成绩	良好

西北大学

本科毕业论文（设计）

题目：嵌入式 JS 引擎模糊测试方法研究

学生姓名 雷璟锟

学　　号 2021117283

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 2021 级

教务处制

二〇二五年六月

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：　2025 年 6 月 4 日

摘　要

随着物联网技术的快速发展，JavaScript 凭借其灵活易用、交互性强和跨平台优势，在嵌入式系统开发中占据了重要地位。为了满足嵌入式设备的开发需求，面向嵌入式平台的 JavaScript 引擎不断涌现。为了提升嵌入式系统的稳定性与性能，确保所采用 JavaScript 引擎的可靠性显得尤为关键。然而嵌入式设备低内存、弱算力等资源受限的特性与 JavaScript 动态语言特性的结合，导致传统测试方法难以有效挖掘引擎的潜在缺陷。同时，由于其广泛应用于物联网终端、工业控制系统等关键领域，若引擎中潜在的缺陷未被及时检测修复，不仅会造成资源浪费，甚至引发嵌入式系统和应用的安全隐患。

为了检测嵌入式 JavaScript 引擎中的缺陷，本文提出了一种基于参数类型推断的测试用例变异方法，并与差分测试结合实现了模糊测试框架。具体研究内容如下：

（1）对编译器进行模糊测试需要大量符合语法语义的测试用例，为满足这一需求，本文选取 GitHub 等主流开源代码平台中维护质量较高的 JavaScript 项目，并从中提取函数构建初始测试样本。

（2）本文设计了一种基于参数类型推断的测试用例变异方法，基本过程为从函数定义中识别并抽取其参数列表，根据函数体内该参数的行为模式，分析该参数可能的数据类型，并基于各类型设计丰富的变异策略与自调用表达式，最终生成了大量语法语义正确、高覆盖率的测试用例。

（3）本文设计并实现了 EJSFuzz 测试系统，并对多个主流的 JavaScript 嵌入式引擎进行了模糊测试，对发现的缺陷进行案例分析。通过与 Fuzzilli、AFL 等模糊测试工具进行对比实验，验证了 EJSFuzz 在代码覆盖率、漏洞发现率等方面的优势。

关键词：嵌入式 JavaScript 引擎　模糊测试　用例变异

Abstract

With the rapid development of IoT technology, JavaScript has gained significant prominence in embedded systems development due to its flexibility, ease of use, strong interactivity, and cross-platform advantages. To meet this demand, various embedded JavaScript engines have emerged. However, the combination of embedded devices' resource-constrained characteristics — such as low memory and weak computational power—with JavaScript's dynamic language features makes it difficult for traditional testing methods to effectively uncover potential defects in these engines. Moreover, given their widespread use in critical domains like IoT terminals and industrial control systems, undetected flaws in these engines may not only lead to resource waste but also pose security risks.

To address the challenges of testing embedded JavaScript engines, this paper proposes a fuzzing framework based on type inference combined with differential testing. The key contributions are as follows:

(1) Fuzzing compilers requires a large number of syntactically and semantically valid test cases. To obtain such inputs, this study crawls high-quality JavaScript projects from open-source code hosting platforms (e.g., GitHub) and extracts function bodies as preliminary test cases.

(2) This paper presents a type inference-based test case mutation method. First, it extracts function parameters and infers their data types by analyzing behavioral patterns within the function body. Then, it designs diverse mutation strategies and self-invocation expressions based on these types, ultimately generating a large set of high-coverage test cases that maintain syntactic and semantic correctness.

(3) Building on these methods, we design and implement a prototype system, EJSFuzz, and conduct fuzzing tests on mainstream embedded JavaScript engines. Case studies are performed on the discovered defects. Comparative experiments with state-of-the-art fuzzing tools (e.g., Fuzzilli, AFL) demonstrate EJSFuzz's superiority in terms of code coverage and defect detection rate.

Keywords： embedded JavaScript engines; fuzzing; test case mutation

目录

1 绪论

1.1 研究背景和意义

1.1.1 选题背景

JavaScript 是一门跨平台、动态类型、面向对象的脚本语言，由 Brendan Eich 于 1995 年在 Netscape 浏览器中首次推出。JavaScript 的出现改变了 HTML 作为标记语言的局限性，为其注入了动态行为与用户交互能力，其语言简单、易于学习的特点让 JavaScript 不仅在 Web 领域大放异彩，在嵌入式开发领域也同样展现出独特优势。各类嵌入式 JavaScript 引擎如雨后春笋般涌现，如 QuickJs、JerryScript，Hermes 等，这些引擎被广泛的使用在资源受限的设备上。嵌入式 JavaScript 引擎的安全与性能问题面临着巨大的挑战，如何对 JavaScript 引擎高效测试的问题亟待解决。

作为软件生态的基础设施，编译器的正确性与可靠性不言而喻。编译器负责将人类可读的编程语句转换为机器码，这一转换过程严格遵循预定义的语言标准。语言规范明确定义了合法语法结构、语义约束条件以及对应的机器操作指令。各类编程语言均具备其独特的语法标准，并且这些标准会随着语言特性的演进而不断发展。语言规范的复杂性使得程序员在阅读和理解时面临很大的挑战，同时也给编译器测试带来了巨大的困难。如果编译器有问题，可能导致语义变化、性能退化等严重问题。对于大多数的开发人员来说，这类问题很难被定位和发现。因此，早期对编译器的测试十分重要。目前编译器的测试方法有很多，包括兼容性测试、自举测试等。

在现代软件安全领域，模糊测试由于其高覆盖率和自动化能力，已成为一种被广泛采用的漏洞检测技术。其核心机制是通过自动化生成或特殊变异策略构造海量合法/非法的输入样本，将这些样本动态注入目标程序，触发程序出现诸如崩溃、内存资源泄露或逻辑紊乱等异常现象，从而暴露潜在漏洞。该技术可被划分为基于生成模型的方法以及通过修改现有输入实现的变异型方法。模糊测试适用于多种测试模式，不论是具备源代码访问权限的白盒测试、仅有部分信息的灰盒测试，还是完全未知结构的黑盒测试。自 1988 年威斯康星大学的 Barton Miller 教授提出模糊测试以来，模糊测试已被广泛应用到各个研究领域，包括编译器，网络协议，Web 漏洞挖掘等。此外，它也常与其他测试方法相融合，例如结合差分测试用于揭示不同编译器版本间的行为差异，或借助动态污点分析与符号执行提升模糊测试在路径探索方面的能力。

1.1.2 选题意义

从软件测试的角度来看，编译器也是一种特殊的软件，其输入就是对应语言编写的程序。对于 JavaScript 引擎来说，测试用例即为符合语法和语义规范的 JavaScript 代码。然而传统的模糊测试工具由于生成策略单一（如 AFL 的位翻转），仅能对单个字节进行有效变异，很难兼顾语法正确性与语义丰富性，导致测试效率低下。

ECMAScript-262 是 ECMA（欧洲计算机制造商协会）提出的一套 JavaScript 语言规范。Test-262 则是由 ECMA 技术委员会维护的官方测试套件，旨在验证 JavaScript 引擎对 ECMAScript 语言规范的符合性。截止 2025 年 3 月，ECMA-262 已经推出了第十五版规范——ES15；Test-262 由 220 位贡献者提供了 16000+测试用例。尽管如此，由于 ECMA-262 规范的复杂性、JavaScript 语言的丰富特性，人工编写的 Test-262 还是无法完全涵盖 JavaScript 语言规范。

对于绝大部分 JavaScript 开发人员来说，核心关心点通常集中于确保自身编写的代码没有逻辑问题与安全风险，在这个过程中，开发者往往将 JavaScript 引擎视作可靠的黑匣子，默认它准确无误的执行和解析符合规范的代码，这种信任惯性往往导致开发者忽略引擎潜在的设计缺陷，最终产生安全问题。为了尽早的发现 JavaScript 引擎中存在的问题，设计一个高效的测试方法尤为重要。本文提出一种基于参数类型的测试用例变异方法，通过此方法可以基于原始语料库生成大量语法规范与语义丰富的测试用例，相比于传统的测试方法可以更广泛的覆盖 JavaScript 引擎中的各项功能模块与语言特性。此外，结合差分测试，通过分析不同 JavaScript 引擎的输出结果去探究其中潜在的漏洞。

本文基于此方法实现了 EJSFuzz 系统原型，在保证测试效率的情况下，尽可能去挖掘 JavaScript 引擎的潜在漏洞。因此，本研究对于嵌入式系统安全与 JavaScript 软件生态的发展有积极的现实意义。

1.2 国内外研究现状

1.2.1 通用模糊测试的相关研究

在自动化测试领域，模糊测试凭借其高效的缺陷检测能力已被信息安全领域的研究人员广泛应用，逐渐演变为漏洞挖掘过程中的关键分析手段。随着 AFL++、LibFuzzer 等工具的相继出现，这些工具以其良好的用户交互设计和简便的操作流程，极大地推动了模糊测试技术的广泛应用。下面介绍一些最新的模糊测试技术相关的安全研究。

Christian Holler 等人提出了一种基于语法和代码片段重组的模糊测试框架 LangFuzz[1]，

利用上下文无关的语法随机生成有效程序，确保输入通过语法检查。从已知缺陷的测试套件中学习代码片段，通过替换和重组生成新测试用例，提高触发异常的概率。LangFuzz 在 Mozilla JavaScript 引擎中发现 105 个高危漏洞。

Suyoung Lee 等人将神经网络语言模型（NNLM）Montage[2]用于 JavaScript 引擎模糊测试。Montage 创新地将 JS 代码的抽象语法树（AST）分解为深度为 1 的子树片段序列，并基于 LSTM（长短期记忆网络，Long Short-Term Memory）模型学习这些片段间的组合关系，生成语法和语义合理的测试用例。

Chenyuan Yang 等人提出了一种基于大语言模型（LLM）的白盒编译器模糊测试框架 WhiteFox[3]，通过解析编译器优化的源代码，生成混合自然语言和伪代码的需求描述。根据需求自动合成符合触发条件的测试程序。通过反馈循环机制，将成功触发优化的测试作为示例动态优化提示词，并利用 Thompson 采样算法提升测试效率。该工作被 PyTorch 团队认可并推动了白盒模糊测试在复杂编译系统中的实用化。

Patrice Godefroid 等人结合随机测试与符号执行技术提出了 DART[4]，通过动态分析程序执行路径并自动生成新测试用例，DART 在程序运行时收集路径约束条件（如分支判断），随后对约束进行逻辑反演并调用求解器生成满足新路径的输入数据。

Xuejun Yang 等人开发了随机测试工具 Csmith[5]。通过系统化生成随机 C 程序检测编译器缺陷。该方法创新性地构建语法约束模型，自动生成符合 C99 标准、严格规避 191 种未定义行为的合法测试代码，确保程序语义确定性。利用差分测试技术，将同一程序在不同编译器（如 GCC、LLVM）及优化等级下的输出结果交叉验证，定位异常行为。

1.2.2 JavaScript 引擎的模糊测试方法

Haoran Xu 等人提出了一种基于图中间表示的 JavaScript 引擎模糊测试方法。FlowIR[8]通过显式建模程序的控制流和数据流，支持细粒度语义变异，解决了传统 AST 和字节码 IR 在生成有效及语义有意义测试用例上的不足。作者设计了 FlowIR 的双向转换机制（JS 代码↔图结构），并开发了原型工具 FuzzFlow，实现了数据流子图变异（修改节点属性/输入）和控制流子图变异（节点调度/插入/删除）两类操作符，提升变异效率与语义有效性。研究成果验证了图结构表示在挖掘 JS 引擎深层漏洞中的显著优势。

Spandan Veggalam 等人提出了一种基于遗传编程的模糊测试工具 IFuzzer[13]。通过语法规则生成有效代码片段，运用交叉、变异等遗传操作进化测试用例，并结合代码复杂度与解释器反馈构建适应度函数，引导生成能触发异常行为的非常规代码。IFuzzer 在 Mozilla 旧版

SpiderMonkey 中发现 40 个漏洞。

Sung Ta Dinh 等人提出了一种针对 JavaScript 引擎绑定层代码的模糊测试方法 Favocado[7]，通过解析 IDL 文件或 API 参考手册提取绑定对象的语义信息（如方法参数类型、属性约束），确保生成的 JavaScript 测试用例语法和语义正确，成果凸显了语义感知与输入空间优化在绑定层测试中的有效性。

韩国科学技术院（KAIST）的 HyungSeok Han 团队提出了 CodeAlchemist[14]，一种基于语义感知的 JavaScript 引擎模糊测试工具。CodeAlchemist 创新性地将种子代码分解为“代码块”，通过静态数据流分析和动态类型推断，为每个代码块添加组装约束，定义变量类型及依赖关系，确保组合后的代码在语法和语义上均有效。该方法无需手动编写语法规则，自动学习 JS 语义，凸显了语义感知方法在漏洞挖掘中的优势。

1.3 本文研究内容

本文提出了一种基于参数类型推断的测试用例变异方法，通过分析原始测试用例得到代码段的抽象语法树，提取语义信息，确定可进行变异的参数，根据数据类型执行变异，提升覆盖率与测试效率。具体研究内容如下：

（1）JavaScript 引擎语法解析策略。研究引擎在处理 JavaScript 代码时如何生成并优化抽象语法树，确保代码的语法和语义信息正确，从而执行高效准确的变异。

（2）测试用例变异策略，通过对 JavaScript 代码段的静态语法树分析，推断其可变异参数和数据类型，基于数据类型对该参数执行不同的变异策略，从而发现引擎更深层次的缺陷与漏洞。

（3）设计并实现 EJSFuzz 系统，通过对上述方法的研究，本文实现了 EJSFuzz 系统。对该系统的模块结构，算法流程做详细介绍，将其应用到各大主流嵌入式 JavaScript 引擎做模糊测试，评估该方案的可行性与实用性。

1.4 本文组织结构

第一章 绪论。本章首先概述了 JavaScript 语言及其应用背景，编译器测试领域的相关研究，对当前 JavaScript 引擎测试的现状进行了简要分析。最后阐明本文的研究的问题和研究目标。

第二章 理论与实验论证。本章主要阐述了本文测试对象 JavaScript 引擎的基本架构，系统中使用的模糊测试技术、测试用例变异技术。

第三章 基于类型推断的嵌入式 JS 引擎模糊测试方法。本章重点阐述了本文所提出的差

分模糊测试方法的整体框架，详细描述了参数类型推断模块和变异算法的具体实现。

第四章 系统设计与实验评估。 本章主要介绍基于本文实现的嵌入式 JavaScript 引擎差分模糊测试系统 EJSFuzz，介绍实验的设置与环境，然后对测试结果进行深入分析，并进一步评估实验的有效性。最后，探讨缺陷产生的根本原因并提出修复方案，同时与其他模糊测试工具进行对比实验。

第五章 结论与展望。 本章通过对实验数据的深入分析，揭示了差分模糊测试系统在推动嵌入式 JavaScript 引擎测试工作中的重要作用。此外，本文还指出了该系统在应用过程中的一些局限性，对未来的测试工作研究展望。

2 理论与实验论证

2.1 嵌入式 JavaScript 引擎

2.1.1 嵌入式 JavaScript 引擎简介

在互联网与物联网（IOT）蓬勃发展的今天，物联网开发日益火热，但这同时也暴露了许多问题。传统的物联网开发对开发者提出了较高的要求，开发者不仅需要具备扎实的 C/C++ 语言基础，如指针操作、内存管理等底层机制，又要熟悉外设驱动的系统调用与寄存器状态读取等硬件交互方式，形成较高的技术准入门槛。开发流程层面，本地化编译-链接-下载的闭环工作流导致跨平台移植困难，同一功能场景在不同硬件架构中需重复构建固件，给物联网终端设备分散化、硬件碎片化与场景多样化的带来了巨大的挑战。JavaScript 作为时下流行的 Web 开发语言，开发者社区活跃、解释型运行、移植性强等特性使其在嵌入式开发也有一席之地。另外 JavaScript 拥有大量现成的开源框架和工具，能大大简化开发流程。JavaScript 语言本身高效又容易上手，其运行速度接近 C 语言，但写起来却简单得多，开发者通过封装好的方法就能控制设备。这样一来，既保证了性能，又能专注在业务功能开发上。

嵌入式设备是针对特定功能定制化设计的小型计算系统，相较于通用计算机，其核心特征体现在硬件资源的高度精简——成本低廉、计算性能有限、内存及存储空间较小，同时对低功耗运行有严格要求。要让 JavaScript 在这类设备上运行，必须为其开发专门的 JavaScript 解释器或编译器即 JavaScript 引擎。由于嵌入式设备的存储空间极为有限，JS 引擎自身必须保持极小的体积，才能确保在资源受限的环境中稳定工作。因此传统的桌面端 JavaScript 引擎如 Chrome V8、SpiderMonkey 等并不适用于嵌入式环境。各类嵌入式 JavaScript 引擎应运而生，TinyEngine 是由阿里云设计的一款高性能 JavaScript 引擎，其专为嵌入式系统设计、资源占用少，可以做到在内存 10KB 大小的系统上运行。JerryScript 是三星公司开源的轻量级 JavaScript 引擎，目前已应用于三星物联网生态与华为鸿蒙生态。

2.1.2 嵌入式 JavaScript 引擎运行原理

JavaScript 引擎负责解释和执行 JavaScript 代码，了解其内部架构对测试工作有重要意义。QuickJS 是 Fabrice Bellard 继 FFmpeg 和 QEMU 的又一力作，于 2019 年开源在 Github 平台。QuickJS 只有 210KB，体积小，启动快，解释执行速度快，支持最新 ECMAScript 标准。本节以 QuickJs 为例，介绍嵌入式 JavaScript 引擎的架构与工作流程。

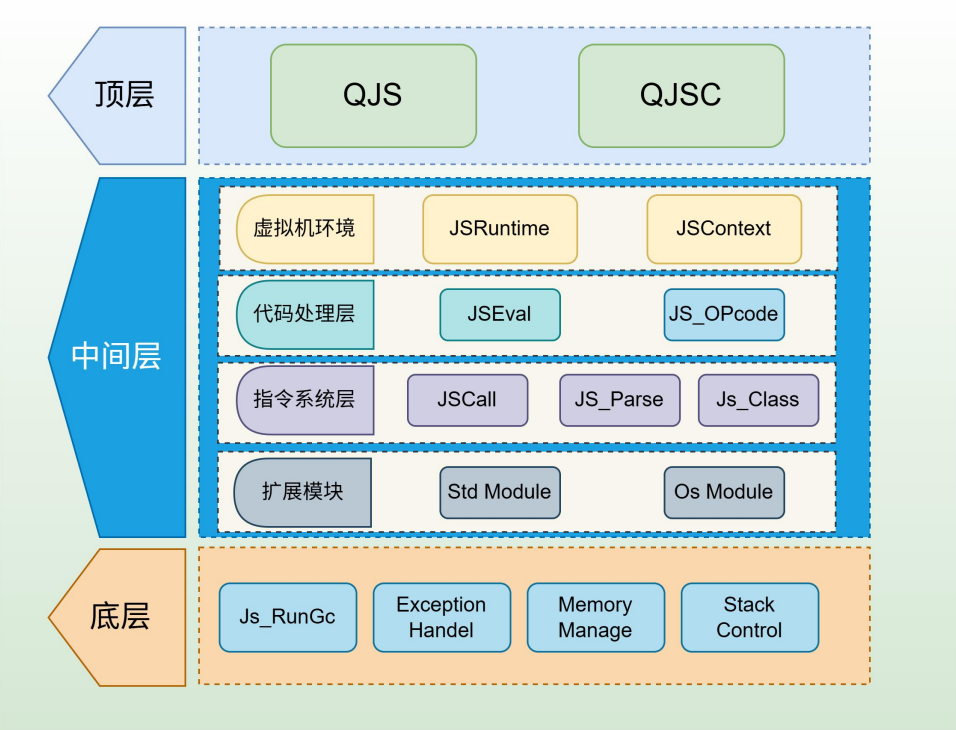

图 2-1 QuickJs 引擎架构

如图 2-1 所示，架构最顶层的是 QJS 与 QJSC，其中 QJS 负责解析命令行参数、引擎环境的初始化、依赖模块的加载以及对 JavaScript 文件的读取、解释、执行；QJSC 则负责对 JavaScript 源代码进行编译，输出可执行的字节码。

QuickJS 的整体架构以中间层为核心，其运行时环境由多层级组件构成。在基础层，JSRuntime 作为 JavaScript 虚拟机环境，提供相互隔离的独立运行空间，不同 JSRuntime 实例之间无法进行跨环境通信或调用。JSRuntime 创建的 JSContext 虽然共享相同的底层运行时资源，但各自拥有独立的全局对象和系统对象。在代码处理层，源代码到字节码的转换通过 JS_Eval 和 JS_Parse 实现，生成的字节码由 JS_Call 负责解释执行。在指令系统层采用 JS_OPCode 定义操作符标识，根据 quickjs-opcode.h 的定义，QuickJS 采用紧凑型存储策略，8 位以下指令与附加信息共享单字节空间，8 位和 16 位指令分别占用 2 字节和 3 字节空间，整数参数直接嵌入后续字节。对象系统方面，JSClass 构建了标准 JavaScript 对象模型，通过 JSClassID 实现类型标识。开发者使用 JS_NewClassID 注册新类型，JS_NewClass 创建类定义，最终通过 JS_NewObjectClass 实例化对象。Unicode 支持由独立模块 libunicode.c 实现，libunicode.c 不仅完整覆盖 Unicode 规范化处理、脚本类别查询和二进制属性管理，还可作为独立库集成到其他项目。核心功能扩展层包含多个专用模块：libbf 库提供高精度数值计算的 BigInt/BigFloat 支持，libregexp 实现正则表达式引擎。系统能力通过扩展模块增强，Std Module

封装标准功能集，OS Module 则暴露文件操作、时间处理等系统级接口，共同构成完整的运行时能力体系。

最底层是基础，JS_RunGC 用于垃圾回收来防止野指针与内存泄露的出现，它通过引用计数法来判断对象是否可以被释放。JS_Exception 通过 JS_ThrowSyntaxError 创建异常并抛出，返回的异常对象 Error 存储在 Context 中。Memory Control 负责管理 JS 运行时的全局内存。Stack Control 负责管理 JS 运行时的堆栈数据结构。

QuickJs 引擎的工作流程如下：

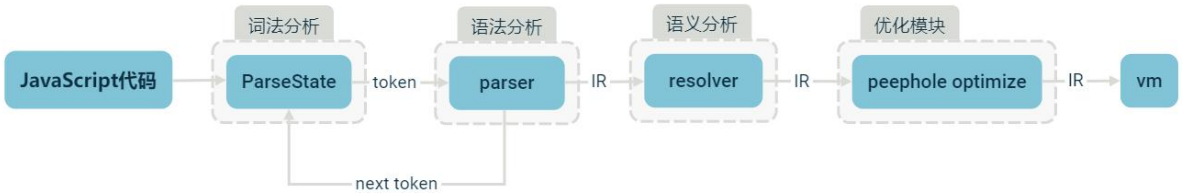

图 2-2 QuickJs 引擎工作流程

（1）代码字节流解析。首先，源代码经引擎读取后以字节流形式注入词法分析器。在词法处理阶段，分析器通过扫描机制逐字符检测 JavaScript 代码，依据预置分词规则执行代码解构：既剥离无意义的空白符与注释内容，又将有效元素封装成具有语义的单词符号（token），最终输出结构化的 token 流传输至语法解析器进行深度处理。

（2）语法分析。在这个阶段，语法解析器（Parser）接收词法层传递的 token 流输入，通过递归式替换操作逐步构造树形结构：首先将每个 token 映射为语法树节点，随后持续用终结符号替换产生式中的非终结符号，直至完成所有符号的实例化转换。最终，这些节点依据语法规则形成层级分明的树状数据结构，即抽象语法树（AST）。

（3）语义分析。Reslover 模块的核心功能是执行语义分析，确保程序符合语言规范。其具体工作涵盖变量引用有效性验证、类型一致性推导、循环结构合规性检测（如 break 语句的合法使用），以及作用域规则审查。在完成上述静态检查后，该模块将 IR 输出至后续优化阶段。

（4）分析优化。优化模块首先选择需要优化的中间代码段，通常是相邻的几条指令，接着识别这些指令中的特定模式，如无用的加载、相同操作的重复或可合并的算术运算。识别后，编译器会应用优化策略，将冗余指令替换为更高效的指令或简化代码，例如将连续的加法合并为一次加法。随后，优化后的代码会被更新，确保逻辑与原始代码保持一致。最后，编译器会对更新后的代码再次进行分析，重复上述过程，直到无法进一步优化为止。

（5）解释执行。最终产生的 IR 会传递给 VM 执行，首先 VM 会加载字节码并初始化执行环境，随后进入主执行循环逐条读取和执行字节码指令。在这一过程中，VM 解码指令并执行相应操作，同时管理栈帧以处理局部变量和返回值。此外，它还会监测异常并进行处理，并在适当时机触发垃圾回收以管理内存。执行完所有指令后，虚拟机会清理环境并返回结果。

2.2 模糊测试理论

模糊测试是一种广泛用于评估软件健壮性与安全性的自动化技术，其基本思想是在无需详细了解系统内部结构的前提下，持续向目标程序注入大量非预期或随机生成的输入数据。通过观察系统在面对异常数据时的行为表现，以识别可能存在的漏洞或稳定性问题。为实现输入的自动生成，测试系统通常配备特定的用例生成模型，能够持续构造具有多样性的测试输入并将其输入至被测软件。在程序运行过程中，监控模块实时捕捉崩溃、错误或其他异常信息，并记录触发条件与相关上下文信息。与传统的手工漏洞检测方式相比，模糊测试技术具有高度自动化、无需专业背景知识、适配范围广等优势，因而在编译器测试领域中被广泛采用。

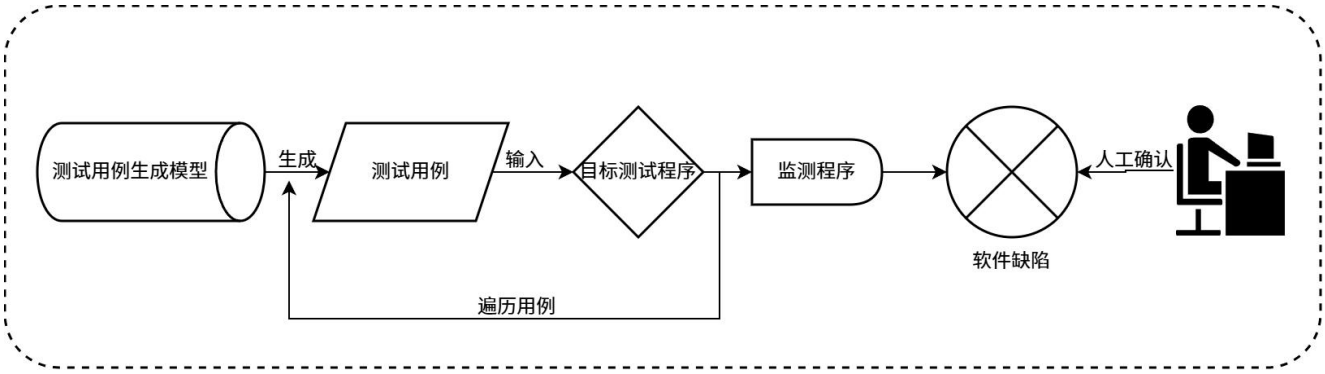

图 2-3 模糊测试过程概述

如图 2-3 所示，模糊测试的基本流程分为四个阶段：测试用例生成、执行测试用例、异常监视以及漏洞确认。

（1） 测试用例生成。为全面评估被测软件的稳定性与安全性，必须持续生成多样化的测试用例，而用例生成的策略对测试质量具有决定性作用。目前常见的方法大致可分为两类：生成型策略与变异型策略。生成型方法通常基于预设的语法规则或结构模板，通过一定的随机机制构造出新的输入数据，代表性工具如 jsfunfuzz。相较而言，变异型技术依赖于已有的测试样本，即“种子用例”，通过对其进行改写、重组或注入代码片段以形成新的测试样本。例如，LangFuzz 通过拼接方式整合多个代码片段生成测试输入。生成式方法需要深入的领域知识以编写适合的规则，而突变式方法则更依赖随机性，可能难以触发特定漏洞。

（2） 执行测试用例。在生成测试用例后，Fuzzer 会将其输入至目标软件并执行，以观察软件对这些输入的反应。为了提高测试效率，该过程通常会自动化运行，并在有限时间内完成对大规模测试样本的快速执行，从而快速发现可能存在的异常情况。

（3） 异常监视。在测试用例执行的整个过程中，Fuzzer 会持续追踪被测程序的运行情况，并识别可能出现的非预期现象，如系统崩溃或逻辑功能失效等。一旦检测到这类异常，相关测试样本将被标记并保存。异常检测手段主要包括两类：一类是以进程级别为基础的检测方式，另一类则依赖于插桩技术。在进程监测方案中，模糊器以父进程的角色运行目标程序，并依据其返回状态码及异常信号推断运行中是否存在错误。而插桩检测则通过在源代码或可执行文件中注入辅助代码片段，从而在程序运行期间采集动态信息，从而对程序执行路径与状态进行进一步评估。

（4） 漏洞确认。在完成测试和异常监视后，研究人员需要对收集到的异常数据进行人工分析，以确认是否存在真正的软件漏洞。这一过程不仅包括验证测试用例是否触发了缺陷，还需要分析漏洞产生的根本原因。通过结合自动化检测和人工分析，模糊测试能够有效提升软件漏洞的发现能力，并为安全性改进提供依据。

2.3 测试用例变异技术

模糊测试需要大量测试用例作为输入。通过对原始测试用例实施特定变异策略，既能批量生成测试用例，又能有效提升代码覆盖率。测试用例变异技术的效果主要取决于两个关键因素：原始语料库的质量和变异策略的设计。原始语料库获取相对容易，可从开源项目、历史测试用例或实际运行日志中收集，这些都为测试提供了丰富的语料。因此设计高效的变异策略是提升模糊测试效果的关键。

AFL（American Fuzzy Lop）是一款基于遗传算法的开源模糊测试工具。目前，该工具已在多个主流软件项目中发现了数十个重大漏洞，涉及项目包括 PHP、OpenSSL、pngcrush、Bash、Firefox、BIND、Qt 和 SQLite 等。在对目标程序进行插桩后，AFL 会对测试样例进行变异操作。主要的代码逻辑位于 afl-fuzz.c。AFL 变异的主要类型有下面这几种：

确定性变异：

(1) bitflip（位翻转）：将 1 变为 0，0 变为 1；即在二进制层面对数据位进行取反处理。

(2) arithmetic（算术运算）：整数加/减算术运算

(3) Interest（特殊值替换）：将 AFL 内置的边界测试值替换到原始测试用例中。

随机性变异：

(4) Havoc（混合变异）：综合运用前述多种变异策略进行随机扰动。

(5) Splice（文件拼接）：合并两个输入文件生成新样本，并对其执行 Havoc 变异。

AFL 的变异策略主要针对线性二进制数据，而 JS 代码具有严格的语法和语义结构，随机变异极易生成无效脚本，降低测试效率。本文通过探索 JavaScript 语法树解析策略与参数类型推断方法，提升测试用例变异的有效性，从而最大限度的测试嵌入式 JavaScript 引擎。

3 基于类型推断的嵌入式 JS 引擎模糊测试方法

3.1 嵌入式 JavaScript 引擎模糊测试方法设计思路

在自动化的模糊测试中，验证测试方法有效性主要依赖两个关键因素：一方面是测试用例的生成质量，另一方面是缺陷检测方法的准确性。高质量的测试用例是发现潜在缺陷的基础，而高效的缺陷检测手段则决定了缺陷是否能够被及时且准确地识别。本文设计了一种基于类型推断的测试用例变异方法。通过该方法生成的测试用例语义信息丰富，能显著提升代码覆盖率和整体测试效率。

3.1.1 方法概述

基于类型推断的 JS 引擎模糊测试方法的简要工作流程如图 3-1 所示：

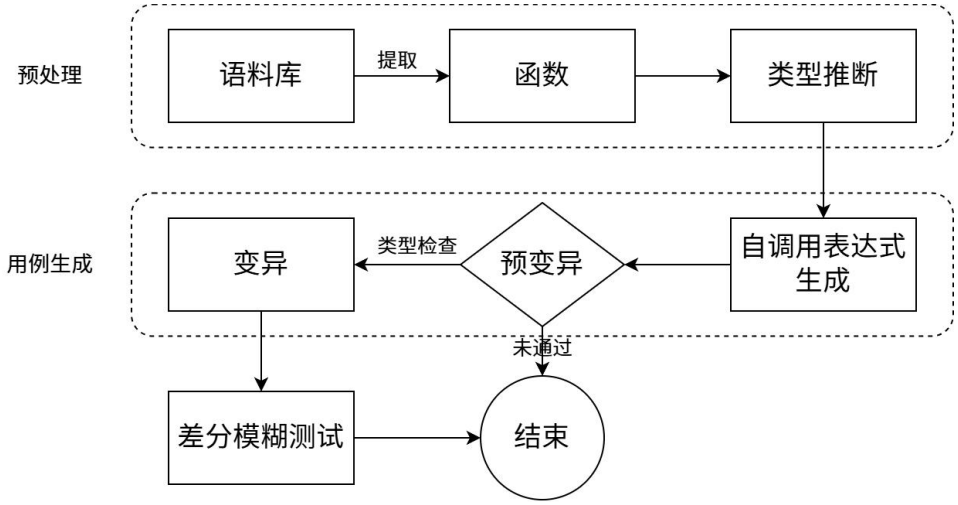

图 3-1 嵌入式 JavaScript 模糊测试方法

在预处理阶段，首先会初始化数据，加载配置文件，读取语料库里的所有原始语料，接着需要从原始语料里提取上下文完整，语义丰富的函数体，这部分通过正则表达式实现，代码 3-5 展示了提取函数名的过程：

代码 3-1 提取函数名算法

```
def extract_function_name(self, function_body: str):

    index_of_function = function_body.find('function', 0)
    index_of_open_parenthesis = function_body.find('(', index_of_function)
    function_name= function_body[index_of_function + 8:index_of_open_parenthesis]
    return function_name.strip()
```

首先在代码字符串中查找“function”关键字，用于找到函数定义的起始位置，接着在

“function”关键字之后查找第一个“（”括号来找到函数名结束的位置，截取“function”关键字和“（”括号之间的字符串来作为函数名，最后通过 strip 函数来去除空格。

提取函数体的算法与函数名类似：首先，它在文件内容中循环查找 “function”关键字，以此确定函数定义的起始位置。一旦找到 “function”，算法会提取从“function”关键字开始，直到与之匹配的闭合大括号“}”之间的所有字符，构成完整的函数体。为了简化函数头的格式，算法会将函数名部分进行替换，将类似于“function name(...)”的形式简化为“function(...)”。最后，将提取出的函数体添加到列表中，并重复此过程，直到文件内容被完全遍历。

通过前面的步骤提取到函数以后，我们需要显式的调用这个函数，即生成函数的调用表达式。在生成的过程中，我们需要根据函数的参数类型制定不同的生成策略，参数类型的推断方式将在下节介绍，而参数的生成策略需要我们自行设计，下面给出本文的设计策略，如图 3.1 所示：

表 3.1 参数生成策略

参数类型	生成策略	示例值
字符串	随机生成指定长度的字符串，可以包含字母、数字和特殊字符	“abcde123!”, “xyz@#456”
数字	随机生成指定范围内的浮点数或整数	1.5，10，10.0，-50
布尔值	随机返回 true 或 false	True，false
数组	随机生成指定长度的数组，内容可以是任意类型，可以包含嵌套	[1,2,3]，[“a”,”b”,”c”]
对象	随机生成指定属性的对象，属性值为随机类型，可以包含嵌套	{name:”John”,age:30,address:{ city:”NY”,zip:”1001”}
函数	随机生成一个简单的函数	()=>Math.random()

生成自调用表达式后，一个完整的测试用例就已生成，随后对这个测试用例进行预变异，预变异的实现如代码 3-2 所示，在执行用例变异前差分运行这个测试用例，如果该测试用例存在参数类型错误，则不进行后续变异。通过预变异，可以避免因上一步参数类型推断错误导致对无意义或不符合要求的测试用例进行进一步处理，提高测试效率。

代码 3-2 预变异算法

```
def premutation(self, original_test_case: str) -> bool:
    harness_result = self.config.harness.run_testcase(original_test_case)
    for output in harness_result.outputs:
        stderr = output.stderr
        if stderr is None:
            return False
        if "TypeError" not in stderr:
            return False
    return True
```

通过预变异的测试用例则继续进行变异操作，具体的变异算法在上节已经介绍。变异后的测试用例存储在数据库里，差分运行这些用例并保存运行结果，通过对比运行结果，保存可能触发崩溃的 JS 文件到本地。

3.1.2 参数类型推断方法

由于 JavaScript 语言动态类型的特性，其变量在定义阶段不要求预先指定具体的数据类型，变量的类型只有在引擎执行时才能被确定。动态类型的设计为 JavaScript 带来了独特的灵活性与开发效率，也不可避免地引入了若干限制与问题，如代码的可读性低，类型错误只能在运行时暴露等。对于含有参数的函数，如果传入错误的参数会导致触发类型错误而提前中止测试流程，导致代码覆盖率降低，因此参数类型推断是执行精准变异不可或缺的一步。

JavaScript 中共有 8 种基本的数据类型，其中值类型有 6 种：字符串、数字、布尔、空、未定义、Symbol。引用数据类型有 3 种：对象、数组、函数。在类型推断模块中，通过静态文本分析推断参数可能的数据类型，以图 3-2 为例讲解。

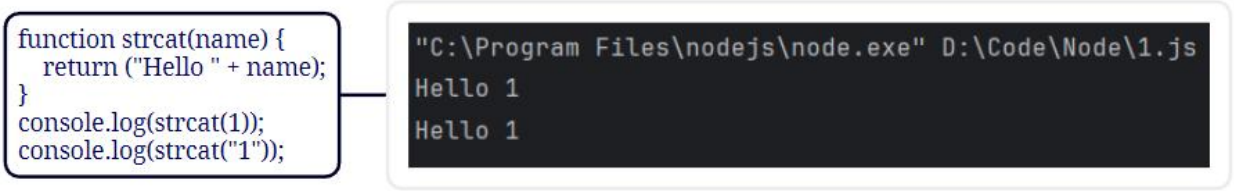

图 3-2 运行结果图

左侧的代码展示了之前示范的字符串拼接函数，我们输入的参数分别为数字 1 和字符串“1”，看到程序打印了相同的结果，表明 name 参数可能接受的变量类型为字符串和数字。在参数类型推断时，首先我们需要提取函数里的参数，代码 3-3 展示了提取的过程，首先通过左右括号索引定位参数位置，再通过分割符“，”获取各个参数名。随后将参数名与特征

因子拼接，并在测试用例里统计拼接变量的个数。以 strcat 函数为例，我们将特征因子“+”与“name”拼接，统计“+name”的个数。统计发现，该函数字符串类型与数字类型的因子均为 1，那么我们推断为字符串或数字。基于此结果，我们在后续的变异中执行字符串变异与数字类型变异。

代码 3-3 提取函数参数

```
    def extract_params(self, callable):
        left_index = callable.find('(')
        right_index = callable.find(')')
        raw = callable[left_index + 1:right_index]
        if raw.__len__() < 1:
            return []
        params = raw.strip(' ').split(',')
        for i in range(0, params.__len__()):
            params[i] = params[i].strip(' ')
        return params
```

3.2 基于类型推断的变异算法设计

本文在上节介绍了一种 JavaScript 函数参数类型推断的方法，通过类型推断的结果我们可以对参数执行更加准确的变异，针对不同的类型设计丰富的变异策略，从而达到提高测试覆盖率的目的。

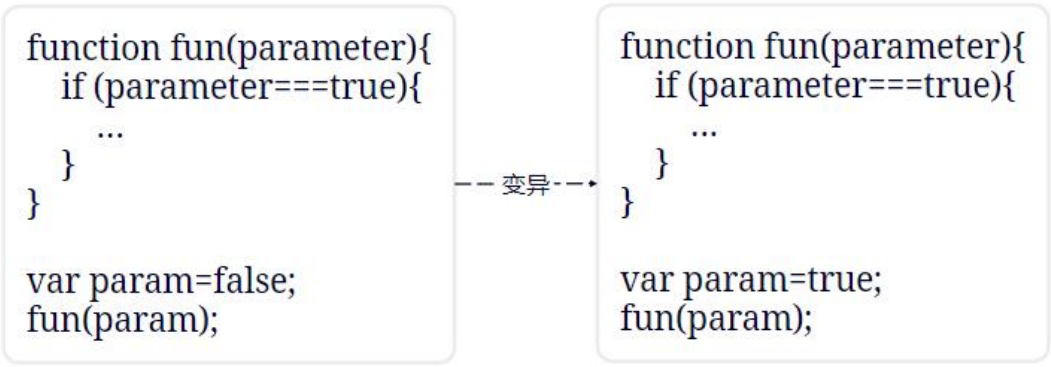

图 3-3 测试用例变异

如图 3-3 所示，我们推断函数 fun 的参数 parameter 的类型为 Boolean 类型，那么就将输入取反，原本的输入无法通过 if 判断，导致更深层次的代码无法被执行，而变异后的输入可以通过 if 判断，更深层次的逻辑可以被触发，提高了测试用例的代码覆盖率。具体的变异策略如表 3.2 所示。

表 3.2 测试用例变异策略

参数类型	变异策略	示例
Booleam	1. 翻转	true->false;false->true
String	1. 大小写转换	1. sttr.toUpperCase()、sttr.toLowerCase();
	2. 字符串截取	2. sttr.substring(0, sttr.length-1);
	3. 字符串替换	3. sttr.replace(\"a\", \"b\");
	4. 插入随机字符	4. str = str + String.fromCharCode(Math.random())
	5. 重复字符串	5. str = str.repeat(2)
	6. 替换为特殊字符	6. str = '!@#$%'
	7. 插入 Unicode 字符	7. str = str + '\u03A9'
Number	1. 值替换	1. let num = 0;
	2. 边界值变异	2. let num = Number.MAX_VALUE;
	3. 符号翻转	3. let num = -5;
	4. 浮点精度调整	4. let num = 3.14159;
	5. NaN 替换	5. let num = NaN;
	6. 随机值替换	6. let num = Math.random() * 100;
Array	1. 反转数组元素顺序	1. arr.reverse()
	2. 随机打乱数组元素	2. arr.sort(() => Math.random() - 0.5)
	3. 插入空值	3. arr.push(null)
	4. 插入未定义值	4. arr.push(undefined)
	5. 删除随机元素	5. arr.splice(Math.floor(Math.random() * arr.length), 1)
	6. 清空数组	6. arr.length = 0
	7. 嵌套数组	7. arr.push([...arr])
	8. 替换为稀疏数组	8. arr = new Array(5)
Function	1. 函数替换	1. fun = () => {}; → fun = function() {};
	2. 参数增删	2. fun(a, b) → fun(a) 或 fun(a, b, c)
	3. 返回值修改	3. fun = () => 1 → fun = () => null
	4. 函数绑定修改	4. fun.call(obj) → fun.apply(obj)

下面对测试用例变异算法做简要介绍：

算法 1 测试用例变异算法

输入：test_case_code：输入的代码段 max_size：函数变异次数，默认为 1
输出：result：变异后的测试用例列表

```
1:  FUNCTION mutate(test_case_code: STRING, max_size: INTEGER = 1) RETURNS LIST
    OF STRING
2:    SET self.max_size = max_size
3:    SET result = EMPTY LIST
4:    SET callable_proc = NEW CallableProcessor("callables")
5:    SET result_type = callable_proc.generate_self_calling(test_case_code)
6:
7:    IF result_type IS NOT NULL THEN
8:      SET params = SPLIT(result_type[0], ',')
9:      SET types = result_type[1]
10:
11:     FOR i FROM 0 TO LENGTH(params) - 1 DO
12:       SET param = params[i]
13:       SET type = types[i]
14:
15:       IF type CONTAINS 'string' THEN
16:         FOR j FROM 0 TO self.max_size - 1 DO
17:           APPEND stringmethod(test_case_code, [param]) TO result
18:         END FOR
19:       ELSE IF type CONTAINS 'integer' THEN
20:         FOR j FROM 0 TO self.max_size - 1 DO
21:           APPEND integermethod(test_case_code, [param]) TO result
22:         END FOR
23:       ELSE IF type CONTAINS 'boolean' THEN
24:         FOR j FROM 0 TO self.max_size - 1 DO
25:           APPEND booleanmethod(test_case_code, [param]) TO result
26:         END FOR
27:       END IF
28:     END FOR
29:
30:     RETURN UNIQUE(result)
31:   END IF
32:
33:   RETURN EMPTY LIST
34: END FUNCTION
```

算法的解析如下：

1 行：定义函数 mutate，输入为测试用例代码字符串 test_case_code 和最大变异次数 max_size（默认值为 1），返回变异后代码组成的字符串列表。

2-4 行：设置实例变量 self.max_size 为 max_size，初始化结果集合 result，并创建 CallableProcessor 对象 callable_proc，用于分析输入代码中的可调用结构。

5 行：通过 callable_proc 对象调用 generate_self_calling 方法，分析输入代码，获取其自调用结构类型信息，存入 result_type。

7-8 行：若 result_type 不为空，则提取其中的参数列表和对应的类型信息，分别存入变量 params 和 types。

11-13 行：进入循环，依次遍历每一个参数和其对应的类型。

15-27 行：根据参数类型执行相应的变异操作：

若类型包含 'string'，则进行字符串变异，每次调用 stringmethod 并将结果添加至 result。

若类型包含 'integer'，则进行整数变异，调用 integermethod。

若类型包含 'boolean'，则进行布尔值变异，调用 booleanmethod。

每类变异方法均执行 max_size 次，形成多个不同的变异结果。

30 行：使用 UNIQUE 操作去重结果集合，并将去重后的列表返回。

33 行：若 result_type 为 NULL，则返回空列表。

4 系统设计与实验评估

4.1 系统设计概述

EJSFuzz 系统的框架如图 4-1 所示，主要由语料库处理模块、测试用例处理模块、差分模糊测试模块、测试结果处理模块以及其他模块组成。

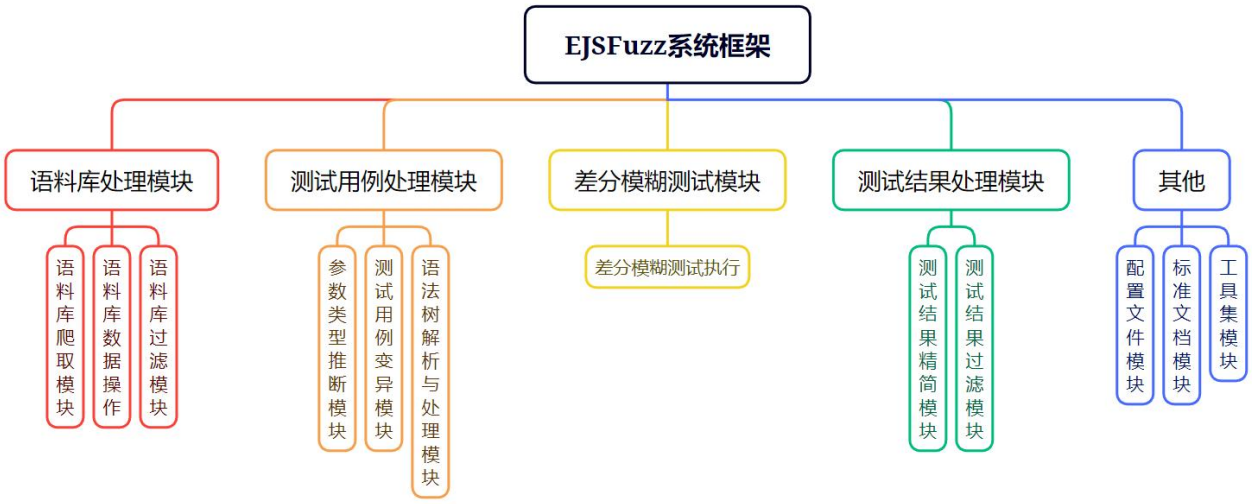

图 4-1 系统框架图

下面对这些模块做详细的介绍：

(1) 语料库处理模块

本模块主要负责语料库相关操作。其中，语料库搜集模块是一个网络爬虫。该爬虫系统主要从 Github 等主流代码托管平台自动抓取 JavaScript 源代码文件，通过筛选和去重处理后，构建初始的种子语料库。语料库数据操作模块封装了系统中所有的 SQLite 数据库操作，如测试用例、测试结果以及相关元数据的存储、查询和更新功能等，并使用预编译处理所有的增删改查操作，语料库数据操作的类图如图 4-2 所示。语料库过滤模块负责过滤无效的测试用例，如重复、语法错误的测试用例。语法错误的测试用例会因为无法正确执行而拖慢测试的执行，降低测试效率。

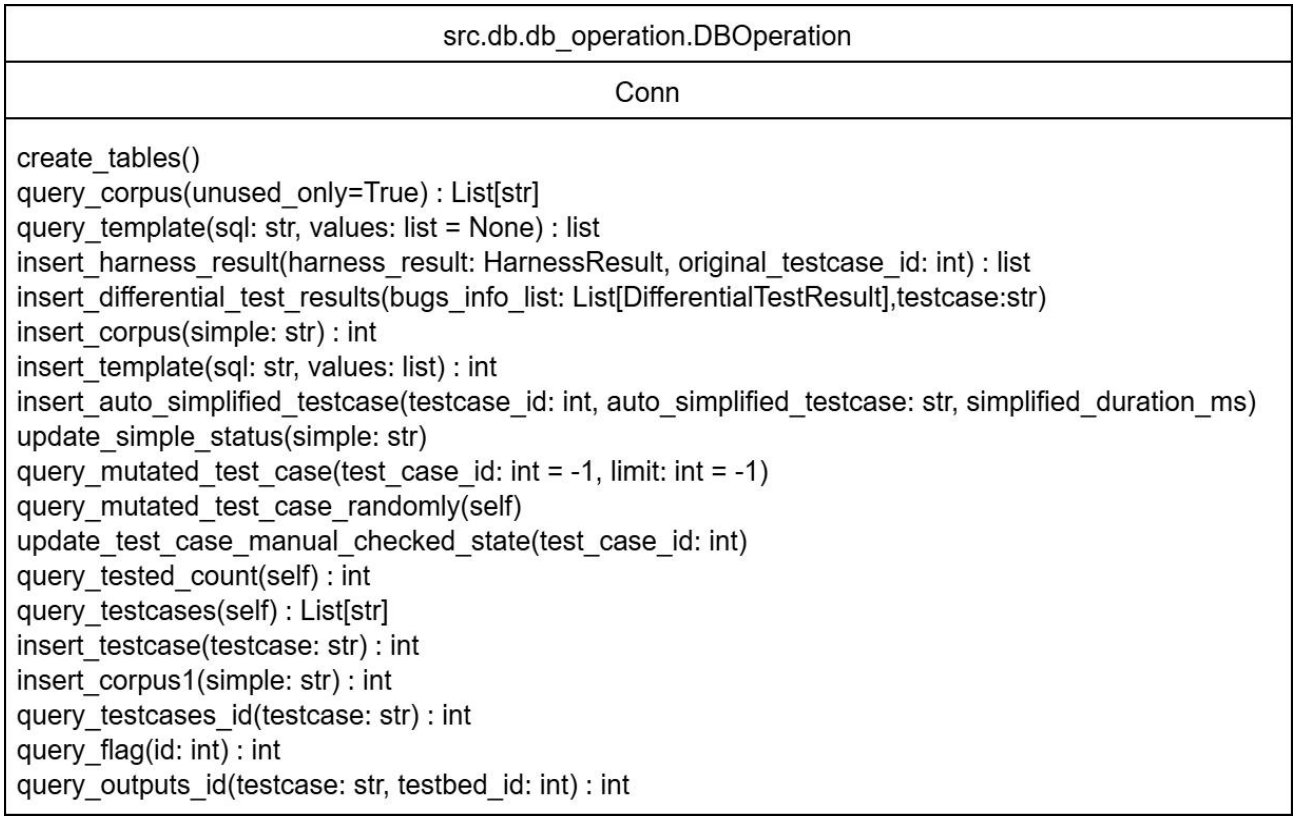

图 4-2 语料库数据操作类图

(2) 测试用例处理模块

测试用例处理模块包括参数类型推断、测试用例变异模块、语法树解析与处理模块三个子模块。其中语法树解析与处理模块由 Esprima 库辅助实现，对其做简要介绍以帮助我们理解测试用例变异模块的实现流程。

代码 4-1 esprima 解析 strcat 函数

```
const esprima = require('esprima');
//strcat 函数
const code = `function strcat(name) {
    return ("Hello " + name);
}`;
//解析语法树
const ast = esprima.parseScript(code);
//以 JSON 格式打印
console.log(JSON.stringify(ast, null, 2));
```

代码 4-1 是一段用 esprima 解析一个简单的字符串拼接函数的抽象语法树，最后以 JSON 格式输出。strcat 函数解析的抽象语法树如图 4-3 所示：

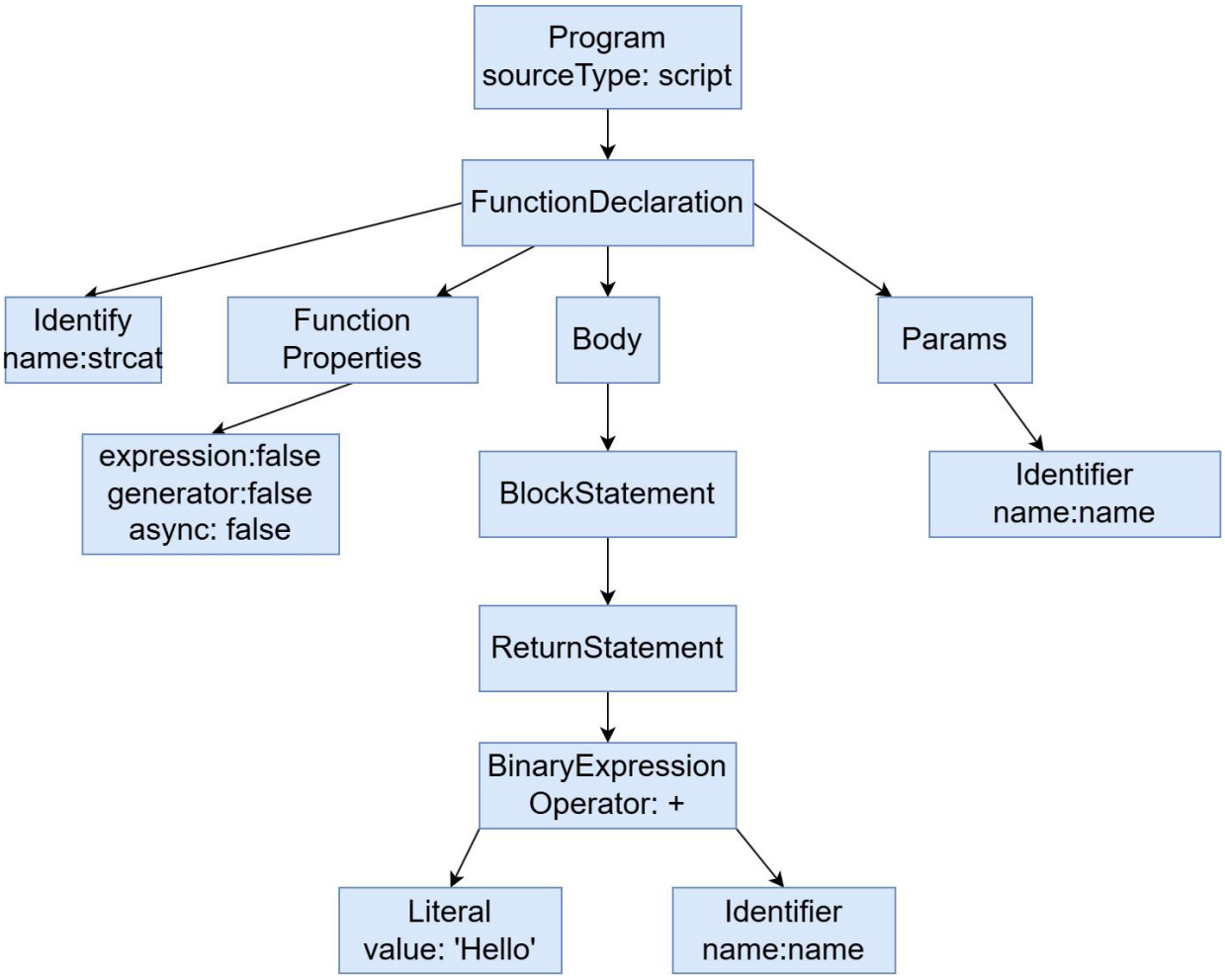

图 4-3 strcat 函数 AST 解析

通过该 AST 语法树可知，整段代码由一个 FunctionDeclaration 组成。函数名为 strcat，保存在 Identifiler 属性中，JavaScript 抽象语法树中所有的标识符都保存在这个属性中。有一个参数名为 name。函数体中有一个返回语句 ReturnStatement，返回了一个二元表达式 BinaryExpression。这个二元表达式的左操作数为字面量“Hello”，右操作数为变量 name。Function Properties 为函数属性，它表明了该函数非生成器、非表达式、非异步。

escodegen 是一款 JavaScript 代码生成库，它用于将抽象语法树结构重新构建为具备可执行特性的源代码，通常与解析器 esprima 配合使用，形成完整的代码解析-修改-生成工作流。它能根据输入的 AST 结构精准还原出符合规范的 JavaScript 源代码，并支持通过配置项控制代码格式（如缩进、分号、引号风格等），常用于构建代码转换工具、编译器或代码自动化处理工具。该库严格遵循 ECMAScript 标准，在保持代码语义一致性的同时，可完整保留原始代码的逻辑结构。

代码 4-2 escodegen 解析 strcat 函数抽象语法树

```
const esprima = require('esprima');
const escodegen = require('escodegen');
const code = `function strcat(name) {
    return ("Hello " + name);
}`;
const ast = esprima.parseScript(code);
var codeFromast=escodegen.generate(ast);
console.log(codeFromast);
```

```
PS D:\Code\Node> node .\test.js
function strcat(name) {
    return 'Hello ' + name;
}
```

图 4-4 运行结果图

代码 4-2 先通过 esprima 库解析 strcat 函数为抽象语法树，再通过 escodegen 库还原为 JavaScript 代码，可以看到还原后的代码没有语法语义信息的损失，我们可以通过解析抽象语法树，遍历 AST 节点，对特定类型节点添加或修改来实现测试用例的变异，通过抽象语法树变异的策略更能满足测试中对 JavaScript 语法规范的要求，具体的变异方法已在上节介绍。

(3) 差分模糊测试模块

差分测试是一种通过对比不同系统或版本对相同输入的输出来检测差异的测试方法。其核心思想是向多个实现相同功能的系统提供相同输入，比较它们的输出或行为差异，通过这种方式，测试人员可以清晰地识别出不同实现之间的差异，从而发现潜在错误。图 4-5 展示了本系统的差分模糊测试流程，具体流程包括输入生成、多个系统的并行执行、输出收集与对比等步骤。

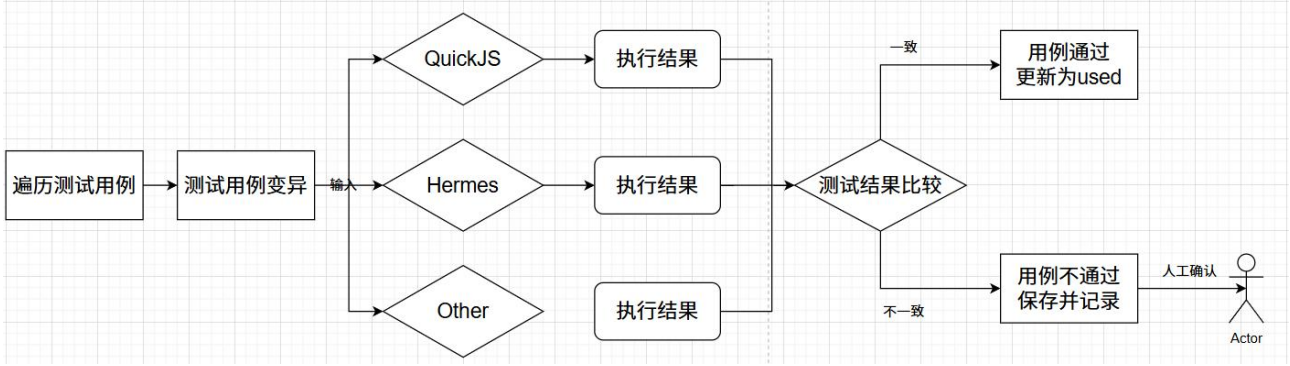

图 4-5 差分模糊测试流程

(4) 测试结果处理模块

测试结果处理模块由俩个子模块构成。分别负责对测试结果进行裁剪以及对无效或冗余信息进行筛除。在执行差分模糊测试的过程中，不同 JavaScript 引擎在持续接收输入用例的情况下，往往会生成数量庞大的输出数据。由于缺陷信息可能混杂于这些测试结果之中，仍需借助人工手段对其进行比对与判别。然而人工分析时受重复测试结果的显著干扰，导致发现未知缺陷的成本大幅增加。由于测试结果复杂多样且可读性差，其重复性判断依赖主观评估，使得去重难度大幅提高。为实现对测试结果的有效去重与缺陷识别，本文设计了一种过滤方法来降低重复结果的测试的影响。具体设计思路如下：首先提取包括异常描述、脚本执行引擎名称、异常函数接口及其对应错误分类等在内的关键信息，并统一进行格式标准化处理。随后，利用这些规范化特征对测试结果进行匹配判断：若与已有记录相符，则视为重复数据而不予保留；若未在历史记录中出现，则暂存该结果以供进一步验证。

(5) 系统界面设计

图 4-6 EJSFuzz 系统主界面

打开 EJSFuzz 系统，首先会进入系统的主界面，如图 4-6 所示。主界面包括测试运行按钮与测试信息输出框。点击开始测试后即可执行模糊测试，并实时显示测试进度与测试输出。下方的详情栏展示了参与差分模糊测试的嵌入式 JavaScript 引擎。

图 4-7 EJSFuzz 系统配置界面

图 4-8 EJSFuzz 测试结果界面

图 4-7 展示了 EJSFuzz 系统的配置界面，其包括数据库配置、引擎配置、输出结果配置三个模块，其中数据库配置支持 Mysql 与 SQLite3 数据库，引擎配置列出了当前所有的 JavaScript 引擎，并提供增删改接口。输出结果配置支持我们自定义模糊测试结果输出位置。

图 4-8 是 EJSFuzz 系统的结果分析界面，当测试完成且测试结果输出后，我们需要对触

发了错误的测试用例进行分析，该模块会获取所有的测试用例，在左侧列出文件名列表，右侧列出具体的测试用例。

4.2 实验设置

4.2.1 实验对象

本文的实验对象为嵌入式 JavaScript 引擎，在本次实验中选取常见的 JavaScript 引擎，具体的清单见表 4.1：

表 4.1 测试对象说明

引擎名	说明	链接
Hermes	Hermes 引擎是 Facebook 开发的一款高性能 JavaScript 引擎。它通过减少内存占用和提升启动速度，增强了移动应用的性能。	https://github.com/facebook/hermes
QuickJs	QuickJS 是由 Fabrice Bellard 开发的高效小型 JavaScript 引擎，具有极快的启动速度和低内存占用，适合嵌入式系统。	https://bellard.org/quickjs/
JerryScript	由三星公司开发的 JerryScript 是一款面向物联网的轻量级 JavaScript 引擎，专为内存受限环境优化。	https://github.com/jerryscript-project/jerryscript
Duktape	Duktape 是一个可嵌入、移植的 JavaScript 引擎，体积小。它适合在资源有限的环境中运行，能够在仅有 160kB 闪存和 64kB RAM 的平台上工作。	https://duktape.org/
MuJS	MuJS 是一个轻量级的嵌入式 JavaScript 解释器。它注重小尺寸、正确性和简单性。	https://github.com/ccxvii/mujs
XS	XS 是 Moddable SDK 中的 JavaScript 引擎，专为微控制器开发而设计。它实现了 2023 年 JavaScript 语言标准，具有超过 99%的兼容性。	https://github.com/Moddable-OpenSource/moddable

4.2.2 实验环境

硬件实验环境如下：

操作系统：Linux Ubuntu 20.04.6 LTS；内核版本：5.4.0-150-generic；

处理器： AMD EPYC 7532，3.3GHz；内存：128G

软件实验环境如下：

EJSFuzz 系统的开发语言为 Python 3.8.10，后端开发使用 Flask 3.0.3 框架，前端开发使用 Vue Vite6.2.3 版本。Node 版本为 v22.14.0。

4.2.3 实验步骤

本论文通过对嵌入式 JavaScript 引擎进行差分模糊测试来挖掘其潜在漏洞，具体实验步骤设计如下：

① 测试用例变异效果评估：将 EJSFuzz 与目前主流的检测方法进行对比，将其他方法和本文用例变异方法生成的测试用例进行比较。

② EJSFuzz 系统评估：使用本文实现的差分模糊测试工具 EJSFuzz 对选用的嵌入式 JavaScript 进行测试，从实验发现的缺陷类型与数量来说明本方法的有效性。

③ 对比实验评估：使用 EJSFuzz 系统与其他模糊测试工具在相同的环境与原始语料下同步测试，通过覆盖率、缺陷数量等指标来评估各个工具挖掘缺陷的能力。

4.2.4 评估指标与对比方法

评估模糊测试包括多种指标，常见的有以下几种：

1) 代码覆盖率：在评估模糊测试性能时，代码覆盖率常被用作核心评估参数之一。该指标反映了测试用例在执行期间能够触达的程序路径相对于所有可能路径的比例。依据检测粒度的不同，覆盖率通常可分为以下几种类型：语句覆盖率，关注所有语句是否被运行；分支覆盖率，每个判断条件的所有分支是否均被执行；路径覆盖率，程序中所有的执行路线是否被执行。例如，若某程序包含 1000 行代码，而模糊测试仅触发其中 600 行的执行，则代码覆盖率为 60%。高覆盖率通常意味着测试更充分，能够暴露更多潜在漏洞。

2) 测试用例生成效率：测试用例生成效率反映用例生成模型在单位时间内生成有效测试用例的能力，通常以（用例数/秒）衡量。高效的模糊器应具备高吞吐量、低冗余性等特点。

3) 漏洞发现率：漏洞发现率用于衡量某种检测手段对目标程序中已知安全缺陷的识别覆盖程度。更具体地说，它是指被测系统中，检测工具成功定位出的漏洞数量与系统内预先确认的漏洞总数之间的比例。例如，若某软件包含 20 个已知安全问题，而某检测工具能够发现其中的 4 个，那么该工具的漏洞识别率即为 20%。因此，漏洞发现率越高，意味着该工具具备更强的缺陷检测能力。

为了进一步说明本文所提出方法的有效性，需要引入对比实验，将 EJSFuzz 与其他模糊测试工具进行对比十分必要。本文选择了 AFL 与 Fuzzilli 这俩个模糊测试工具，相关工具的说明见下表 4.2：

表 4.2 对比实验对象说明

模糊测试工具	测试方法说明	链接
AFL	AFL 采用基于遗传策略的自动化方法生成测试输入样本，依据路径遍历信息与覆盖率统计数据迭代优化流程，不断优化测试用例。	https://github.com/google/AFL
Fuzzilli	Fuzzilli 定义了一种中间语言，采用基于覆盖率的指导方法，通过自定义中间语言（FuzzIL）来生成和变异测试用例。	https://github.com/googleprojectzero/fuzzilli

通过分析测试结果与对比实验，能准确评估 EJSFuzz 系统的模糊测试能力，并根据评估指标与具体漏洞，改进原型系统，进一步提升其模糊测试的能力。

4.3 实验结果分析

本实验对各大主流嵌入式 JavaScript 引擎进行模糊测试，以下从测试用例变异、模糊测试结果评估、对比实验结果等方面对实验结果进行分析评估。

4.3.1 测试用例变异实验

参数类型推断的准确性对后续测试用例的变异起着至关重要的作用，为了验证本文所提出的参数类型推断方法的有效性，本文设计了对照实验来评估该方法的表现。

我们采用标准的对照实验设计，分为实验组和对照组，实验组使用我们提出的参数类型推断方法，对照组则采用传统的随机参数传递方法，俩个组采用相同的输入配置：一千份测试用例。俩个组的输出设置为五大 JavaScript 数据类型：Array，Boolean、Number、Function、String。在实验初期，通过人工分析提前对输入测试用例的参数类型进行统计，即可在实验结束后自动计算出准确率，最终得到的实验结果如下：

表 4.3 参数类型推断对比实验结果

分组	类型准确率				
	Array	Boolean	Number	Function	String
对照组	21.28%	9.94%	19.84%	15.32%	20.54%
实验组	86.71%	84.04%	90.10%	93.65%	89.79%

实验结果如表 4.3 所示，由于对照组采用随机策略，导致其类型准确率普遍偏低，范围为 9.94%~21.28%。而在实验组中，各数据类型的准确率明显高于对照组。

代码 4-3 用例变异前

```
 var FuzzingFunc =
function(data) {
    var b = data.slice().sort();
    return exports.interp(b, .5);
}
;
```

代码 4-4 用例变异后

```
 var FuzzingFunc =
function(data) {
    //array method mutation
    var count=data.length;
    for (let i = 0; i < 100; i++) {
        for (let j = 0; j <count ; j++) {
            data.push(data[j]);
        }
    }
    for (i = 0; i < 1e4; i++) {

data.unshift("Lemon","Pineapple");
    }
    var b = data.slice().sort();
    //end
    return exports.interp(b, .5);
}
```

基于上述参数类型变异实验，我们进一步将此策略应用于测试用例变异模块，聚焦于五类参数类型的修改操作。代码 4-3 与代码 4-4 中分别呈现了变异实施前后，测试用例结构和内容的具体差异。

可以看到，在推断出 data 参数的类型为 Array 后，用例变异程序对 data 参数进行了多种变异，追加原始数组内容 100 次，在 data 数组前端添加了一万次“Lemon”和“Pineapple”字符串，最后创建了一个数组副本并排序。通过上述这些操作，使 data 数组的大小快速膨胀，从而有触发引擎潜在缺陷的可能。

在对 JavaScript 引擎进行模糊测试时，由于引擎通常会对输入代码进行严格的语法解析，若测试用例未能遵循 JavaScript 语言的结构和语义规范，将难以通过语法层面的有效性验证。因此，生成的代码片段必须满足基本的语言规则，以确保模糊测试顺利执行。JShint 是一款 JavaScript 代码静态语法检测工具，我们可以通过这款工具静态检查变异后的测试用例，统计其中语法正确的测试用例，依据公式（4-1），计算语法正确的测试用例在总变异测试用例里的比例，以此衡量变异测试用例的质量。

$$\text{语法正确性}(\alpha)=\frac{\text{语法正确的测试用例}}{\text{变异的测试用例总数}} \tag{4-1}$$

通过 JShint 工具验证与人工分析，在随机输入的 1000 个测试用例中，78 个原始测试用

例未通过语法检测，85 个测试用例函数参数为空，未参与变异，837 个测试用例进行参数类型推断后执行变异，其中 661 个测试用例语法正确，176 个测试用例未通过语法检查。语法正确性为 78.97%，相比于传统的逐字节变异方法正确性有了明显的提升。

4.3.2 差分模糊测试效果评估

一个差分模糊测试系统的主要目的是有效地触发被测试引擎中的缺陷，本节使用 EJSFuzz 系统对 4.1.1 介绍的实验对象进行模糊测试，通过分析 EJSFuzz 系统所检测到的引擎缺陷数量，来说明本文方法的有效性。具体的触发情况如表 4.4 所示：

表 4.4 引擎缺陷触发情况

引擎名称	缺陷数量
JerryScript	2
MuJs	3
XS	3
QuickJs	1
Duktape	2
Hermes	2

通过分析差分模糊测试结果，下面对本次实验中触发的崩溃案例进行分析，探究其缺陷触发的场景与原因。

一、崩溃案例分析

触发 JerryScript 崩溃的测试用例如代码 4-5 所示，它的参数 data 类型推断结果为 Array，执行过 Array 变异。首先简单介绍该测试用例的语义：第一行声明了一个数组排序函数 arraySort，它有一个参数 data。函数体内，首先是对函数参数的变异，它将 data 数组的原始内容追加了 1000 次，使得数组的大小膨胀为原来的 1001 倍，最后调用数组的 sort 方法并返回排序后的数组。函数体外，首先定义了一个空数组 a，接着向数组 a 里填充了 1000 个 0 到 100 的随机数，然后调用这个函数。最后是一个打印语句，表明如果该测试用例正常执行，那么则输出“right”字符串到控制台。

代码 4-5 变异后的 arraySort 函数

```
function arraySort(data) {
    //array method mutation
    var count = data.length;
    for (var i = 0; i < 1000; i++) {
        for (var j = 0; j < count; j++) {
            data.push(data[j]);
        };
    };
    //end
    print(data.length);
    data.sort();
    return data;
};
var a = [];
for (var i = 0; i < 1000; i++) {
    a.push(Math.floor(Math.random() * 100));
}
var customArray = arraySort(a);
Console.log("right");
```

图 4-9 展示了 arraySort 测试用例触发了 JerryScript 引擎的缺陷而导致崩溃并且没有任何错误输出，而其他引擎则正常打印出数组 a 的大小。这表明 JerryScript 引擎的 sort 实现与其他引擎不同，并且存在缺陷。

```
root@5d15f714c376:/home/EmbeddedFuzzer# python test.py /home/htm/1.js
|---engine---|---output---|---error---|
|---hermes---|---1001000---|---none---|
|---engine---|---output---|---error---|
|---qjs---|---1001000---|---none---|
|---engine---|---output---|---error---|
|---jerry---|------|---触发崩溃---|
|---engine---|---output---|---error---|
|---xst---|---1001000---|---none---|
|---engine---|---output---|---error---|
|---duk---|---1001000---|---none---|
|---engine---|---output---|---error---|
|---mujs---|---1001000---|---none---|
```

图 4-9 案例触发引擎崩溃缺陷

下面我们分析该案例造成 JerryScript 引擎崩溃的原因：JerryScript 引擎的代码开源在 GitHub 上，其 sort 的实现逻辑位于\jerry-core\ecma\builtin-objects\ecma-builtin-array-prototype.c，当 JerryScript 将未排序的数组拷贝到内部的排序缓冲区时，由于数组的大小过大，导致产生

JERRY_FATAL_OUT_OF_MEMORY 错误，即没有足够的内存分配给该缓冲区。通过打印 Linux 的特殊变量$?与 JerryScript 的错误消息结构体定义也可佐证这一点。

```
typedef enum
{
  JERRY_FATAL_OUT_OF_MEMORY = 10, /**< Out of memory */
  JERRY_FATAL_REF_COUNT_LIMIT = 12, /**< Reference count limit reached */
  JERRY_FATAL_DISABLED_BYTE_CODE = 13, /**< Executed disabled instruction */
  JERRY_FATAL_UNTERMINATED_GC_LOOPS = 14, /**< Garbage collection loop limit reached */
  JERRY_FATAL_FAILED_ASSERTION = 120 /**< Assertion failed */
} jerry_fatal_code_t;
```

图 4-10 JerryScript 错误消息结构体定义

二、功能缺陷案例分析

JavaScript 引擎功能的正确性直接影响到应用的稳定性与安全性，错误的计算或处理结果会导致逻辑错误，进而产生严重的缺陷甚至漏洞。同时由于功能缺陷难以调试和排查，开发人员需要花费大量时间修复，对软件生态有严重的影响。如代码 4-6 所示，该测试用例触发了 Duktape 引擎的功能缺陷。该测试用例的语义如下：1-4 行定义了一个函数 FuzzingFunc，函数体内定义了一个变量 num 并赋值为 1，接着返回 num 变量与它自增的差值。第 5 行调用 FuzzingFunc 函数并将结果赋值给变量 CallingResult，第 6 行打印 CallingResult 的值。测试用例的差分运行结果如图 4-11 所示，其他引擎都正确打印了 0，但 Duktape 引擎错误地返回了 1。

代码 4-6 FuzzingFunc

```
var FuzzingFunc = function () {
   var num = 1;
   return num - num++;
};
print(FuzzingFunc());
```

```
root@5d15f714c376:/home/EmbeddedFuzzer# python test.py  /home/htm/24-duktape.js
|---engine---|---output---|---error---|
|---hermes---|---0---|---none---|
|---engine---|---output---|---error---|
|---qjs---|---0---|---none---|
|---engine---|---output---|---error---|
|---jerry---|---0---|---none---|
|---engine---|---output---|---error---|
|---xst---|---0---|---none---|
|---engine---|---output---|---error---|
|---duk---|---1---|---none---|
|---engine---|---output---|---error---|
|---mujs---|---0---|---none---|
```

图 4-11 案例触发引擎功能缺陷

经分析，Duktape 引擎在执行 num - num++这一操作时，首先执行了 num++操作，num++操作会先赋值右操作数为 1，然后进行自增，然而自增操作会影响左操作数 num 值变为 2，而后进行减法运算得到错误结果 1。正确的操作顺序为将左操作数 num=1 压入栈或存入寄存器，而后对 num 变量执行赋值自增操作，此时对 num 的修改不会影响到已经存入栈或寄存器里的值，且右操作数为先赋值后自增，它的值也为 1。最后取出左右操作数和操作符，计算 1-1 得到正确结果 0。

4.3.3 对比实验分析

为进一步验证本文方法的有效性，本文将 EJSFuzz 与其他模糊测试工具进行对比实验，本文选择 AFL 与 Fuzzilli 作为对照组，其中 AFL 是经典的模糊测试工具，Fuzzilli 也是近年来最先进的 JavaScript 引擎模糊测试工具之一。通过对比三种测试方法在代码覆盖率、测试用例生成效率、漏洞发现率来评估不同方法的优劣。

AFL 的工作原理已在 2.3 节中介绍，在此简要介绍 Fuzzilli 的工作原理:

- Fuzzilli 定义了一种中间语言 FuzzIL，将 JavaScript 代码转换为 FuzzIL 来进行后续变异，FuzzIL 能反映函数调用、循环等操作，具有易于静态推理、易于转换为 JavaScript 代码的特点。图 4-12 展示了从 FuzzIL 转换为 JavaScript 的示例。
- FuzzIL 有四种基本突变方法：输入变异、操作符变异、拼接变异、代码生成。
- Fuzzilli 通过覆盖率反馈来筛选有效测试用例，通过插桩监控引擎的代码覆盖路径，保留触发新覆盖路径的测试用例，并进一步变异探索新覆盖路径。

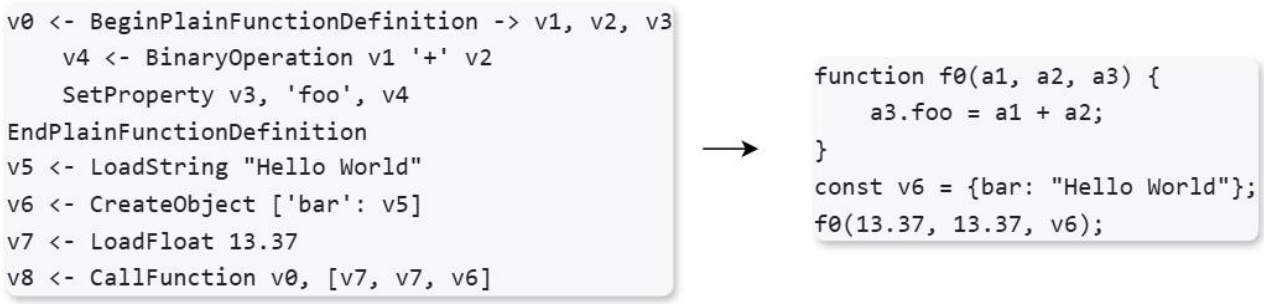

图 4-12 FuzzIL 与 JavaScript 转换图

代码覆盖率作为衡量测试用例执行效果的重要参数，能够较为直观地反映其在测试过程中的使用效率。通常而言，当覆盖率提升时，说明测试过程中已遍历更为广泛的程序路径，从而增强了测试用例在缺陷挖掘的有效性。通过对比不同工具所生成的测试用例的覆盖率，可以评估各测试方法在生成测试用例质量方面的优劣。具体的实验步骤为：对三个工具输入相同

的 1000 条用例，并各自运行变异生成测试用例，最后计算各自的覆盖率并统计。

图 4-13 测试用例通过率与覆盖率结果对比

统计的结果如图 4-13 所示，可以看到 EJSFuzz 生成的测试通过率比 Fuzzilli 高，而覆盖率比略低于 Fuzzilli，这是由于 Fuzzilli 是以覆盖率为导向的变异，因此生成的用例覆盖率高。而 AFL 俩种指标都偏低，这是因为其作为一个通用模糊测试工具，无法有效处理 JavaScript 语言丰富的特性。测试用例生成效率通常指单位时间内工具生成的用例数，能反映工具用例变异策略的高效性与有效性，实验的结果如图 4-14 所示。

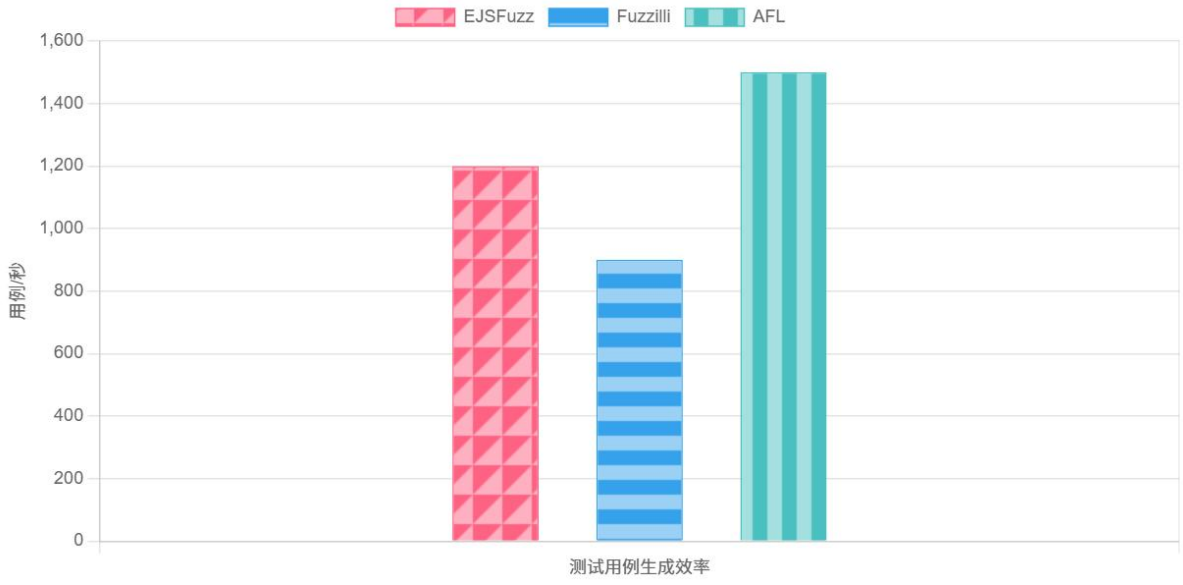

图 4-14 测试用例生成效率结果对比

结合测试用例的通过率、覆盖率与生成效率，我们可以看到 EJSFuzz 生成的测试用例质

量各方面都有较为均衡的表现，为了进一步验证各模糊测试工具的缺陷检测能力，我们设计缺陷数量对比实验，即对比一定时间内不同工具触发的缺陷数量。

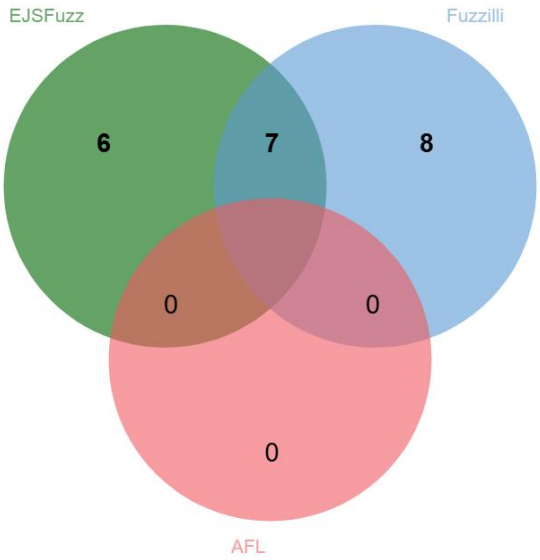

图 4-15 缺陷发现数量结果对比

图 4-15 展示了这次的对比实验结果，可以看到 Fuzzilli 检测到的缺陷最多，数量为 15 个，EJSFuzz 紧随其后，检测到 13 个，而 AFL 未能检测出缺陷。这样的结果并不意外，因为 Fuzzilli 的工作原理依赖于对 JavaScript 引擎源代码插桩从而实现覆盖率追踪，具有高度的定制性，对被测系统有较高的要求。AFL 作为经典模糊测试工具在本次实验中的失效，则揭示了通用模糊测试在复杂语言环境的局限性。而 EJSFuzz 作为一种黑盒测试工具，结合类型推断的变异策略，既保持了黑盒方法的通用性，又显著提升了测试用例的有效性。最终让 EJSFuzz 在黑盒环境下，仍能实现接近白盒工具的检测效率，证明了本文提出的基于类型推断的差分模糊测试方法的有效性。

5 结论与展望

5.1 结论

物联网的飞速发展给 JavaScript 社区带来了新的活力与挑战。作为 Web 生态的核心语言，JavaScript 凭借其动态特性、丰富的 API 支持以及庞大的开发者社区，正逐步向嵌入式领域扩展。这一趋势催生了众多优秀的嵌入式 JavaScript 引擎，随着各类引擎的广泛应用与功能性的不断提升，嵌入式生态对 JavaScript 引擎安全性与可靠性的要求也在快速提高，作为嵌入式生态的基础设施，JavaScript 引擎的缺陷会导致大量的嵌入式设备面临风险，一个小的功能缺陷会让整个程序运行错误，性能缺陷会让本就资源受限的嵌入式设备难以正常工作，而安全缺陷甚至会成为物联网被攻破的入口。为了对嵌入式 JavaScript 引擎高效的测试，本文提出了一种基于类型推断的测试用例变异方法，并在此方法的基础上实现了 EJSFuzz 系统。

具体的研究内容如下：

（1） 本文研究了当前具有代表性的软件模糊测试技术，并进一步对 JavaScript 引擎模糊测试方向已有的研究成果进行了归纳分析。测试用例生成模型很大程度上决定了模糊测试方法的有效性，当前主流的方法分为生成算法与变异算法，生成式算法很难生成涵盖语言的丰富特性，而传统的变异算法基于字节变异，无法兼顾用例语法语义的正确性。

（2） 本文提出了基于类型推断的测试用例变异方法。JavaScript 语言动态类型的特点使其无法在运行前进行静态类型检查。而类型推断模块通过分析参数在函数内部的操作模式，确定其数据类型，本文还根据数据类型的不同设计了丰富的变异策略，提升了测试用例的代码覆盖率。

（3） 本文基于上述的测试用例变异方法实现了 EJSFuzz 系统，对系统内部各个模块做了简要介绍，并对关键的参数类型推断和用例变异算法进行详细描述。使用 EJSFuzz 对多款嵌入式 JavaScript 引擎进行缺陷检测，并与多款模糊测试工具进行对比实验，以代码覆盖率、测试用例生成效率、漏洞发现率为评估指标，最终的实验结果说明了本文模糊测试方法与 EJSFuzz 系统的有效性。

5.2 展望

相比于传统的模糊测试工具，本文提出的基于类型推断的测试用例变异方法与 EJSFuzz 系统测试效率更高效，但还是存在可以改进的地方，具体包含以下几个方面：

（1）本文结合了差分测试与模糊测试对嵌入式 JavaScript 引擎进行测试，然而差分测试存在一个局限，即当不同引擎对同一测试用例产生相异的输出时才能有效识别潜在缺陷，但

如果被测引擎对某一语言特性的实现均存在错误，则会产生相同的错误输出，导致此类缺陷无法被检测。虽然各引擎同时出现相同错误的概率较低，但该问题确实存在，后续研究需探索针对此类情况的检测方案。

（2）本文提出的类型推断方案通过静态的参数行为计数来实现，可推断的参数类型受条件限制未能涵盖 Object 类型以及其他嵌套类型，而随着机器学习技术的飞速发展，可以通过训练相关模型来实现参数类型的推断，从而增加测试用例的变异方向。

（3）在对触发缺陷的测试用例进行分析时，由于引擎触发崩溃导致执行中断，给缺陷定位与分析带来了巨大的挑战，后续应优化差分测试模块，设计一套故障分析方案，用于记录触发崩溃的调用栈、内存快照等关键信息。

参考文献

[1] Holler C, Herzig K, Zeller A. Fuzzing with code fragments[C]//21st USENIX Security Symposium (USENIX Security 12). 2012: 445-458.

[2] 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 (USENIX Security 20). 2020: 2613-2630.

[3] Yang C, Deng Y, Lu R, et al. Whitefox: White-box compiler fuzzing empowered by large language models[J]. Proceedings of the ACM on Programming Languages, 2024, 8(OOPSLA2): 709-735.

[4] Godefroid P, Klarlund N, Sen K. DART: Directed automated random testing[C]//Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. 2005: 213-223.

[5] 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.

[6] SOYEON PARK, WEN XU, INSU YUN, et al. Fuzzing JavaScript Engines with Aspect-preserving Mutation[C]//2020 IEEE Symposium on Security and Privacy: IEEE Symposium on Security and Privacy (SP 2020), 18-21 May 2020, San Francisco, CA, USA.:Institute of Electrical and Electronics Engineers, 2020:1629-1642.

[7] Dinh S T, Cho H, Martin K, et al. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases[C]//NDSS. 2021.

[8] Xu H, Jiang Z, Wang Y, et al. Fuzzing JavaScript Engines with a Graph-based IR[C]//Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. 2024: 3734-3748.

[9] Groß S, Koch S, Bernhard L, et al. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities[C]//NDSS. 2023.

[10] Wang J, Zhang Z, Liu S, et al. {FuzzJIT}:{Oracle-Enhanced} Fuzzing for {JavaScript} Engine {JIT} Compiler[C]//32nd USENIX Security Symposium (USENIX Security 23). 2023: 1865-1882.

[11] Bin, Zhang,Jiaxi, Ye,Xing, Bi, 等 .Ffuzz: Towards full system high coverage fuzz testing on binary executables.[J].PLoS ONE.2018,13(5).e0196733.DOI:10.1371/journal.pone.0196733 .

[12] Klaus Greff,Rupesh K. Srivastava,Jan Koutník,等.LSTM: A Search Space Odyssey[J].IEEE Transactio ns on Neural Networks and Learning Systems","pubMedId":"27411231.2017,28(10).2222-2232.DOI:10.1109/ TNNLS.2016.2582924 .

[13] Veggalam S, Rawat S, Haller I, et al. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming[C]//Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21. Springer International Publishing, 2016: 581-601.

[14] Han H S, Oh D H, Cha S K. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines[C]//NDSS. 2019.

[15] 喻垚慎，黄志球，沈国华，等. 基于抽象解释的嵌入式软件模块化 Cache 行为分析框架. 计算机学报, 2019, 42(10): 2251-2266

[16] 徐浩然,王勇军,黄志坚,等. 基于前馈神经网络的编译器测试用例生成方法[J]. 软件学报,2022,33(6):1996-2011. DOI:10.3969/j.issn.1000-9825.2022.06.004.

[17] 杨克,贺也平,马恒太,等. 有效覆盖引导的定向灰盒模糊测试[J]. 软件学报,2022,33(11):3967-3982. DOI:10.13328/j.cnki.jos.006331.

[18] 喻波,苏金树,杨强,等. 网络协议软件漏洞挖掘技术综述[J]. 软件学报,2024,35(2):872-898. DOI:10.13328/j.cnki.jos.006942.

[19] 杨克,贺也平,马恒太,等. 有效覆盖引导的定向灰盒模糊测试[J]. 软件学报,2022,33(11):3967-3982. DOI:10.13328/j.cnki.jos.006331.

[20] 梁杰,吴志镛,符景洲,等. 数据库管理系统模糊测试技术研究综述[J]. 软件学报,2025,36(1):399-423. DOI:10.13328/j.cnki.jos.007048.

[21] 杨克,贺也平,马恒太,等. 面向递增累积型缺陷的灰盒模糊测试变异优化[J]. 软件学报,2023,34(5):2286-2299. DOI:10.13328/j.cnki.jos.006491.

[22] 崔展齐,张家铭,郑丽伟,等. 覆盖率制导的灰盒模糊测试研究综述[J]. 计算机学报,2024,47(7):1665-1696. DOI:10.11897/SP.J.1016.2024.01665.

[23] 王琴应,许嘉诚,李宇薇,等. 智能模糊测试综述:问题探索和方法分类[J]. 计算机学报,2024,47(9):2059-2083. DOI:10.11897/SP.J.1016.2024.02059.

[24] 余媛萍,苏璞睿. HeapAFL:基于堆操作行为引导的灰盒模糊测试[J]. 计算机研究与发展,2023,60(7):1501-1513. DOI:10.7544/issn1000-1239.202220771.

[25] 况博裕,张兆博,杨善权,等. HMFuzzer:一种基于人机协同的物联网设备固件漏洞挖掘方案[J]. 计算机学报,2024,47(3):703-716. DOI:10.11897/SP.J.1016.2024.00703.

致　谢

行文至此，我的四年本科生涯即将结束，我的求学之路也要告一段落。

感谢叶贵鑫老师对我论文的悉心指导，从论文选题、修改到最后论文的完成，每一个环节都离不开您细心的指导和宝贵的建议，祝老师万事顺遂，桃李芬芳。

感谢我的朋友们，祝我们的友谊长存。

感谢我的家人，你们的陪伴是我求学四年来的支持与动力。

感谢自己。