我，在我难以抉择的时候引导我。感谢我的女朋友王蓓蕾，她在我论文写作没有思路时给予诸多建议，帮助我在困境中砥砺前行。

感谢所有教育和帮助过我的每一位良师益友，他们的帮助和指引使我不断成长和进步，祝愿他们一生平安。

感谢国家自然科学基金项目（61972314）和陕西省重点研发计划（2020KWZ-013）对本文的资助。