

分类号: TP391

学校代码: 10697

密 级: 公开

学 号: 201821022



西北大学
Northwest University

硕士学位论文

MASTER' S DISSERTATION

基于标准文档分析的 JavaScript 引擎缺陷检测方法研究

学科名称: 计算机应用技术

作 者: 田 洋

指导老师: 汤战勇 教授

西北大学学位评定委员会

二〇二一年

Research on JavaScript Engine Bug Detection Method Based on Standard Document Analysis

A thesis submitted to
Northwest University
in partial fulfillment of the requirements
for the degree of Master
in Computer Applied Technology

By
Tian Yang
Supervisor: Tang Zhanyong Professor
June 2021

西北大学学位论文知识产权声明书

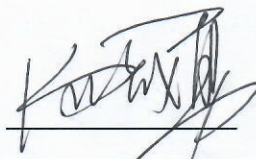
本人完全了解西北大学关于收集、保存、使用学位论文的规定。学校有权保留并向国家有关部门或机构送交论文的复印件和电子版。本人允许论文被查阅和借阅。本人授权西北大学可以将本学位论文的全部或部分内 容编入有关数据库进行检索,可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。同时授权中国科学技术信息研究所等机构将本学位论文收录到《中国学位论文全文数据库》或其它相关数据库。

保密论文待解密后适用本声明。

学位论文作者签名:

田 洋

指导教师签名:



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:

田 洋

2021 年 6 月 8 日

摘要

JavaScript 是一种流行的、平台无关的编程语言，为了保证 JavaScript 程序在不同平台上的互操作性，JavaScript 解释器（即引擎）的实现必须符合 ECMAScript-262 标准。然而，标准的频繁变动使得引擎开发者往往难以做出及时的更新，从而使 JavaScript 引擎出现不符合标准的行为，即一致性缺陷。一致性缺陷不但会导致正确的 JavaScript 脚本在运行后得出错误的结果，还有可能造成在某个平台下能正常运行的 JavaScript 程序迁移到其他平台后出现异常，严重危害 JavaScript 程序的稳定性和拓展性。

为了有效地对 JavaScript 引擎的一致性缺陷进行检测，本文设计了一种基于标准文档分析和差分模糊测试的自动化缺陷检测方法，主要研究内容如下：

（1）本文采用差分模糊测试的思想，通过构造大量测试用例输入到多个 JavaScript 引擎中执行，监测引擎的执行结果并对其进行差分分析，以得到触发引擎不一致行为的可疑用例，最后对可疑用例进行人工分析来确定是否触发了引擎缺陷。该方法自动化程度较高，仅需要少量的人工介入，应用此方法能够对 JavaScript 引擎进行持续地自动化缺陷检测。

（2）利用 ECMAScript-262 标准对于一致性缺陷检测的指导作用，本文设计了一种基于标准文档分析的用例突变方法，该方法能够从标准文档中自动地解析出有效的语义信息，以提升测试用例突变的导向性，突变后的用例能够覆盖引擎的更多分支，从而触发更多的引擎一致性缺陷，提高检测效率。

（3）本文设计并实现了一个一致性缺陷检测原型系统 ESfunfuzz，并利用该系统对当前四个主流的 JavaScript 引擎进行了真实环境下的测试，成功检测出各类引擎的一致性缺陷共 22 个，其中 19 个已被引擎开发商确认。实验结果表明，基于标准文档的自动化测试方法有效、可靠。

关键词：JavaScript 引擎， 一致性缺陷， 标准文档分析， 差分模糊测试

ABSTRACT

JavaScript is a popular, platform-independent programming language. In order to ensure the interoperability of JavaScript programs on different platforms, the implementation of the JavaScript interpreter (i.e., engine) must conform to the ECMAScript-262 standard. However, the frequent changes of the standard make it difficult for engine developers to make timely updates, so that the JavaScript engine has behaviors that do not meet the standards, that is, conformance bugs. Conformance bugs will not only cause correct JavaScript scripts to get wrong results after running, but also may cause abnormal JavaScript programs that can run normally under a certain platform to migrate to other platforms, seriously endangering the stability and extensibility of JavaScript programs.

In order to effectively detect the conformance bugs of the JavaScript engine, this thesis designs an automated bug detection method based on standard document analysis and differential fuzzing. The main research contents are as follows:

(1) This thesis adopts the idea of differential fuzzing, by constructing a large number of test cases and inputting them to multiple JavaScript engines for execution, monitoring the execution results of the engines, and performing differential analysis on them, to obtain suspicious test cases that trigger the inconsistent behavior of engines, finally, artificial analysis was carried out on suspicious cases to determine whether to trigger the engine bug. With a high degree of automation and only a small amount of manual intervention, this method can be applied to the continuous automatic bug detection of JavaScript engines.

(2) Taking advantage of the guiding role of ECMAScript-262 standard for conformance bug detection, this thesis designs a test case mutation method based on standard document analysis, which can automatically parse effective semantic information from standard documents to improve mutation orientation of test cases, the test cases after mutation can cover more branches of the engine, thereby triggering more engine conformance bugs and improving detection efficiency.

(3) This thesis designs and implements a prototype system of conformance bug detection, ESfunfuzz, and uses it to perform bug detection in the current four mainstream JavaScript

engines in a real environment. A total of 22 conformance bugs of various engines have been successfully detected, 19 of which have been confirmed by the engine developers. The experimental results show that the automated testing method based on standard documents is effective and reliable.

Keywords: JavaScript engine, Conformance Bug, Standard Document Analysis,
Differential Fuzzing

目录

摘要	I
ABSTRACT.....	III
目录	V
第一章 引言	1
1.1 研究背景与意义	1
1.1.1 JavaScript 引擎的一致性缺陷.....	1
1.1.2 研究目的与意义	3
1.2 国内外研究现状	3
1.3 本文研究内容	5
1.4 本文组织结构	6
第二章 相关理论与技术	9
2.1 JavaScript 语言背景	9
2.1.1 ECMAScript 语言及其规范	9
2.1.2 JavaScript 语言及其引擎.....	10
2.1.3 一致性缺陷的定义	11
2.2 测试相关技术	11
2.2.1 模糊测试	11
2.2.2 差分测试	13
2.2.3 差分模糊测试	14
2.3 基于语言模型的代码生成	15
2.3.1 统计语言模型	15
2.3.2 神经语言模型	16
2.3.3 基于语言模型的代码生成任务	16
2.4 本章小结	17
第三章 基于标准文档分析的测试用例定向突变方法	19
3.1 测试用例突变方法概述	19
3.2 基于标准文档分析的用例定向突变方法概述	20

3.3	标准文档解析方法	22
3.3.1	标准文档解析方法概述	22
3.3.2	标准转化方法	23
3.3.3	标准转化方法的效果评估	25
3.3.4	语义信息提取方法	27
3.3.5	参数类型推断及非预期类型变异	28
3.4	语义信息指导的用例定向突变算法	28
3.5	本章小结	30
第四章	基于差分模糊测试的一致性缺陷检测方法	33
4.1	方法概述	33
4.2	生成模型构建	34
4.2.1	语料库收集	35
4.2.2	数据预处理	35
4.2.3	模型构建和训练	36
4.3	测试用例生成方法	37
4.3.1	种子用例收集	37
4.3.2	语义信息指导的用例突变生成方法	38
4.3.3	基于神经语言模型的用例续写生成方法	39
4.4	差分模糊测试方法设计	40
4.4.1	模糊测试的执行结果说明	40
4.4.2	差分机制设计	41
4.4.3	种子池扩充方法设计	43
4.4.4	人工分析	44
4.5	本章小结	44
第五章	实验评估与分析	47
5.1	ESfunfuzz 原型系统的设计与实现	47
5.1.1	系统模块设计	47
5.1.2	系统具体实现	48
5.2	实验环境与实验设置	49
5.2.1	实验环境	49

5.2.2 实验设置	49
5.3 语言模型调优实验	50
5.3.1 语言模型性能的衡量标准	50
5.3.2 数据编码级别实验	51
5.4 与其他工具的对比实验	52
5.4.1 生成用例质量对比	53
5.4.2 缺陷检测能力对比	54
5.5 真实环境下的缺陷检测实验	56
5.6 典型案例研究	57
5.6.1 案例研究（一）——续写生成方法的有效性验证	58
5.6.2 案例研究（二）——定向突变方法的有效性验证	59
5.6.3 案例研究（三）——非预期类型变异生成方法的有效性验证	60
5.7 本章小结	60
总结与展望	63
参考文献	65
攻读硕士学位期间取得的科研成果	69
1. 申请（授权）专利	69
2. 参与科研项目及科研获奖	69
致谢	71

第一章 引言

1.1 研究背景与意义

1.1.1 JavaScript 引擎的一致性缺陷

JavaScript 编程语言是一种流行的、平台无关的前端脚本语言，它普遍地应用于 Web 页面中，承担着用户与页面进行动态交互的重要作用^[1]。JavaScript 语言的成功离不开性能强大的 JavaScript 引擎的支持，JavaScript 引擎是嵌入在 Web 浏览器内部负责解释执行 JavaScript 代码的一类计算机程序，是现代浏览器最重要的组成部分之一。为了保证 JavaScript 程序在不同浏览器平台上的互操作性，JavaScript 引擎的实现必须符合 ECMAScript-262 标准（以下简称“ES 标准”），ES 标准中定义了 JavaScript 语言所需要满足的所有功能及其具体实现细节。

然而，如表 1 所示，自 2015 年开始 ES 标准便保持着每年一版的频率进行更新^[2]，如此之快的更新频率使得引擎的开发者们难以做出及时的更新，从而使得 JavaScript 引擎表现出未按标准实现的问题，本文称之为引擎的“一致性缺陷”。

表 1 ECMAScript-262 标准版本发布时间

版本名称	发布时间
ECMA-262 5.1 th edition	2011 年 6 月
ECMA-262 6 th edition	2015 年 6 月
ECMA-262 7 th edition	2016 年 6 月
ECMA-262 8 th edition	2017 年 6 月
ECMA-262 9 th edition	2018 年 6 月
ECMA-262 10 th edition	2019 年 6 月
ECMA-262 11 th edition	2020 年 6 月

ES 标准内容的频繁变更是 JavaScript 引擎产生一致性缺陷的主要原因。例如在对 Object.keys 这一方法的实现定义上，ES5.1 标准和 ES6.0 标准就给出了两种不同的处理方式。如图 1 所示，当传入的参数 O 不是 Object 类型时，ES5.1 标准规定引擎立即抛出一个 TypeError 异常；而 ES6.0 标准则要求引擎对 O 执行 ToObject 操作，将其强制转换为一个 Object 对象，并在此基础上进行后续的操作，而不会抛出异常。显然，

这两种处理方式是截然相反的,倘若 JavaScript 引擎的开发未能及时地按照变更后的 ES 标准对引擎进行相应的更新,那么就会出现一致性缺陷。

Object.keys (O) 1. If the Type(O) is not Object , throw a TypeError exception. 2. Let <i>n</i> be the number of own enumerable properties of O. 3. ...	Object.keys (O) 1. Let <i>obj</i> be ToObject (O). 2. ReturnIfAbrupt(<i>obj</i>). 3. ...
(a) ES5.1 标准中的定义 ^[3]	(b) ES6.0 标准中的定义 ^[4]

图 1 不同版本的标准对 Object.keys 函数的不同定义

一致性缺陷的后果是致命的。一方面,它可能导致正确的 JavaScript 脚本在运行后得到错误的结果,如图 2(a)所示,该脚本旨在用某正则表达式 *reg* 去字符串 *str* 中查找某些内容,并根据查找到的结果 *flag* 来执行某些操作。然而,由于 JavaScriptCore 引擎的正则模块出现了一致性缺陷,导致脚本在该引擎下执行后得到的 *flag* 变量值为 -1,而正确的值是 0,这一错误的结果直接使得第 5 行中的代码无法执行,从而对脚本结果产生严重影响。

另一方面,一致性缺陷还可能导致在某个平台下正常运行的 JavaScript 程序在迁移到其他平台后出现异常。如图 2(b)所示,该脚本在微软公司的浏览器平台(如 Edge 和 IE 浏览器)下运行时,由于该平台内嵌的 ChakraCore 引擎缺乏对于“if(1)”这类语法错误的检查,导致该脚本能够正常运行而不会抛出任何异常,这会误导开发者认为该脚本是正确的。然而,当用户使用其他浏览器执行该脚本时则会直接抛出语法错误并终止执行,此类代码一旦上线必然会引发严重的后果。因此,引擎的一致性缺陷亟待解决。

<pre> 1 var reg = /(.*d.*){5,}/; 2 var str = 'adadshare.com/files/61674290/ a.a.T hird_GG_SG_part lasPl ayground.zip C:\ Documents and Settings'; 3 var flag = str.search(reg); 4 if (flag != -1) { 5 // 其他操作 6 } </pre>	<pre> 1 var count = 0; 2 [].length = { 3 valueOf: function() { 4 count++; 5 return 1; 6 } 7 }; 8 if (1) // 此处应该抛出语法错误 </pre>
(a) JavaScriptCore 引擎缺陷实例 ^[5]	(b) ChakraCore 引擎缺陷实例 ^[6]

图 2 引擎一致性缺陷实例

1.1.2 研究目的与意义

为了解决 JavaScript 引擎普遍存在的一致性问题，由 ECMA（European Computer Manufacturers Association，欧洲计算机制造商协会）组织的第 39 号技术委员会公开创立了 Test-262 测试套件^[7]，专门用于检测 JavaScript 引擎的具体实现是否与 ES 标准一致。截止 2021 年 3 月，Test-262 套件已有 140 余名贡献者人工编写的超过 38000 条测试用例。

Test-262 套件的测试范围虽然很广，但由于用例都是人工编写的，所以仍然避免不了遗漏和错误。本文观察到，尽管现有的主流 JavaScript 引擎在新版本发布之前都会通过运行 Test-262 来检测是否存在一致性缺陷，但是引擎的一致性问题仍然长期存在。据统计，仅 2020 年就有 13 个关于 ChakraCore 引擎的一致性问题的缺陷报告被提出^[8]，这说明仅仅依靠 Test-262 来解决 JavaScript 引擎的一致性缺陷是非常困难的。

鉴于 JavaScript 引擎的一致性缺陷可能导致的严重后果，以及目前检测手段的缺乏，本文提出一种新式的针对 JavaScript 引擎的一致性缺陷检测方法，本方法采用深度学习自动地生成测试用例，并使用从 ES 标准文档中解析出的语义信息指导用例进行定向突变，随后基于新生成的用例巧妙地通过差分模糊测试机制来同时对多个 JavaScript 引擎进行一致性缺陷检测。此方法拥有较高的自动化水平，仅需要较少的人工参与，便能有效地对引擎的一致性缺陷进行检测。

本文基于此方法实现了一个一致性缺陷检测系统原型，使用其切实地检测出了主流引擎中存在的大量一致性缺陷，保障了引擎执行的正确性。无论是对于浏览器用户还是对于 Web 服务供应商都有很大帮助，同时也对推动整个 JavaScript 软件生态更好地发展具有积极的现实意义。

1.2 国内外研究现状

近年来，模糊测试技术（Fuzzing）凭借高效、自动化等特点被广泛应用于软件缺陷的检测。Mozilla 组织提出的 jsfunfuzz 工具^[9]能够基于预定义的 JavaScript 语法模板来生成测试用例，并对 SpiderMonkey 引擎进行了测试，自 2008 年以来它已累计发现超过 2800 个缺陷。然而，jsfunfuzz 的语法模板需要持续地进行手动更新，因此需要较强的专家知识，拥有较高的使用门槛，并且需要一定的人力成本。另外，它针对 SpiderMonkey 引擎做了大量的硬编码，从而欠缺了通用性，难以迁移到其他引擎上。

Christian Holler 等人提出的 LangFuzz^[10]是首个通过将种子代码库中的代码切片

后重组来生成新的测试用例的模糊测试工具。在预处理阶段，它预先收集了大量的 JavaScript 代码文件，将其解析成抽象语法树后切分成片段并构成片段池；在用例生成阶段，它随机选取种子用例的语法树中需要被替换的节点，随后从片段池中挑选同类型的新节点来进行替换，从而产生新的测试用例。而 Spandan Veggalam 等人提出的 IFuzzer^[11]则在 LangFuzz 的基础上加入了遗传算法，通过对优秀个体用例的筛选，使得 IFuzzer 能够生成更不常见的用例，从而提高触发引擎缺陷的可能性。这两种方法都是在语法树上进行替换操作，能够在一定程度上保证新生成用例的语法正确性。然而，二者都没有采取措施保证生成用例的语义正确率，从而导致生成的大部分用例都会在执行时发生异常而提前终止，使得用例得不到充分地利用。

Samuel Groß 提出的 Fuzzilli^[12]则是自定义了一套中间语言，在中间语言的层面上做随机突变从而产生新的语法正确的测试用例，并且在测试过程中以覆盖率为导向，不断保留能够提升覆盖率的用例，以此尽可能产生覆盖率更广的用例，这种方法被证明是有效的。然而，此方法需要极强的专家知识，以及充分的前期准备。不仅需要自定义一套中间语言，以及相应的随机突变策略，还需要在 JavaScript 引擎编译时对其进行插桩，以便于实时地获取到引擎的覆盖率信息，用以指导保留用例，同样降低了方法的泛用性。

HyungSeok Han 等人观察到了以往的测试工具在生成用例时的语义正确率较低这一现象，提出了 CodeAlchemist^[13]工具。它同样是使用代码片段拼接的方法来产生新的测试用例，并且它还内置了约束分析模块，能够在拼接的过程中，通过数据流分析和动态类型跟踪机制，给每个片段加上前置和后置约束，从而在片段拼接时为用户提供语义层面上的指导，以提升生成用例的语义正确率。然而，最新的研究 Favocado^[14]显示，这种方法所带来的语义正确率的提升效果非常有限——其生成用例中只有 28% 是语义正确的。

Suyoung Lee 等人则收集到了一批能够触发 JavaScript 引擎安全漏洞的用例，通过将触发过引擎漏洞的用例与各个引擎的回归测试用例的语法片段进行对比，他们发现新的缺陷通常由回归测试用例中已有的代码片段组成这一现象。据此，他们提出了 Montage^[15]工具，其使用神经语言模型在语法树层面上对收集到的回归测试用例进行建模，随后通过在语法树层面上对种子用例做突变的方式来生成新的测试用例。然而，Montage 的变异方式是随机的，完全由训练好的神经语言模型来决定。这种方式在用例生成过程中缺乏指导，从而难以触发某些特定的引擎缺陷。

Soyeon Park 等人观察到现有的基于生成和变异的 Fuzzer 由于巨大的输入空间或代码结构损坏等原因而不能有效地利用语料库的这一特点, 提出了 DIE 工具, 其采用一种基于关键路径保护的测试用例变异方法, 在对由漏洞证明用例和回归测试用例组成的高质量语料库进行变异时, 通过保留种子用例中与触发漏洞相关的重要变量和代码依赖关系, 替换其他无关的代码片段来实现对种子用例的变异, 从而发现引擎的安全漏洞^[16]。然而, 这种方法发现漏洞的能力极度地依赖初始语料库的质量, 又引入了如何收集到高质量语料库的问题。

上述工作都旨在对 JavaScript 引擎中的安全漏洞进行检测, 并没有专门针对于一致性缺陷的检测机制。然而, 安全漏洞只是引擎缺陷中的一部分, 保障引擎以及操作系统的安全性固然重要, 但也不能因此忽略了引擎中的功能性缺陷, 尤其是一致性缺陷。一致性缺陷会对 JavaScript 程序的正常功能和拓展性产生严重危害, 对其进行针对性的检测是很有必要的, 本文创新地设计了一种基于标准文档分析的用例突变算法, 能够针对性地对引擎的一致性缺陷进行检测。

1.3 本文研究内容

本文的研究内容包括以下几个方面:

(1) 基于标准文档分析的测试用例定向突变方法研究

本文通过对 ES 标准文档进行研究分析, 发现其对于 JavaScript 引擎的一致性缺陷检测具有重要指导作用, 从而创新地设计了一种基于标准文档分析的测试用例突变方法。该方法能够从标准文档中自动地解析出有效的语义信息, 来指导测试用例进行定向地突变, 突变后的用例能够精准地覆盖到引擎的更多分支, 从而能够触发更多的引擎一致性缺陷, 提高了检测效率。

(2) 基于差分模糊测试的 JavaScript 引擎一致性缺陷检测方法研究

一致性缺陷检测, 实际上也是一种软件测试方法, 本文对软件测试领域的常见技术 (尤其是差分测试与模糊测试) 进行了研究, 分析其优势与不足, 最终设计了一种基于差分模糊测试机制的自动化测试方法。该方法基于自动生成的测试用例, 能够持续地对 JavaScript 引擎进行差分模糊测试, 以检测引擎的一致性缺陷。同时, 本文对现有的测试用例生成方法进行了广泛地研究, 最终设计并实现了两种有效的用例产生方法, 除了上文提到的基于标准文档分析的用例定向突变生成方法以外, 还有一种基于神经语言模型的测试用例生成方法, 其能够自动地从 JavaScript 代码语料库学习编

码规范和语法语义知识,最终能够根据给定的代码上文续写下文,得到新的测试用例。

(3) 实现 ESfunfuzz 原型系统并在真实环境下对引擎的一致性缺陷进行检测

本文根据提出的方法设计并实现了名为 ESfunfuzz 的原型系统,并且使用该系统对真实环境下的四大主流 JavaScript 引擎进行了长期的缺陷检测实验,最终检测出了 22 个真实存在的引擎缺陷。之后通过对其中的 3 个典型的缺陷案例进行研究,探明了本文所提出的各个方法的具体生效机制,证明了本文方法的有效性。同时,本文还将 ESfunfuzz 与如今最先进的几种模糊测试工具进行了对比实验,进一步说明了本文方法的可靠性和实用性。

1.4 本文组织结构

本文共划分为五个章节来对本文提出的基于标准文档分析和差分模糊测试的引擎一致性缺陷检测方法进行论述,全文的组织结构安排如下。

第一章首先介绍了研究背景与意义,从 JavaScript 语言入手,介绍 JavaScript 引擎目前普遍存在的问题,即一致性缺陷,并通过实例说明其巨大的危害性。之后明确指出本文的主要工作就是对 JavaScript 引擎的一致性缺陷进行检测,说明本文的研究目的与意义。紧接着跟进 JavaScript 引擎模糊测试领域的国内外最新的研究进展,了解其工作方法,并分析其优势与不足。随后明确指出本文提出的基于标准文档分析的一致性缺陷检测方法的主要研究内容,并概括本文的组织结构。

第二章主要介绍本文的方法所用到的相关理论与技术,首先对 JavaScript 语言的相关背景做了介绍,还简要说明了本文方法设计的动机;随后对文本后续要用到的测试相关技术和语言模型相关知识进行了介绍。

第三章对本文的核心内容,即基于标准文档分析的测试用例突变方法进行了介绍,首先对现有的用例突变方法进行了研究和分析,说明其存在不足,随后说明了标准文档中的语义信息对用例突变过程的重要指导作用,之后介绍了本文设计的从标准中解析语义信息的具体算法,最后说明如何使用提取出的语义信息进行用例突变。

第四章对本文提出的基于差分模糊测试的 JavaScript 引擎一致性缺陷检测方法进行了一个详细的介绍。首先进行了方法概述,说明方法的整体流程,随后对生成模型构建、测试用例生成以及差分模糊测试等三个主要模块的基本思路和方法设计等内容进行了深入介绍。

第五章则是原型系统实现以及对应的实验部分。首先介绍了基于本文方法实现的

名为 ESfunfuzz 的原型系统，对其模块设计与实现进行了描述，主要包括模块设计、核心算法和系统页面设计等三个主要方面的内容。之后详细介绍了本文围绕该系统所开展的具体实验，主要包含语言模型的调优、与其他模糊测试工具的比较以及真实环境下对引擎的测试等三个方面的实验，进一步验证了本文的一致性缺陷检测方法的有效性。

最后是总结与展望部分，首先对本文的研究背景、研究内容以及研究的结果进行了总结，随后对本文目前存在的几点不足进行了分析，明确了之后的研究方向。

第二章 相关理论与技术

本文主要的研究内容是使用基于标准文档分析和差分模糊测试的方法对 JavaScript 引擎进行一致性缺陷检测，本章将对其中涉及到的相关理论和技术进行介绍。其中，JavaScript 语言背景部分将介绍 ECMAScript 语言、JavaScript 语言以及 JavaScript 引擎之间的关系；测试相关技术部分将介绍模糊测试技术和差分测试技术，以及将二者联合起来使用的差分模糊测试技术；语言模型部分将对语言模型的基础知识以及基本功能做简单介绍，并说明基于语言模型实现代码生成功能的过程。

2.1 JavaScript 语言背景

2.1.1 ECMAScript 语言及其规范

ECMAScript 是一种程序设计语言，它由 ECMA 国际组织通过 ECMAScript-262 语言规范进行标准化，旨在确保不同浏览器之间对 Web 页面的互操作性。ECMAScript-262 语言标准（以下简称“ES 标准”）是 ECMAScript 的语言规范，它以结构化的伪代码形式，分模块地详细介绍了 ECMAScript 语言所需要满足的所有功能及其具体实现步骤，图 3 是截取自 ES 标准的一部分实例，它规定了 JavaScript 引擎内部该如何实现 `String.prototype.startsWith` 方法。ES 标准自 1997 年发布第一个版本以来不断推陈出新，目前已发展到第 11 个版本（即 ES11）。值得注意的是，ECMAScript 语言仅仅是一种规范，其本身并没有进行相应地实现^[17]。

```
String.prototype.startsWith ( searchString [ , position ] )  
1. Let O be ? RequireObjectCoercible(this value).  
2. Let S be ? ToString(O).  
3. Let isRegExp be ? IsRegExp(searchString).  
4. If isRegExp is true, throw a TypeError exception.  
5. Let searchStr be ? ToString(searchString).  
6. Let pos be ? ToInteger(position).  
7. Assert: If position is undefined, then pos is 0.  
...
```

图 3 ES 标准对 `String.prototype.startsWith` 方法的规定（节选）

2.1.2 JavaScript 语言及其引擎

JavaScript 语言作为 ES 标准的一个具体实现,目前已成为世界上最流行的编程语言¹,尤其是作为 Web 页面的前端脚本语言。截止 2021 年 3 月,约有 97.1%的 Web 页面都使用 JavaScript 作为它们的前端脚本语言,如表 2 所示。

表 2 前端脚本语言使用率历史趋势统计^[18]

前端脚本语言	2021	2020	2019	2018	2017	2016	2015
JavaScript	97.1%	95.0%	95.1%	94.9%	94.4%	92.1%	88.1%
Flash	2.2%	2.8%	4.0%	5.5%	7.3%	9.5%	12.1%
None	2.9%	4.9%	4.9%	5.0%	5.5%	7.8%	11.7%
Silverlight	<0.1%	<0.1%	0.1%	0.1%	0.1%	0.1%	0.1%
Java	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	0.1%

随着 JavaScript 语言的日渐流行,用来运行 JavaScript 代码的解释器——JavaScript 引擎也受到了越来越多的重视。JavaScript 引擎最主要的用途是嵌入在 Web 浏览器内部,作为 JavaScript 代码的解释器来负责执行 JavaScript 代码,以实现 Web 页面元素的动态控制,目前所有现代浏览器都内置了高性能 JavaScript 引擎^[19]。

在种类繁多的 JavaScript 引擎中有四个最具代表性的,如表 3 所示,它们分别嵌入在四个主流的桌面浏览器中,为全球数十亿网络用户提供服务^[20]。

表 3 四大主流 JavaScript 引擎

引擎开发商	引擎名称	对应浏览器名称
Apple	JavaScriptCore ^[21]	Safari
MicroSoft	ChakraCore ^[22]	Edge
Google	V8 ^[23]	Chrome
Mozilla	SpiderMonkey ^[24]	Firefox

除了嵌入在浏览器内部,JavaScript 引擎也被应用到许多其他场景中,例如 V8 引擎就被用来作为 Node.js 的内核^[25],便于在服务器端运行 JavaScript 代码;Hermes 引擎专注于移动端,为基于 React Native 开发的安卓应用带来更快更好的用户体验^[26];另外,还有 JerryScript 等专为小内存设备使用的引擎^[27],以及 Rhino^[28]、Nashorn^[29]

¹ https://madnight.github.io/github/#/pull_requests/2020/4

和 GraalJS^[30]等嵌入在 Java 开发工具包（JDK，Java Development Kit）中的用于多语言融合编程的 JavaScript 引擎。

2.1.3 一致性缺陷的定义

JavaScript 语言作为 ECMAScript 的一种具体实现，其理应实现 ES 标准中的所有语法和功能特性。同理，JavaScript 引擎作为 JavaScript 代码的执行环境，也必须严格按照 ES 标准规定的操作步骤对方法和功能进行实现。然而，JavaScript 引擎常常由于开发人员的疏漏而产生一致性缺陷。现给出一致性缺陷的严格定义：给定一个 JavaScript 引擎的具体实现 J 与某一个版本的 ECMAScript-262 标准 E，一致性缺陷是指由于引擎开发者违反了 E 中的规定而在 J 中发生的非预期行为。

通过对部分引擎的一致性缺陷实例进行分析研究，本文发现引擎的一致性缺陷大都发生在对 JavaScript 常用方法的实现上，尤其是其中涉及到的强制类型转换步骤以及对边界值和特殊值的条件判断步骤，很多缺陷或是遗漏了强制类型转换步骤^{[31][32]}，或是在处理边界值时出现了问题^{[33][34]}。因此，假如能有效地对这些步骤进行重点测试，则能有效地对引擎的一致性缺陷进行检测。

本文随后对 ES 标准文档进行了分析，发现标准文档中存在着很多有效的语义信息，利用它们可以定向地对测试用例进行突变，以便于对上述步骤进行针对性的测试，能够更有效地检测引擎的一致性缺陷。方法的具体细节将在第三章进行陈述。

2.2 测试相关技术

2.2.1 模糊测试

自从 1990 年模糊测试（以下简称“Fuzzing”）思想诞生以来，它便成为了验证软件正确性和可靠性的使用最广泛的技术之一。Fuzzing 本质上是一种自动化测试技术，它通过构造大量的随机输入来查找软件中的安全漏洞^[35]。模糊测试方法主要包含用例生成、用例执行和监控以及人工分析三个主要步骤，其基本流程如图 4 所示。

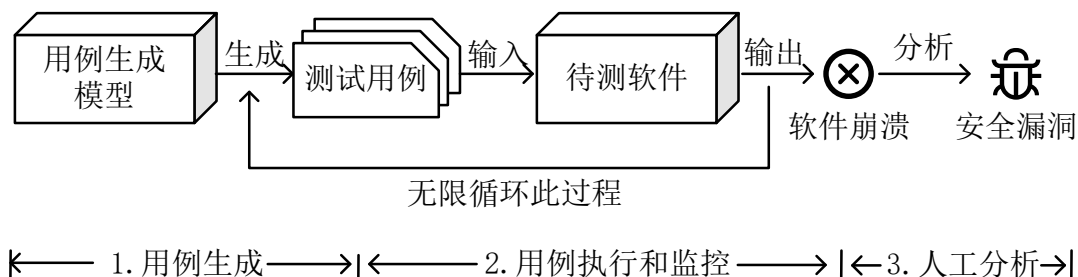


图 4 模糊测试过程概述

(1) 用例生成

所有模糊测试工具（以下简称“Fuzzer”）都需要一个用例生成模型（或一种生成方法）来持续地产生新的测试用例。如何生成新的有效测试用例是所有 Fuzzer 关注的核心问题，测试用例的质量将直接决定一个 Fuzzer 的有效性。当前主流的用例生成方法可以概括为生成式和突变式两类。

生成式方法通常是由人工根据经验和专家知识来编写语法规则或者生成模板，然后依据规则和模板来随机产生新的测试用例，比如 jsfunfuzz、Dharma^[36]以及 Favocado。而突变式方法往往需要先收集一批待测软件对应的代码文件作为种子用例，随后要么按某种规则在种子用例基础上进行修改，从而得到新的测试用例，比如 Fuzzilli 和 Montage；要么将种子用例切分为代码片段，然后通过将代码片段进行拼接的方式来生成新测试用例，比如 LangFuzz、IFuzzer 和 CodeAlchemist 等。常见的用例生成方法如图 5 所示。

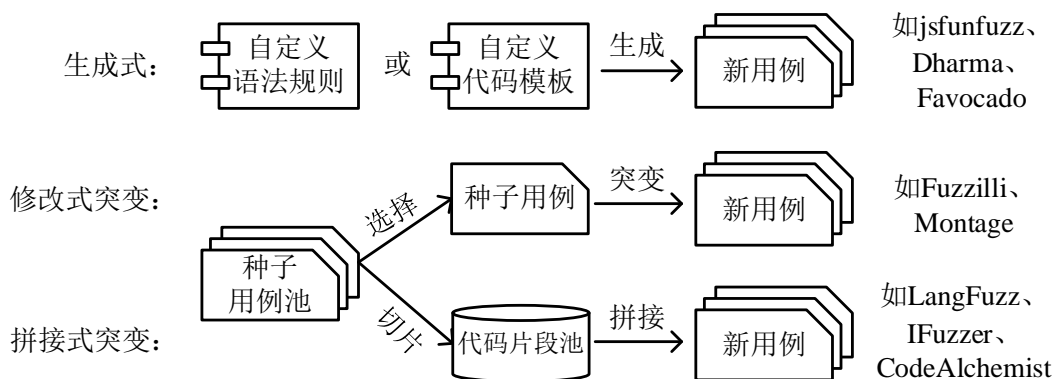


图 5 常见的用例生成方法

通常来说，生成式方法需要更多的专家知识，因为其无论是自定义语法规则还是自定义代码模板，都需要对目标语言和待测对象极为了解，才能定义出适当的规则或模板。而突变式方法由于用例的主体都来源于收集到的种子用例，所以只需要定义适当的突变规则即可。但是，突变式方法往往具有较大的随机性，难以触发某些特定缺陷。本文提出的测试方法将同时使用生成式和突变式方法来产生用例，具体细节将在第三、四章进行描述。

(2) 用例执行和监控

在得到新生成的测试用例之后，需要将其输入到待测软件（比如编译器）中执行，并对用例执行过程进行监控。当发现执行过程中待测软件表现出了异常的行为（比如软件崩溃）时，便将该用例标记为“可疑用例”并记录下来，便于后续对其进行更细

致的人工分析。

（3）人工分析

人工分析是模糊测试过程中必不可少的一个步骤，因为并非所有的可疑用例都是真正的软件缺陷，因为用例在执行时会受到多方面因素的影响，比如操作系统环境、其他程序的内存堆栈操作等，所以需要人工介入进行确认。人工分析一方面是为了确定该用例是否真的触发了软件缺陷，另一方面是当缺陷确认后还需要调查触发缺陷的可能原因，便于向开发商提交缺陷报告。

早期的 Fuzzing 通常指“黑盒”测试，即将待测对象看作一个黑盒，仅仅依靠随机输入和对应输出来对其进行测试。在之后的发展中，人们为了使黑盒测试更具引导性，从而提高 Fuzzing 的广度和深度，已经发展出了一系列白盒、灰盒的 Fuzzing 工具。白盒 Fuzzing 通常基于动态符号执行，通过分析待测对象的内部状态以及在待测对象执行时收集的信息来指导测试用例的生成^{[37][38]}。因此，白盒 Fuzzing 通常更加复杂，并伴随着很大的时间开销^[39]。而灰盒 Fuzzing 可以看作是轻量级的白盒测试，它通过在待测对象上执行轻量级的静态分析或者收集有关其执行的少量动态信息，如代码覆盖率等，用以对测试用例做出评价，便于利用进化算法引导用例向更好的方向生成^{[40][41]}。

模糊测试最大的优点在于其测试过程中仅需要少量的人工参与，便可以对待测软件进行持续的自动化测试。然而，模糊测试方法无法直接用于检测 JavaScript 引擎的一致性缺陷，因为一致性缺陷属于功能性缺陷，其执行过程并不会导致软件崩溃，因此只能通过对用例的实际输出与预期输出做比较才能发现问题。然而，模糊测试过程中可能会执行上百万条测试用例，测试人员无法预先推导出每一个用例的预期输出，自然也就无法进行输出结果的比较。本文通过引入差分测试机制来解决这一问题。

2.2.2 差分测试

差分测试是一种用来代替传统断言式测试的机制^[42]。在传统的断言式测试中，测试人员需要提前推导出测试用例的预期结果，然后再将用例输入到待测软件（如编译器）中执行以得到实际结果，最后比较预期结果与实际结果是否一致，以判断软件执行该用例是否正常，如图 6(a)所示；而差分测试则另辟蹊径，它将一个用例同时输入到多个同类型的待测软件中，获取到多个执行结果。随后，它并不需要人工推导出用例的预期结果，而是对得到的多组执行结果进行比较。当多组执行结果中存在“不一致行为”时，则认为该用例不通过，即该用例触发了该待测软件的缺陷。差分测试的

具体流程如图 6(b)所示。

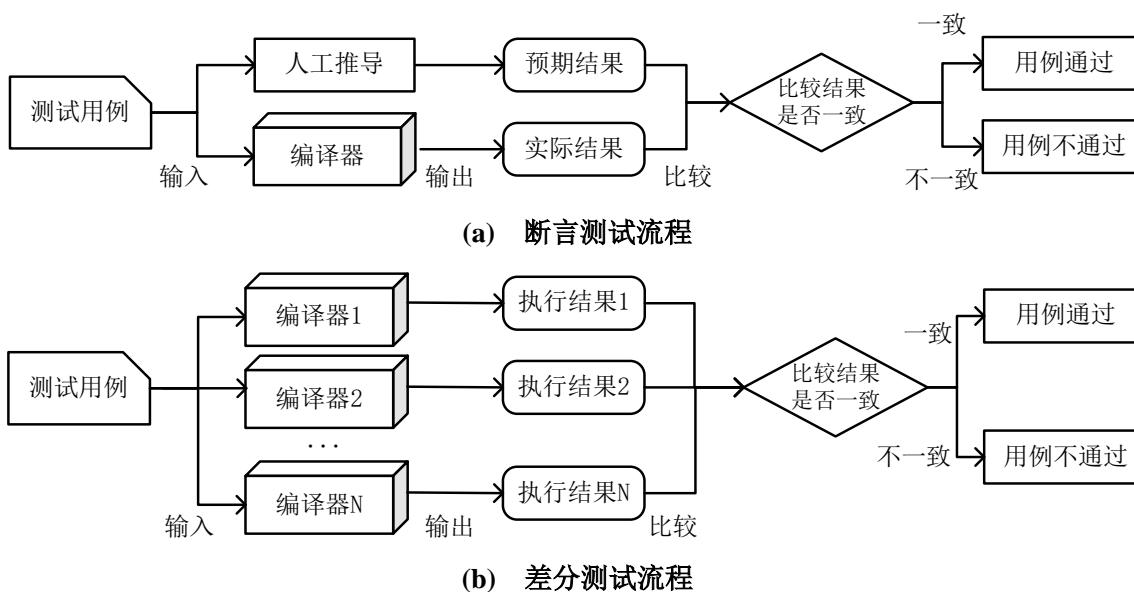


图 6 两种测试方法的流程比较

差分测试机制有效的前提是必须有多个同类型但不同开发商的待测软件，并且它们都遵循着同样的语言规范，即多个待测软件对于同一个用例应该输出同样的执行结果，只有在满足这样的前提下才能使用差分测试。在本文的工作中，尽管 JavaScript 引擎的特性和具体用途各异，但其实现上都必须严格遵守 ES 标准，其基本功能是一致的，即对于有效输入必须给予正确的输出，对于无效输入必须抛出对应的标准错误，这一特性为本文使用差分测试机制来对多个 JavaScript 引擎进行缺陷检测提供了理论依据。

2.2.3 差分模糊测试

章节 2.2.1 对模糊测试思想进行了全面的介绍，它通过构造大量的随机输入来对软件进行测试，但是由于无法预知模糊测试过程中产生的大量测试用例的预期输出，所以无法直接使用模糊测试方法对 JavaScript 引擎的一致性缺陷进行检测。章节 2.2.2 介绍的差分测试机制则巧妙地通过比较多个引擎的执行结果解决了这一问题。因此，本文采取了将模糊测试思想与差分测试机制相结合的测试方法，称为“差分模糊测试”，该方法测试的整体流程基于模糊测试思想，而在执行结果比较的部分又使用了差分测试机制，其整体流程如图 7 所示。

首先构建一个用例生成模型，它能源源不断地产生新的测试用例，随后依次将这些用例输入到多个 JavaScript 引擎中执行，并记录下多组执行结果。之后采用差分测试机制，分析比较各组结果判断其中是否存在不一致。当存在不一致时，便会将该用

例交由人工分析，以确定是否真的触发了引擎的一致性缺陷。无限循环从生成到执行再到分析的整个流程，从而对引擎进行长期的自动化测试。

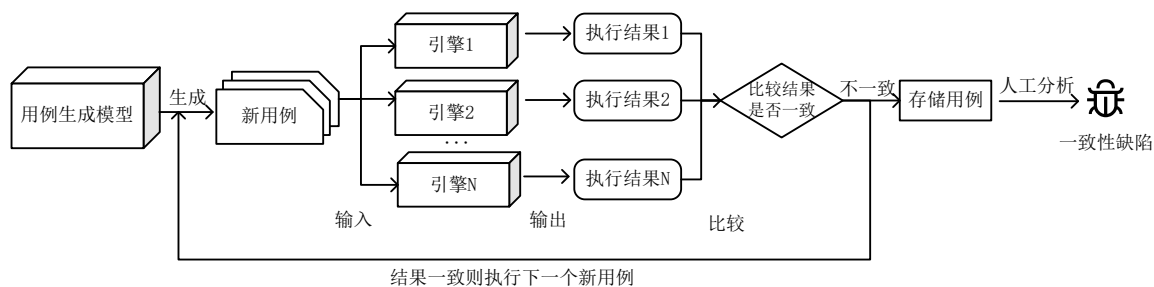


图7 差分模糊测试过程概述

差分模糊测试方法充分结合了模糊测试和差分测试的优点，规避了各自的缺点，使用该方法能够自动化地对 JavaScript 引擎的一致性缺陷进行检测。该方法的具体实现将在第四章进行详细介绍。

2.3 基于语言模型的代码生成

语言模型是一种文本建模的模型，使用它能够根据给定的语料库计算出一个语言序列出现的概率^[43]。例如，对于一个序列 S ，它能表示成若干个词序列的形式，如公式（2.1）所示。

$$S = w_1, w_2, \dots, w_n \quad (2.1)$$

其中， w_i 表示具体的词，那么根据语言模型便可以计算出序列 S 出现的概率，如公式（2.2）所示。

$$P(S) = P(w_1, w_2, \dots, w_n) \quad (2.2)$$

使用语言模型能够计算出每个词序列出现的概率，从而能够通过比较每种词序列的概率来实现根据给定的词序列预测最有可能的下一个词的功能。这种应用场景十分广泛，其中最典型便是输入法联想功能，即用户使用输入法进行文字输入时，输入法软件会根据用户已输入的字来预测下一个最可能的字词，以帮助用户提高输入效率。

2.3.1 统计语言模型

统计语言模型是语言模型最早的一种实现方式，顾名思义，即通过统计学的方法来实现对文本的建模，具体做法如公式（2.3）所示，根据链式法则将词序列的概率等价于多个条件概率相乘的形式，而这些条件概率根据极大似然估计理论能够从给定的语料库中通过统计方法得到^[44]。

$$\begin{aligned}
& P(w_1, w_2, \dots, w_n) \\
& = P(w_1)P(w_2|w_1) \dots P(w_n|w_1, \dots, w_{n-1})
\end{aligned} \tag{2.3}$$

然而, 这种直接统计的方式存在两个致命缺陷, 其一是计算空间过大, 即条件概率 $P(w_n|w_1, w_2, \dots, w_{n-1})$ 可能性太多, 导致难以估算; 其二是存在数据稀疏的问题, 即所求词序列中的某些词对组合在语料库中从未出现^[45], 此时根据极大似然估计求得的概率将会是 0, 从而影响相乘后的结果。

为了解决这个问题, 通常统计语言模型还会辅以马尔科夫假设进行计算, 马尔科夫假设是指假设当前词出现的概率只依赖于前 $n-1$ 个词, 通过这种方式来缩减参数空间, 便于计算, 缩减过程如公式(2.4)所示, 通过这种方式获得的模型称为“n-gram”模型。

$$\begin{aligned}
& P(w_i|w_1, w_2, \dots, w_{i-1}) \\
& = P(w_i|w_{i-n+1}, \dots, w_{i-1})
\end{aligned} \tag{2.4}$$

此处的 n 可以根据应用场景和设备的计算能力来灵活选取, 这种方式便是统计语言模型的基本实现方式。

2.3.2 神经语言模型

随着深度学习技术的不断发展, 人们开始探索利用神经网络来实现的语言模型^[46], 其中最成功的便是 Mikolov 等人提出的基于循环神经网络构建的语言模型, 称为循环神经网络语言模型^[47]。其核心思想是从大规模的语料库中进行无监督的深度学习, 使用循环神经网络模型来拟合语料库中的序列数据, 并通过计算实际输出与预期输出的损失函数利用反向传播方式来优化模型参数, 最终得到的模型便能根据给定的词序列来预测下一个词^[48]。由于循环神经网络模型的时间步形式非常地契合词序列, 此方法获得了空前的成功, 它能基于给定的整个词序列进行预测, 弥补了传统的 n-gram 语言模型仅仅考虑前 $n-1$ 个词的不足, 达到了更好的效果。

为了进一步提升模型对数据拟合效果, Sundermeyer 等人提出了使用长短期记忆网络替换传统的循环神经网络来构建语言模型的方法^[49], 其通过其特有的门控神经元, 能够选择性地对上下文进行加强记忆或者遗忘信息, 这种方式进一步提升了神经语言模型的效果。

2.3.3 基于语言模型的代码生成任务

上文提到, 使用语言模型可以实现根据给定的词序列预测下一个最可能的词的功能, 基于这一点便可以使用语言模型来自动地生成文本。生成过程如下: 根据所给定

的文本上文（假如没有上文，则指定一个占位符），预测出下一个最有可能的词，随后将预测出的词与文本上文拼接起来，作为新的上文继续预测下一个词。不断循环这个过程，直到生成了预设的结束符或者达到次数限制为止，如此一来便实现了文本的自动生成。文本自动生成任务目前已经比较成熟，常见的应用场景比如对话机器人、自动写诗、自动写对联等。

将代码视作一种特殊的文本，则理所当然地可以借助文本生成的框架来实现代码的自动生成，只需要将语料库数据替换成大量目标语言的代码文本，便可以实现自动生成代码的功能。这一点为本文使用神经语言模型来自动生成测试用例奠定了基础。

2.4 本章小结

本文的核心内容是使用基于标准文档分析和差分模糊测试的方法对 JavaScript 引擎进行一致性缺陷检测，而本章则对其相关的理论和技术进行了概述。首先介绍了 JavaScript 的语言背景，说明其重要作用及其存在的一致性缺陷；随后介绍了本文所使用到的测试技术，包括模糊测试、差分测试，以及将二者结合使用的差分模糊测试技术；最后对语言模型的基础知识以及基本功能做了简单介绍，并说明基于语言模型实现代码生成功能的过程，为第四章使用语言模型续写代码做铺垫。

第三章 基于标准文档分析的测试用例定向突变方法

测试用例突变是指在原始用例的基础上，通过对其进行某些修改，使得测试用例呈现出新的行为的一种用例产生方法。由于模糊测试过程中的测试用例通常是使用用例生成模型自动生成的，故用例具有极大的随机性。这种随机性会大大降低用例触发引擎缺陷的概率，从而影响整个缺陷检测的效率。因此，模糊测试方法通常都会通过某种策略对原始的测试用例进行某些定向或非定向的突变，从而提升其发现软件缺陷的能力。本章将对本文设计的基于标准文档分析的测试用例突变方法进行介绍，该方法能够有效地提升用例对引擎的覆盖率，扩大测试范围，从而触发更多的引擎缺陷。

3.1 测试用例突变方法概述

现有的测试用例突变方法主要有两种实现方式：随机修改和代码片段拼接。例如 AFL（American Fuzzy Lop，美国模糊测试工具）就使用了随机修改方法，它预先定义了 6 种随机突变策略，在测试过程中会自动地按照预设策略对现有的种子用例进行不同程度地突变，以得到新的用例^[50]，如图 8(a)所示。而 CodeAlchemist 工具则使用了代码片段拼接的方法，它将预先收集到的种子用例按语句的粒度切分成片段，随后将这些片段进行随机拼接以产生新的用例，如图 8(b)所示。

原始用例： 1 var count = 1; 2 function add(a, b) { 3 return a + b; 4 }	随机修改后的新用例1： 1 lar count = 1; 2 function add(a, b) { 3 return a + b; 4 }	随机修改后的新用例2： 1 var count = 1; 2 function add(a, b) { 3 return a - b ; 4 }
--	---	--

(a) AFL 的用例突变方法展示

原始用例： 1 var n = 1; 2 var arr = new Array(100); 3 for (let i = 0; i < n; i++) { 4 arr[i] = n; 5 arr[n] = i; 6 }	随机拼接后的新用例： 1 var s0 = new Array(100); 2 var s1 = 1; 3 for (let s2 = 0; s2 < s1; s2++) { 4 for (let s3 = 0; s3 < s2; s3++) { 5 s0[s3] = s2; 6 s0[s2] = s3; 7 } 8 }
---	--

(b) CodeAlchemist 的用例突变方法展示

图 8 两种典型模糊测试工具的用例突变方法展示

现有的随机突变方法的优势是实现起来比较简单且突变效率高,但其缺陷也十分明显:一是随机突变在本质上和使用生成模型产生随机的新用例并没有区别,仍然是依靠大量随机的测试用例进行“碰运气”式地检测,在效率上并没有质的提升;二是随机突变方法往往需要对同一个用例突变并测试非常多次,而这些随机突变后的用例实际上差别可能非常小,使得如此多次的测试实际上都是在运行同样的测试用例,这严重拖慢了测试效率。

为了更有效地对 JavaScript 引擎的一致性缺陷进行检测,本文提出一种新式的测试用例突变方法,即通过从 ES 标准中提取出的 JavaScript 语言的某些特定语义信息来定向地指导测试用例进行突变,这种“定向”能够让用例产生恰当的突变,使得突变后的新用例能够分别覆盖到引擎实现的不同分支,从而对引擎进行更加全面的测试,避免了盲目的随机突变对测试效率所带来的影响,极大地提升用例触发引擎一致性缺陷的可能性。

3.2 基于标准文档分析的用例定向突变方法概述

本文中的“标准文档”特指 ECMAScript-262 标准(简称“ES 标准”)的文档,它对 JavaScript 语言中所有语法、特性、内置对象及其属性和方法进行了详细的规定,并且要求所有 JavaScript 引擎内部必须按照规定一步一步地来实现。

本文观察到,ES 标准中有许多有效的语义信息能够给后续的模糊测试过程带来帮助。如图 9 所示,Number.prototype.toString 方法用来将一个 Number 类型的变量转换为 String 类型的变量。按照 ES 标准规定,它需要接收一个可选参数 radix,并且在整个处理流程中,根据 radix 输入内容的不同,流程会走向不同的分支:当 radix 没有输入,或者输入的是 undefined 时(第 2 行和第 3 行),那么就将 radixNumber 的值设定为 10;否则(即明确输入了 radix 时),会将输入的 radix 经过 ToInteger 方法处理后赋给 radixNumber(第 4 行)。随后,根据 radixNumber 的值不同,方法又会有三种不同的返回结果:当 radixNumber 值小于 2 或者大于 36 时,程序应该抛出一个 RangeError 异常(第 5 行);当 radixNumber 的值等于 10 时,程序应该返回对变量 x 进行 ToString 处理之后的结果(第 6 行);当 radixNumber 的值为其他情况时,又会返回其他结果(第 7 行)。由于 ES 标准对 Number.prototype.toString 方法做出了这样的规定,所以 JavaScript 引擎也必须按照此流程来进行该方法的具体实现。

通过上述分析可以得知,在执行包含 Number.prototype.toString 的单个用例时,

由于 `radix` 通常是单一值（假设该值取 10），则该用例只能执行到引擎的某个单一的分支，按照软件测试领域的覆盖率测试理论，该用例对引擎中该方法实现的“分支覆盖率”仅有 50%²。换句话说，该用例仅仅测试到了引擎实现的一半分支，而另一半分支根本没有测试到，可能会漏掉一些潜在的引擎缺陷。

```

Number.prototype.toString ( [ radix ] )
1. Let x be ? thisNumberValue(this value).
2. If radix is not present, let radixNumber be 10.
3. Else if radix is undefined, let radixNumber be 10.
4. Else, let radixNumber be ? ToInteger(radix).
5. If radixNumber < 2 or radixNumber > 36, throw a RangeError exception.
6. If radixNumber = 10, return ! ToString(x).
7. Return the String representation of this Number value using the radix specified by
radixNumber. Letters a-z are used for digits with values 10 through 35. The precise
algorithm is implementation-dependent, however the algorithm should be a generalization
of that specified in 7.1.12.1.

```

图 9 ES 标准对 `Number.prototype.toString` 方法的规定

那么，该如何提升用例对引擎方法的覆盖率呢。很明显，输入参数 `radix` 是导致后续分支变化的主要原因，并且在分支判断时 ES 标准中会有明确的边界值和特殊值（如图 9 中加粗部分）。因此，假如能在测试之前就获取到所有类似于这样的边界值和特殊值，就能在测试时向用例中对应的 JavaScript 方法或者函数传递这样的特定参数，便可以引导着引擎在执行用例时走向不同分支，提高用例对引擎的覆盖率，从而检测出引擎更多的潜在缺陷。

本文将 ES 标准中这些能够决定引擎分支走向的边界值和特殊值统称为“有效的语义信息”，通过这些语义信息来定向地对测试用例进行突变能够扩大用例的测试范围，从而发现更多的引擎一致性缺陷。如图 10(a)所示，原始用例中 `Number` 方法传入的参数是 10，因此按照 ES 标准，该用例执行时标准第 2、第 3 和第 5 步时，判断条件都为“假”，因此无法进入内部的分支，最终执行步骤 6，然后返回一个值并结束。而本文通过解析 ES 标准，发现步骤 2、3、5 中的条件判断要求 `radix` 的值必须包含不同的范围，比如步骤 2、3、5 分别需要 `radix` 不传参、传入 `undefined`、传入小于 2 或者大于 36 的值（比如 0）。因此，本文提出的定向突变算法会将该用例中 `Number` 方法的传入参数修改为多种不同的值，得到多个新用例，如图 10(b)所示。这样的多个用例使得对引擎方法的分支覆盖率达到 100%，即保证所有的分支都被测试到，这样就不会漏掉一些潜在缺陷，从而提升测试效果，并且这种“精准”的突变也克服了

² ES 标准描述中共有 4 个条件语句，即 8 个真假条件判断，而用例仅仅覆盖到了其中的 4 个，故覆盖率为 50%

传统的随机突变方法对测试效率产生的不利影响。

```
1 var tmp = Number(1).toString(10);  
2 print(tmp);
```

(a) 原始用例

```
用例1:  
1 var tmp = Number(1).toString();  
2 print(tmp);  
用例2:  
1 var tmp = Number(1).toString(undefined);  
2 print(tmp);  
用例3:  
1 var tmp = Number(1).toString(0);  
2 print(tmp);
```

(b) 定向突变产生的多个新用例

图 10 定向突变方法的实例展示

值得注意的是，图 10(b)中的用例 3 曾经触发过一个真实的引擎一致性缺陷³。由此可见，本文提出的定向突变方法能够有效地引导着用例覆盖到 JavaScript 引擎的更多分支，从而发现引擎更多的潜在缺陷。

3.3 标准文档解析方法

3.3.1 标准文档解析方法概述

章节 3.2 说明了 ES 标准中的边界值和特殊值等语义信息在指导用例进行突变上的重要作用，然而，如何准确地解析出这些隐含的语义信息又成了新的难题。

如何获取到 ES 标准的文本内容是首先需要考虑的问题。本文考虑到，由于 ES 标准本身是以 HTML（Hyper Text Markup Language，超文本标记语言）代码的形式呈现在 Web 页面中，而 HTML 页面必然是十分结构化的。因此，通过对 ES 标准页面的 HTML 结构进行观察，本文按其结构设立了特定的解析规则，便可以从解析出 ES 标准的完整文本。

获得 ES 文本之后便需要从中准确地提取出有效的语义信息，主要是与传入参数有关的边界值和特殊值。由于 ES 标准都是以自然语言文本的形式展现的，直接对其进行解析的难度较大，尤其是标准中可能包含的赋值操作，使得传入参数会与其他变量产生关联，如何在自然语言中分析这样的关联也是一个难点。比如图 9 所示的标

³ <https://github.com/jerryscript-project/jerryscript/issues/3229>

准的第 5 步，将传入参数 `radix` 经过 `ToInteger` 方法处理后赋值给了 `radixNumber`，使得 `radixNumber` 与 `radix` 产生了关联，因此，第 6 步和第 7 步中的 `radixNumber` 的特殊值实际上就可以看作是 `radix` 的特殊值，也需要被提取出来。想要在自然语言中捕捉这样的关系无疑是非常困难的。

因此，本文的解决方案是首先将自然语言形式的 ES 标准转化为对应的 JavaScript 代码。有了代码便可以将其通过现有的语法解析工具得到对应的抽象语法树，抽象语法树是将 JavaScript 代码经过语法和语义解析后得到的结构化表示，在语法树层面上可以方便地对代码的语义信息进行解析，从而提取出所有与传入参数有关的边界值和特殊值，最后根据这些边界值和特殊值的范围来确定最终的候选参数，以供后续的变异阶段使用。其基本流程如图 11 所示。

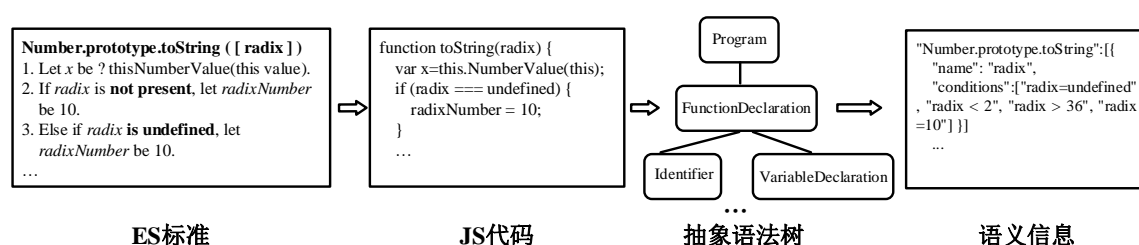


图 11 ES 标准解析方法概述

具体的标准到代码的转化算法以及语义信息的提取算法将在后续章节进行详细介绍。通过这种方式，本文能够获取到 ES 标准中所有的与 JavaScript 方法和对象有关的语义信息，从而在模糊测试阶段指导用例进行定向地变异，以便于更有效地触发引擎的一致性缺陷。

3.3.2 标准转化方法

将自然语言形式的 ES 标准转化为对应的 JavaScript 代码并不是一件容易的事情。本文观察到，ES 标准的形式虽然是自然语言描述，但其用词与表达形式均是非常规范的，通过寻找其文本描述的规律，便可以编写预定义规则将特定的规范转化为相应的 JavaScript 代码。

因此，本文的实现思路是首先经过人工对标准进行全面地观察，总结出标准中主要出现的自然语言描述语句的形式；随后根据总结出的语句形式，为其编写出相应的符合 JavaScript 语法的代码，并利用正则表达式从语句中提取关键的变量信息填入代码中，以便于将二者联系起来；最后使用这些表达式来依次对 ES 标准的每个方法中的每一条语句进行匹配，并根据匹配的结果将其转化为 JavaScript 形式的代码即可，假如某条语句没有对应的规则，则丢弃掉该语句。这样最终便能得到标准对应的

JavaScript 代码，需要注意的是，转化后的 JavaScript 代码只需要保证语法正确即可，而不需要语义上的正确性，因为该代码仅仅用来解析，而不需要执行。

为了简化正则表达式的编写，本文首先定义了如表 4 所示的 8 种正则关键字。随后，本文基于这 8 种关键字人工设计了超过 150 个正则表达式规则，通过它们来将 ES 标准转化为对应的代码，部分转化规则实例如表 5 所示。

表 4 本文定义的 8 种正则关键字

关键字名称	正则表达式	匹配内容描述
Var	<code>([a-zA-Z0-9_]+?)</code>	变量名
Func	<code>([?!a-zA-Z0-9\$_()^\[\]\.,"@\\-+<>"]+?)</code>	函数名（包含特殊符号）
Num	<code>([0-9+\\-]+?)</code>	数字（包含正负号）
Mark	<code>([!]?*)</code>	特殊标记
End	<code>[,]*?</code>	语句结束符
Op	<code>([>=<<=<=>])</code>	操作符
Con	<code>(false null undefined true NaN)</code>	特殊值
All	<code>([\\s\\S]?)</code>	极端情况下匹配所有字符

表 5 人工定义的部分转化规则

ES 标准描述	正则表达式	转换后的代码
Let x be 1.	<code>^Let \${Var} be \${Num}\${End}\$</code>	<code>var x = 1;\n</code>
Let R be the empty String.	<code>^Let \${Var} be the empty String\${End}\$</code>	<code>var R = ""; \n</code>
Let s be the length of S.	<code>^Let \${Var} be the length of \${Var}\${End}\$</code>	<code>var s = S.length;\n</code>
If Type(O) is number,	<code>^If Type\\(\${Var}\\) is number\${End}\$</code>	<code>if (typeof O === "number")\n</code>
If radix is not present,	<code>^If \${Func} is (not present absent)\${End}\$</code>	<code>if (radix === undefined)\n</code>
Repeat, while done is false,	<code>^Repeat, while \${Func} is \${Con}\${End}\$</code>	<code>while (done=== false)\n</code>

通过表 5 所示的正则转化规则，本文实现了从自然语言描述到 JavaScript 代码的转化。图 12 是转化成功的一个实例，图 12(a)是 ES 标准对于 `Number.prototype.toString` 方法的规定，共包含 7 条执行步骤。通过本文的正则转化方法，成功地将前 6 条转化为了对应代码，第 7 条标准过于复杂所以被跳过（这并不影响最终的解析结果），最终转化得到的代码如图 12(b)所示。

需要注意的是，ES 标准是按 JavaScript 语言的方法（包括函数、构造函数等）为单位进行组织的，因此本文也将按方法的粒度对其进行转化，即一个方法描述转化为一段 JavaScript 代码。

Number.prototype.toString ([radix])

1. Let *x* be ? thisNumberValue(this value).
2. If *radix* is **not present**, let *radixNumber* be 10.
3. Else if *radix* is **undefined**, let *radixNumber* be 10.
4. Else, let *radixNumber* be ? ToInteger(*radix*).
5. If *radixNumber* < 2 or *radixNumber* > 36, throw a RangeError exception.
6. If *radixNumber* = 10, return ! ToString(*x*).
7. Return the String representation of this Number value using the radix specified by *radixNumber*. Letters a-z are used for digits with values 10 through 35. The precise algorithm is implementation-dependent, however the algorithm should be a generalization of that specified in 7.1.12.1.

(a) ES 标准对 Number.prototype.toString 方法的规定

```
function NumberprototypetoString(radix) {
    var x = this.NumberValue(this);           // 对应第1条标准
    if (radix === undefined) {                 // 对应第2、3条标准
        radixNumber = 10;
    } else {
        radixNumber = ToInteger(radix);        // 对应第4条标准
    }
    if (radixNumber < 2 || radixNumber > 36) { // 对应第5条标准
        throw new RangeError();
    }
    if (radixNumber === 10) {                  // 对应第6条标准
        return ToString(x);
    }
    ...                                         // 第7条无法完成转化，略过
}
```

(b) 标准转化后的 JavaScript 代码**图 12 ES 标准转化为代码的实例****3.3.3 标准转化方法的效果评估**

为了进一步验证转化方法的效果，本文设立了充分度和准确度两个指标来对其进行评估。其中，充分度表示对 ES 标准进行转化的充分程度，即有多少句子能够成功地被转化为代码；而准确度则用于衡量转化后的结果是否正确。

以 ES10 版本的标准文档为例，本文从中收集到了 374 个有效的方法描述，共计 2813 条自然语言句子，按照本文的方法，需要将它们全部使用预定义的正则规则转化对应的 JavaScript 代码。据统计，在实际的转化过程中共有 2497 条成功进行了转化，剩余 316 条因没有匹配到对应的正则规则而被丢弃，整体的转化充分度达到了 88.7%。为了探明转化失败的原因，本文对转换失败的句子进行了分析，发现它们都是比较复杂的自然语言描述，预定义的正则规则难以对其进行匹配。如图 13 所示，这些句子不仅非常复杂，而且其中包含的语义信息非常有限，没有包含明显的边界值和特殊值，故不必对其进行进一步处理。

- If O does not have all of the internal slots of a String Iterator Instance (21.1.5.3), throw a `TypeError` exception.
- Set m to the string-concatenation of the first $e + 1$ code units of m , the code unit `0x002E` (FULL STOP), and the remaining $p - (e + 1)$ code units of m .
- Set c to the code unit whose value is the integer represented by the four hexadecimal digits at indices $k + 2, k + 3, k + 4$, and $k + 5$ within $string$.
- Return the String value containing *resultLength* consecutive code units from S beginning with the code unit at index *intStart*.

图 13 转化失败的句子的抽样展示

随后进行准确度评估实验，具体方法是从转化成功的 2497 条句子中随机抽样 3 次，每次抽取 100 条结果，得到三组样本，随后人工对这三组结果进行评估，衡量转化是否准确。评估的结果是三组样本中分别有 95 条、92 条和 94 条被认为是准确的，因此，求平均值后得到最终的转化准确度为 93.7%，其意味着绝大部分的转化都是准确的。同样地，本文对转化不准确的句子也进行了抽样检查，图 14 展示了三组转化不准确的实例，每组上半的“ES”部分是 ES 标准文档中的原始句子，而下半的“JS”部分则是将原始句子转化后得到的代码。观察可知，转化不准确的原因几乎都是原始句子过于生僻，导致没有能匹配上的正则规则，从而被某些通用正则转化为不正确的代码。所幸，此类生僻句子占比很少，并且几乎不包含任何有效的语义信息，故不需要对其进行特殊处理。

ES: Let *weekday* be the Name of the entry in Table 49 with the Number `WeekDay(tv)`;
JS: var weekday = the Name of the entry in Table 49 with the Number `WeekDay(tv)`;

 ES: Let m be the String value consisting of $f + 1$ occurrences of the code unit `0x0030`;
JS: var m = the String value consisting of $f + 1$ occurrences of the code unit `0x0030`;

 ES: Let a be the first code unit of m , and let b be the remaining f code units of m ;
JS: Let a = the first code unit of m, and let b be the remaining f code units of m;

图 14 转化不准确的句子的抽样展示

除了以 ES 标准中的自然语言句子作为评估粒度以外，本文还按 ES 标准中的单个方法描述为基本单位进行了评估。上文提到，ES10 标准文档中共有 374 个有效的方法描述，据统计，使用本文方法对这些方法描述进行解析，共有 349 个能够被解析成语法正确的代码，并且其中有 153 个能解析出明确的语义信息。这意味着使用本文的突变方法来生成新用例时，能够提高将近一半的触发引擎缺陷的可能性。另外需要注意的是，本文在对剩余的一半进行抽样调查时发现，未能解析出语义信息的原因并非本文的方法存在缺陷，而是其本身并没有有效的信息值得被抽取。

3.3.4 语义信息提取方法

将 ES 标准描述转化为对应的 JavaScript 代码之后，便需要考虑如何从中提取出有效的语义信息。本文提出了一种基于抽象语法树的语义信息提取算法，通过将 JavaScript 代码解析成抽象语法树之后，借助语法树特有的节点类型来准确地定位到其中的条件语句，并提取出其中与“条件”相关的边界值和特殊值，然后根据条件的范围确定参数有效的候选值，最终将这些信息以 JSON（JavaScript Object Notation，JavaScript 对象简谱）文件的格式存储，该文件将作为“语义数据库”，便于在后续的测试阶段来指导用例进行定向变异。

本文首先使用现有的 Esprima 工具⁴来完成从 JavaScript 代码到抽象语法树之间的转换任务，随后按照预先写好的节点筛选算法遍历语法树中的所有节点，以获取到所有与传入参数相关的边界值和特殊值的语句。同时，为了更好地捕捉如章节 3.2 中如同 `radix` 与 `radixNumber` 这种变量之间的赋值关系，以获取到更多隐含的语义信息，还需要特别建立一个映射字典来保存变量之间可能的赋值关系。最后，在提取出与传入参数相关的边界值和特殊值之后，将根据其范围来确定最终的候选参数。例如，从 `Number.prototype.toString` 方法中提取出的有效的语义信息共四条：`radix=undefined`、`radix<2`、`radix>36`、`radix=10`。其中，除了第 1 和第 4 条中的 `undefined` 和 10 这两个确定的值以外，还有小于 2 与大于 36 这两个范围。因此，还将额外地从小于 2、大于 36 以及 2 到 36 之间这三个范围内分别进行随机取值，以确定更多的候选参数。

最终将所有语义相关信息以 JSON 文件的形式保存。图 15 是该文件的一部分实例，其中 `name` 字段表示传入参数的名称，`type` 字段表示对该参数的预估数据类型，`conditions` 字段表示从该方法中提取出的语义相关的条件语句，`scopes` 字段表示从条件语句中分析出的参数的候选范围，`values` 字段表示最终确定的对于 `radix` 参数的所有候选值，`values` 字段将直接用于后续的用例变异步骤以提供有效地指导。

```
1 "Number.prototype.toString": [{
2   "name": "radix",
3   "type": "integer",
4   "conditions": ["radix === undefined", "radix === undefined", "radix < 2",
5     "radix > 36", "radix === 10"]
6   "scopes": [2, 36],
7   "values": [undefined, 10, 1, 37, 5]
8 }]
```

图 15 从 `Number.prototype.toString` 标准中提取出的语义信息

⁴ <https://esprima.org/>

3.3.5 参数类型推断及非预期类型变异

除了章节 3.3.4 介绍的通过边界值和特殊值来确定参数的候选值以外，本文还根据 JavaScript 语言和 ES 标准的特性额外设计了一种类型推断和非预期类型变异方法，来提高用例触发引擎缺陷的可能性。

由于 JavaScript 是一种弱类型的语言，所以其不能在变量声明时明确指定数据类型，只能等到引擎执行它时才来对其数据类型进行检查和判断^[51]。因此，在 ES 标准中为了保险起见，通常都会有明确的强制类型转换步骤，也就是将未知类型的传入参数强制转换为预期类型变量的步骤。如图 16 所示，ES 标准规定在 `Number.prototype.toString` 方法的第 4 步需要对传入的 `radix` 参数进行了 `ToInteger` 的强制类型转换操作，这从侧面说明了该方法预期的 `radix` 参数类型是 `Integer`。

```
Number.prototype.toString ( [ radix ] )
...
4. Else, let radixNumber be ? ToInteger(radix).
...
```

图 16 ES 标准要求强制类型转换的实例

沿着这一思路，本文实现了一种新的类型推断算法，即通过对 ES 标准中与传入参数相关的强制类型转换步骤进行定位和分析，便能够确定该方法预期的传入参数类型。这种算法与具体的代码无关，能够从根本上准确地推断出传入参数的类型。

获取到传入参数的预期类型之后，便能够指导着用例进行非预期类型变异。在本文前期的研究工作中发现，许多引擎缺陷都是由于引擎在实现上遗漏了强制类型转换步骤导致的^{[31][32]}。因此，本文特意地将用例中方法的传入参数突变为不符合预期类型的值，以检测引擎是否正确实现了强制类型转换步骤。

为了实现这一功能，本文为 JavaScript 语言的七种基本数据类型（即 `Number`、`Undefined`、`Null`、`String`、`Object`、`Boolean`、`Symbol`）分别编写了对应的生成器，每种生成器都能产生该类型的随机数据。当推断出变量预期的数据类型是某一类时，便从其他六种类型的生成器中随机选择一个，用来产生一个该类型的数据，并将其添加到语义数据库中的候选值列表中，供后续指导用例变异。

3.4 语义信息指导的用例定向突变算法

使用本章所描述的语义信息提取方法将整个 ES 标准文件处理一遍之后，便得到了一个总的 JSON 文件（称为“语义数据库”），其中保存着从标准中抽取出的所有

有效的语义信息，这些信息将对用例突变过程提供有效指导。在模糊测试过程中，当获取到一个种子用例之后，便开始对该用例进行有指导地定向突变，从而产生若干新用例，具体过程如算法 1 所示。

算法 1 语义信息指导的用例突变算法

Input:*prog*: A JS test case program*ecma*: A ECMAScript-262 document**Output:** T_{new} : A list of mutated test cases1: $Sdb \leftarrow \text{extractSemanticInfo}(ecma)$ 2: Let T_{new} be a list3: $ast \leftarrow \text{parse}(prog)$ 4: **for** $node \in ast$ **do**5: **if** $\text{isFunction}(node)$ **then**6: $funcName \leftarrow \text{getFuncName}(node)$ 7: **if** $Sdb.\text{containsName}(funcName)$ **then**8: $Sinfo \leftarrow \text{getSinfo}(Sdb, funcName)$ 9: $t_{new} \leftarrow \text{mutate}(prog, node, Sinfo)$ 10: $T_{new}.\text{append}(t_{new})$ 11: **end if**12: **end if**13: **end for**14: **return** T_{new}

算法需要两个输入 *prog* 和 *ecma*，前者表示一个 JS 测试用例，也就是本次突变所使用的种子用例；后者表示一个 ECMAScript-262 标准文档。算法的输出为 T_{new} ，表示突变后产生的新用例的列表。

第 1 行：通过 `extractSemanticInfo` 函数对 *ecma* 变量代表的标准文档进行解析，并将得到的语义信息存入语义数据库 *Sdb*；

第 2-3 行：将 T_{new} 初始化为空列表；将种子用例 *prog* 解析为语法树 *ast*；

第 4-5 行：遍历 *ast* 中的每个节点 *node*，并调用 `isFunction` 函数判断其是否为一个方法节点，假如是，则执行第 6-10 行；

第 6-7 行：调用 `getFuncName` 函数获取到 *node* 节点调用的方法名 *funcName*；并判断其是否在语义数据库 *Sdb* 中；假如在，则执行 8-10 行；

第 8-9 行：调用 `getSinfo` 函数，从语义数据库 *Sdb* 中查找到方法名 *funcName* 对应的语义信息 *Sinfo*，并将其与用例 *prog*、当前节点 *node* 一起传入 `mutate` 函数，`mutate`

函数将根据 Sinfo 信息将用例 prog 中的 node 节点的参数值进行定向突变，突变的新用例记做 t_{new} ；

第 10 行：将新生成的用例 t_{new} 添加到总和 T_{new} 中；

第 11-14 行：待所有循环和判断结束后，返回最终的结果列表 T_{new} ，它存储着本次突变产生的所有新用例。

复杂度分析：此算法分为两个阶段，第 1 行为一个独立阶段，主要实现语义信息的提取；其余为第二阶段，主要实现种子用例的定向突变。在第一阶段中，假设 ES 标准文档中文本段的个数为 n ，由于整个过程是对这些文本段进行线性地解析，且不需要额外的空间，故其时间和空间复杂度均为 $O(n)$ ；在第二阶段中，假设种子用例解析后的语法节点个数为 m ，整个过程需要遍历每个语法节点，且不存在多重循环操作，故其时间和空间复杂度为 $O(m)$ 。

综上所述，算法 1 的整体时间复杂度为 $O(m + n)$ ，并且由于文本段个数 n 远远大于种子用例节点 m ，其可以简化为 $O(n)$ ， $O(n)$ 复杂度表示该算法的时空消耗随着问题规模的增加而呈线性增长，属于较为理想的情况。目前，其他以突变为主的模糊测试工具，比如 AFL、CodeAlchemist 和 Montage，其时空复杂度均为 $O(n)$ ，与本文的算法在时空消耗上并无明显区别。

利用算法 1，一个种子用例便可以突变为多个参数值不同的新用例，并且新用例将准确地覆盖到引擎的不同分支，以全面地对引擎进行测试，提高用例触发缺陷的可能性。值得注意的是，当一个方法中有多个参数，且每个参数有多个候选值时，将会采用笛卡尔乘积的方式将多个参数的值进行组合，从而产生更多的新用例。一个用例突变的实例将在章节 4.3.2 展示；一个由定向突变方法触发的引擎缺陷实例将在章节 5.6.2 进行分析，在该实例中，种子用例本身并无法触发任何引擎缺陷，而定向突变后的新用例则精准地触发了由微软公司开发的 ChakraCore 引擎的一个缺陷，此例很好地说明了定向突变算法的有效性。

3.5 本章小结

本章主要对本文的核心内容，即基于标准文档分析的测试用例突变方法进行了详细的说明。首先介绍了现有的测试用例突变方法及其固有缺陷，即缺乏指导性的随机突变并没有为用例突变带来质的提升，反而可能会因为大量无用的突变而降低测试效率。之后结合实例，说明了 ES 标准在指导用例进行突变上的重要作用，并通过具体

用例展示了将用例进行定向突变后对引擎覆盖率所带来的巨大提升，说明了定向突变的有效性，即能够覆盖引擎的更多分支，从而触发更多的引擎缺陷。随后对本文设计
的标准解析方法进行了详细地介绍，即先提取标准文本，再将其按照正则规则转化为
JavaScript 代码，然后将代码转化为抽象语法树并在语法树层面上完成节点的解析，
以获取到标准中有效的语义信息，最后再根据语义信息确定最终的候选值，从而指导
用例进行突变。本章最后则对用例突变的具体实现方法进行了介绍。本文提出的基于
标准文档分析的测试用例突变方法能够克服传统突变方法的缺陷，定向地指导用例进
行突变，突变后的用例能够覆盖到引擎的更多分支，从而触发更多的引擎缺陷。

第四章 基于差分模糊测试的一致性缺陷检测方法

第三章对本文提出的基于标准文档分析的用例突变方法进行了详细的介绍,通过该方法能够对用例进行定向地突变以提升用例触发引擎一致性缺陷的可能性。本章将系统性地介绍本文所使用基于差分模糊测试的引擎缺陷检测方法的整个流程,主要包含生成模型构建、测试用例生成以及差分模糊测试等步骤,使用本方法能够有效地对JavaScript引擎的一致性缺陷进行检测。

4.1 方法概述

本文提出的检测方法整体基于差分模糊测试的思想,通过生成大量测试用例输入到JavaScript引擎中执行,监测引擎的执行结果并对其进行差分分析,以得到触发引擎不一致行为的可疑用例,最后通过对其进行人工分析来确定是否触发了引擎缺陷。并且为了提升用例触发引擎缺陷的可能性,使用了基于神经语言模型的用例续写以及基于标准文档分析的用例突变等两种用例生成方法。

本方法整体流程如图 17 所示, 主要包含语法标准解析、生成模型构建、以及差分模糊测试三个主要部分。

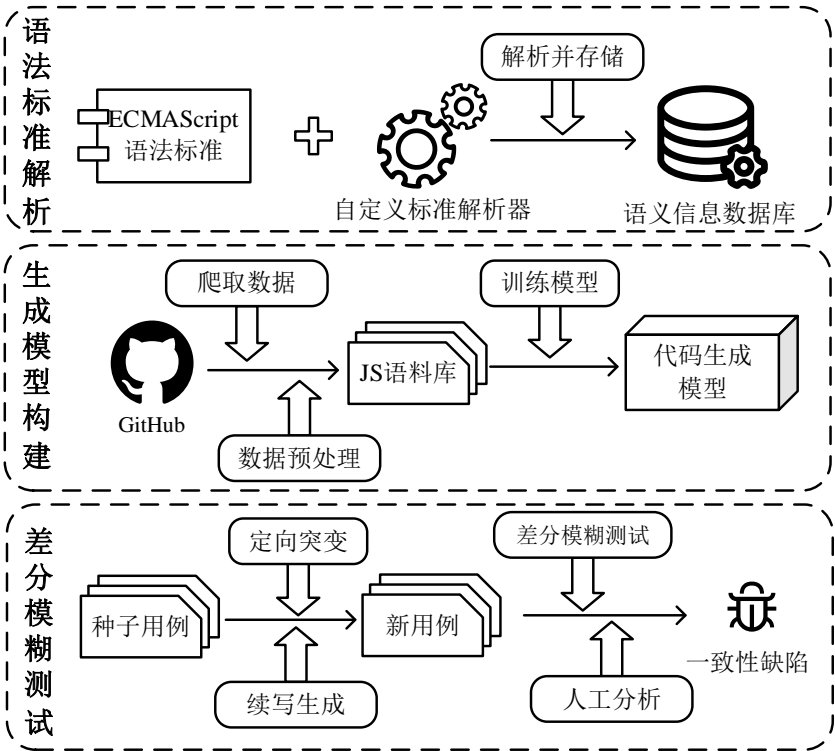


图 17 JavaScript 引擎一致性缺陷检测方法概述

（1）语法标准解析

为了提升测试用例触发引擎一致性缺陷的可能性，本文提出一种新的测试用例突变方法，即通过从 ES 标准中解析出的边界值和特殊值等有效的语义信息来指导用例进行定向突变。因此，在执行模糊测试之前必须先使用自定义的标准解析方法对 ES 标准进行解析，得到语义信息数据库，它将来在用例突变阶段指导用例进行突变。语法标准解析过程已在第三章进行了详细的描述，此处不再赘述。

（2）生成模型构建

任何模糊测试方法都需要一个能够不断产生测试用例的代码生成模型作为基础，以便于支持后续整个测试过程，本文使用基于深度学习的神经语言模型来实现测试用例的自动生成。此方法首先需要获取到大量的 JavaScript 文件，并将其经过预处理后构成一个语料库，之后基于该语料库训练一个神经语言模型，该模型将会在用例生成阶段来实现根据代码上文自动续写下文的功能。

（3）差分模糊测试

差分模糊测试部分是整个方法的核心内容，语法标准解析与生成模型构建两个部分都是为它服务的，模糊测试过程主要包括种子用例收集、新用例生成、差分模糊测试以及人工分析等四个主要步骤。首先需要构建一个由若干现有用例组成的种子用例池（简称“种子池”），之后开始生成新用例。生成时，每次从种子池中随机选择一个用例作为种子用例，随后基于该用例一方面使用训练好的神经语言模型对其进行续写，另一方面使用从 ES 标准中解析出的语义信息来对其进行定向突变，通过这两种方式共同产生新用例。之后将新生成的用例依次输入到多个 JavaScript 引擎中进行差分模糊测试。在测试过程中会得到若干触发引擎不一致行为的可疑用例，最后对其进行人工分析，以确定是否真的是引擎缺陷，并且向引擎开发商提交缺陷报告。

4.2 生成模型构建

章节 2.3 中介绍了有关神经语言模型以及代码生成任务相关的内容，根据代码上文续写下文是一种典型的代码生成任务，因此，本文同样选择神经语言模型来完成根据代码上文续写下文的用例生成任务。构建代码生成模型的主要流程如图 18 所示，主要包含语料库收集、数据预处理以及模型构建和训练等三个步骤。

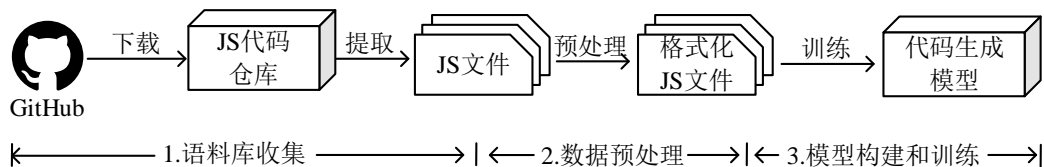


图 18 代码生成模型的构建过程

4.2.1 语料库收集

想要构建一个有效的神经语言模型，首先需要建立一个高质量的 JavaScript 代码语料库。GitHub 是世界上最大的开源代码托管平台，其上托管了大量优秀的开源项目，能够提供满足本文要求的 JavaScript 代码文本。并且 GitHub 上的每个项目都有一个“Star”属性，表示他人对该项目的认可程度，Star 的数目在一定程度上能够代表项目的代码质量，通过该属性能够轻易地筛选出质量更高的仓库，从而获得更优秀的数据。

综上所述，本文从 GitHub 平台上以 JavaScript 为关键词、以 Star 数目为排序依据⁵，下载了 JavaScript 分类下的前 4000 个代码仓库的源代码，作为最初始的语料库。

尽管都是 JavaScript 分类下的仓库，但其中可能仍然包含少量其他语言的代码文件。因此，本文按文件类型区分，从这 4000 个仓库中提取出了所有的 JavaScript 类型的代码文件，总共获得了 499501 个 JavaScript 文件，约 6GB 的代码文本⁶，这些数据初步构成了本文的 JavaScript 代码语料库，随后将会对其进行预处理。

4.2.2 数据预处理

为了使得语料库更加标准和统一，便于模型更好地进行建模学习，本文对每个收集到的 JavaScript 文件都进行了如下预处理。

首先，本文依据人工经验设立了过滤黑名单，黑名单中都是 JavaScript 语句及某些引擎所特有的语法和函数，这一步会过滤掉包含其中任意关键词的 JavaScript 文件，这一步的目的是为了减少后续的差分模糊测试过程中可能产生的误报。例如，对于如图 19 所示的用例，由于 SpiderMonkey 和 JavaScriptCore 这两个引擎均内置了“version”对象，所以能够正常地执行而不抛出任何异常；然而 Chakra 和 SpiderMonkey 引擎并没有实现这个内置对象，因此将会在执行第 8 行代码时抛出 ReferenceError 异常。该用例会触发多个引擎之间的不一致行为，所以会被认为是可疑用例，然而该用例仅仅是由于引擎的实现特性不同，而并非是由真正的引擎缺陷所导致的。类似这样的用例

⁵ <https://github.com/topics/javascript?l=javascript&o=desc&s=stars>

⁶ 代码文本中可能包含注释

会给后期的人工分析带来极大的困扰，因此本文选择在模型训练之前就从语料库中删掉相关的数据，以减少后续的人力成本。

```
1  var count = 0;
2  [].length = {
3    valueOf: function() {
4      count++;
5      return 1;
6    }
7  };
8  print(version);
```

图 19 由“version”关键字引起的误报用例

紧接着，为了缩减模型训练时的词汇表大小，本文过滤掉了所有包含非英文字符的数据；另外还过滤掉了字符长度在 50 以下以及 1000 以上的数据，本文认为这样过短或者过长的数据不利于模型学习。

最后，由于这些数据是从不同的开源项目中获取到的，因此其具有不同的编码和注释风格。因此，为了让模型能够更好地学习代码本身的语法和语义信息，本文使用在 Esprima 和 Esgen 工具⁷去掉了 JavaScript 文件中所有类型的注释，并对代码格式进行了统一。

经过上述一系列的处理，最终本文的 JavaScript 代码语料库中保留了由 125966 个 JavaScript 文件构成的共计 47.8MB 的代码文本。

4.2.3 模型构建和训练

神经语言模型是一类模型的统称，并没有要求具体的模型结构，本文使用主流的 LSTM（Long Short Term Memory Network，长短期记忆网络）模型作为底层实现，它是一种特殊的循环神经网络模型，能够通过门控神经单元来选择性地对信息进行遗忘和添加，通过这种方式它能够学习数据间到更长期的依赖关系，克服了传统模型存在的难以处理长序列数据的问题。

本文采用流行的深度学习框架 Pytorch⁸来构建模型，模型整体由词嵌入层、LSTM 层和全连接输出层三部分构成。其中，词嵌入层负责实现训练数据的分词和编码成向量的功能，数据分词是指将训练数据（即 JavaScript 代码）拆分成一个个的 Token 序列的过程，如何对代码数据进行分词是一个值得探索的问题，本文将在实验章节 5.3.2

⁷ <https://www.npmjs.com/package/escodegen>

⁸ <https://pytorch.org/>

进行具体介绍；随后，词嵌入层会将 Token 编码成向量，便于向模型中输入，此处设置词嵌入维度为 256；LSTM 层完成具体的计算和权重参数的调整，此处设置层数为 2，每层的神经元个数为 512；全连接层负责将每一步得到权重参数进行计算后，并获得一个概率分布，便于进行下一个 Token 的预测，此处设置神经元个数为 256。

在模型实现后便可以开始训练过程。训练数据为提前处理好的 JavaScript 语料库与数据，设置训练的批处理大小为 64，使用交叉熵损失函数来计算损失、Adam 优化器进行权重参数的优化，共训练 50 个迭代。最终，经过约 48 小时的训练，可以得到一个有效的神经语言模型，该模型能够完成根据代码上文自动续写代码下文的任务。

4.3 测试用例生成方法

本节将对本文使用的两种测试用例生成方法进行说明，一种是第三章介绍的基于标准文档分析的测试用例突变方法，另一种是基于神经语言模型的用例续写生成方法，这两种方法的共同点是都需要基于已有的种子用例来产生新用例，因此本节还将对本文使用的种子用例收集方法进行介绍。

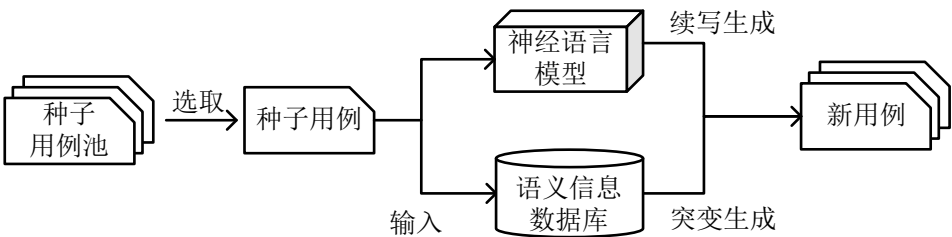


图 20 测试用例生成方法概述

图 20 是对测试用例生成方法的一个概述，首先需要收集到若干 JavaScript 测试用例作为种子用例，共同组成一个种子池，之后开始循环地生成用例。每次生成首先需要从种子池中随机选取一个种子用例，随后基于该用例分别使用续写生成和突变生成两种方式各自生成多个新用例，最后将所有生成用例合并，便可以开始后续的模糊测试阶段。

4.3.1 种子用例收集

种子用例将直接决定生成的新用例的质量，确定适当的种子用例选取方法是很有必要的。章节 4.2.1 介绍了从 GitHub 上的开源项目中获取 JavaScript 代码的方法，然而，开源项目中的代码基本都是使用 JavaScript 编写的业务代码，业务代码通常逻辑比较重复，且语法特性覆盖较少，因此不能充当有效的种子用例。

上文提到, Suyoung Lee 等人发现了触发新的 JavaScript 安全漏洞的代码通常由已存在的回归测试中已有的代码片段组成这一规律,它为本文提供了新的种子用例选取思路,即选择 JavaScript 引擎的回归测试用例作为种子,能够更有效地生成触发引擎缺陷的新用例。

因此,本文下载了多个引擎的回归测试套件作为种子用例,为了弥补数量上的不足,本文还额外从 JavaScript 的官方测试套件 Test-262 中获取了用例。同样的,种子用例也需要进行数据预处理,除了章节 4.2.2 中介绍的常规预处理的步骤之外,还需要额外进行一个“个性化消除”的步骤,因为种子用例都是从不同引擎的测试套件中收集的,因而带有不同引擎的特性。而不同引擎的实现特性之间存在差异,这就使得一个引擎的测试套件难以在另一个引擎上运行,因此需要人工制定规则来对这些个性化的差异进行消除。例如对于从 ChakraCore 引擎中获取到的用例就需要使用“print”替换掉“WScript.Echo”语句,因为前者是所有 JavaScript 引擎通用的语法,而后者只是 ChakraCore 引擎独有的特性。若不进行此替换,该特性就会导致基于该种子用例生成的新用例在后续的模糊测试过程中直接被其他引擎抛出异常,无法正常执行。

最终,本文从各个引擎的回归测试套件以及 Test-262 测试套件中收集到了共 7564 条种子用例,这些用例共同组成种子池,供后续的用例生成方法来使用。

4.3.2 语义信息指导的用例突变生成方法

本文首先使用语义信息指导的用例突变生成方法来产生新用例,该方法在第三章进行了详细的介绍,因此此处仅给出一个具体实例。

图 21 是用例定向突变方法的一个实例。Array.prototype.fill 方法的功能是用特定值对数组进行填充,其接收 3 个参数: value、start 和 end,其中 value 表示要填充的值, start 和 end 分别表示填充的开始和结束索引。本文通过对其对应的 ES 标准进行解析,得到如图 21(a)所示的语义信息,其中 value 参数未能解析出有效的信息;而 start 参数有 3 个候选值,即[1, -1, 0];而 end 参数有 4 个候选值,即[undefined, 1, -1, 0],因此其参数共有 12 种有效组合。因此,会依次将该用例按这 12 种参数组合对图 21(b)所示的种子用例进行突变,得到如图 21(c)所示的共 12 个新测试用例。

与原始的测试用例相比,突变后的多个新用例能够覆盖到引擎更多的代码分支,增加了检测范围,从而能够更有效地检测出引擎更多的潜在缺陷。

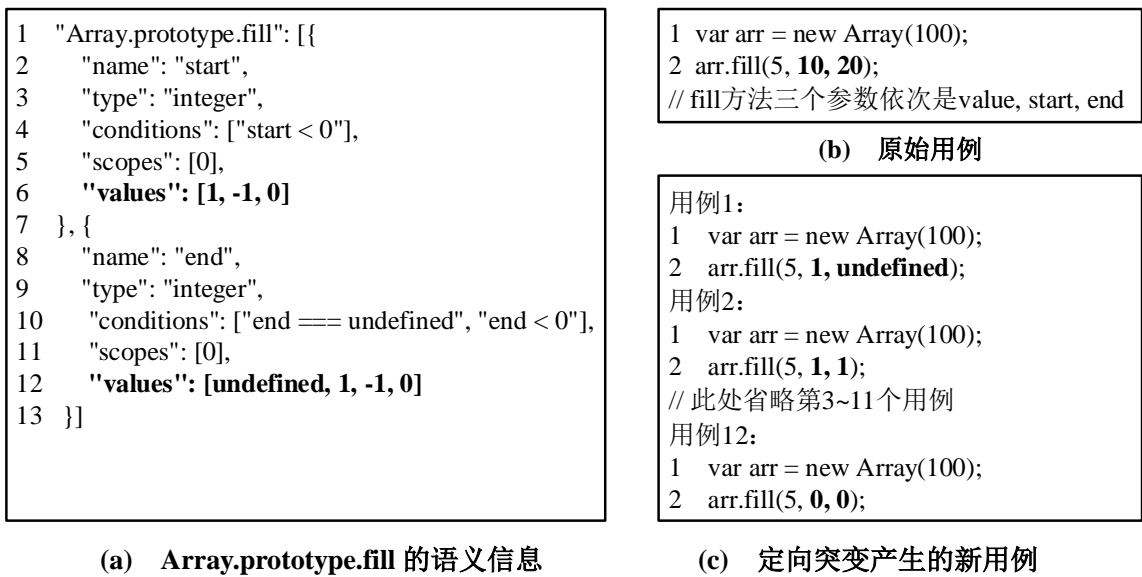


图 21 语义信息指导的用例定向突变实例

4.3.3 基于神经语言模型的用例续写生成方法

本文使用的第二种用例生成方法是基于神经语言模型的用例续写生成方法，该方法包括两个步骤，其一是从种子用例中选取适当的突变点以确定代码上文，其二是使用神经语言模型根据代码上文自动续写代码下文，并将上下文拼接以获得最终的新用例。突变点实际上就是一个字符的索引序号，从种子用例的开始处到突变点索引的这一部分代码内容称为代码上文。突变点的选取通常是随机的，得到突变点后，便可以从种子用例中截取出对应的代码上文，用于下一步对其进行续写生成。

续写生成实际上是使用训练好的神经语言模型不断地根据现有上文预测下文的过 程。首先将代码上文切分成 Token 形式后编码成向量，分时间步依次输入模型，每个时间步模型都会计算出一个概率分布以及一个新的隐藏状态，其中概率分布是对下一个 Token 的预测，而隐藏状态则是模型从当前输入中学习到的信息。当输入完所有代码上文的 Token 之后，得到的隐藏状态便可视作模型从代码上文中学到的全部信息，此时便可以开始采样。采样实际上就是指根据预测的概率分布按照一定策略选取下一个 Token 的过程，本文使用 top-k 采样方法，并根据实践经验将 k 设为 10，即每次从概率分布中取概率值最大的 10 个 Token，将其归一化后再进行按概率采样，以得到最终的 Token，然后根据词汇表将 Token 映射成对应字符，作为续写生成的内容。重复多次这样的采样过程，直到达到生成上限或者某种终止条件为止，这样就能根据代码上文自动地续写下文。图 22 是使用续写生成方法生成新用例的一个实例，从种子用例图 22(a)中随机选取突变点，截取出图 22(b)所示的代码上文，之后使用语言模型

根据代码上文续写下文，并将上下文拼接得到如图 22(c)所示的新用例，该用例有效地触发了 V8 引擎关于 `TypedArray.prototype.set` 方法实现不正确的缺陷⁹。

```
1 var a = new Int32Array([1, 2, 3, 4, 5]);
2 print(a.subarray().toString() === '1,2,3,4,5');
3 print(a.subarray(3).toString() === '4,5');
```

(a) 种子用例

```
1 var a = new Int32Array([1, 2, 3, 4, 5]);
2 print(a.subarray().toString()[突变点])
```

(b) 根据随机突变点截取的代码上文

```
1 var a = new Int32Array([1, 2, 3, 4, 5]);
2 print(a.subarray().toString());
3 a.set(a.length, a.length);
```

(c) 续写生成的新用例

图 22 语言模型续写生成的用例实例

4.4 差分模糊测试方法设计

在获取到新生成的用例之后，便可以使用该用例对 JavaScript 引擎进行差分模糊测试，以检测引擎的一致性缺陷。本节将详细描述本文所使用的差分模糊测试的设计思路 and 具体实现过程。

4.4.1 模糊测试的执行结果说明

章节 2.2.3 描述了差分模糊测试的一般流程，即通过对多个引擎执行同一个用例的结果进行统计分析，从而得到执行结果不一致的引擎，并加以人工分析以确定该引擎是否存在缺陷。因此，首先要明确引擎执行一个用例的“执行结果”中都包含哪些有效的信息。

JavaScript 引擎完整地执行一个用例的过程实际上就是操作系统中一个进程从生到死的过程。在 Linux 系统下，每个进程的生命周期都伴随着三种数据流：标准输入流（Standard Input）、标准输出流（Standard Output）和标准错误流（Standard Error），进程执行过程中将从标准输入流中读取输入数据，并将正常的打印数据输出到标准输出流中，而将错误信息发送到标准错误流中。当 JavaScript 引擎执行一个测试用例时，标准输入流就是用例的内容，标准输出流就是用例执行过程中被打印语句所输出的内

⁹ <https://bugs.chromium.org/p/v8/issues/detail?id=11294>

容，而标准错误流就是执行用例时抛出的异常信息。在差分模糊测试时，由于多个引擎执行的是同一个用例，所以它们的输入流都是相同的，不会存在差异，因此该信息对差分测试来说是无用的，故舍去；而标准输出流和标准错误流取决于引擎执行用例时的具体行为，不同的引擎有可能会产生不一致，因此这两个信息是有效的，需要对其进行收集和统计。

另外，当一个进程执行结束时，会向操作系统发送一个信号，同时伴随着一个 0 到 255 范围内的整数值，称为返回值 (Return Code)，它标志着此进程的执行状态^[52]。例如返回值 0 表示进程正常执行完，没有发生任何异常；1 表示进程遇到异常产生了中断；3 表示进程被强制退出；9 表示进程被操作系统杀死（比如执行超时）；11 表示进程执行时发生了无效的内存引用等¹⁰。不同的引擎执行同一个用例时其返回值也有可能会不同，因此返回值也是需要记录的。

综上所述，在本文中设定一个引擎对一个用例的“执行结果”共包括返回值、标准输出流和标准错误流这三个有效信息，因此在引擎执行用例时需要对其进行专门地记录，而收集到的执行结果信息将用于下一步的结果差分阶段。

4.4.2 差分机制设计

上一节对引擎执行一个用例时得到的执行结果信息进行了描述，本节将介绍如何利用这些信息基于章节 2.2.2 中描述的差分测试思想，通过分析其中的不一致行为来寻找可疑用例，进而对引擎的潜在缺陷进行挖掘。

本文的差分思路是，对每个引擎执行结果中的三种信息进行分析，进而将其分成四个类别，每个类别分别代表着不同的引擎行为。当多个引擎执行结果的类别中存在着类别与其他引擎不一致的情况时，便认为当前用例触发了引擎的不一致行为，将其标记为可疑用例。表 6 是对本文划分的四种类别的介绍。

表 6 引擎执行结果的四种类别

英文名称	中文名称	类别描述
Pass	通过	引擎正常执行完该用例
Timeout	超时	引擎执行该用例超时
Crash	崩溃	引擎执行该用例时产生崩溃
RuntimeError	运行时错误	引擎执行用例时抛运行时异常

¹⁰ 不同的操作系统和指令集，其返回值代表的含义略有差异。

例如，当四个引擎执行同一个用例，能得到四个执行结果信息，并将其分类。当四个执行结果的类别不完全相同时，便认为该用例触发了引擎的不一致行为，该用例就是可疑用例。

需要注意的是，不同的引擎对于用例执行结果的表现方式上有着不同的方式，并没有严格按照返回值、标准输出流和标准错误流三部分的定义来实现。例如，ChakraCore 引擎在执行用例时即使因抛出异常而终止，其返回值也仍然为 0；V8 引擎在执行用例时，抛出的异常信息并不保存在标准错误流中，而是输出到标准输出流，因此在设计执行结果分类算法时需要针对各个引擎的具体实现情况分别讨论。

图 23 是用例结果差分的一个实例，使用四个不同的引擎执行图 23(a)所示的用例后得到四个执行结果，将其分类后便可以通过差分思想确认是否触发引擎的不一致行为。如图 23(b)所示，四个执行结果中只有 ChakraCore 的类别是 Pass，其余三个都是 RuntimeError，说明该用例触发了引擎的不一致行为，即 ChakraCore 可能存在缺陷，因此需要将该用例视为可疑用例，记录下来供后续分析。

```
1 function func() {
2   this++;
3 }
```

(a) 测试用例¹¹

①ChakraCore-V1.11.24(a75335b): - return_code: 0 - stdout: (empty) - stderr: (empty) 分类结果: Pass	②JavaScriptCore-d940b47: - return_code: 3 - stdout: SyntaxError: Postfix ++ operator applied to value that is not a reference - stderr: (empty) 分类结果: RuntimeError
③SpiderMonkey-C69.0a1: - return_code: 3 - stdout: (empty) - stderr: SyntaxError: invalid increment/ decrement operand 分类结果: RuntimeError	④V8-e39c701: - return_code: 1 - stdout: SyntaxError: Invalid left-hand side expression in postfix operation - stderr: (empty) 分类结果: RuntimeError

(b) 四个引擎的执行结果及类别

图 23 用例结果差分实例

¹¹ <https://github.com/chakra-core/ChakraCore/issues/6550>

4.4.3 种子池扩充方法设计

前文介绍了本文所使用的用例生成方法以及差分模糊测试方法，循环执行这两个过程便能自动化地对 JavaScript 引擎进行缺陷检测。然而，由于本文使用的两种用例生成方法本质上都是基于种子用例来产生新用例，因此新用例的质量和测试覆盖范围都极度依赖于种子用例，又因为种子用例的数量有限，所以新用例的测试范围也会相应地受到限制，不利于进行长期的模糊测试过程。因此，为了进一步提高缺陷检测的效率，本文还设计了反馈式的种子池扩充方法。

种子池扩充，直观来说就是向种子池中不断添加有效的测试用例作为新的种子，增大种子用例的测试范围，以便于提升新测试用例触发引擎缺陷的可能性。那么，如何选择新的种子便成为了关键问题，新种子用例来源于测试过程中产生的可疑用例，但不能盲目地将所有可疑用例都加入种子池中，否则会导致池中相似的种子用例的数量增加，而稀释了独特的种子用例的占比，这样反而降低了整体的测试效率。因此，需要设立合理的“新种子用例”的标准。

所谓“新”并不是要求测试用例本身是新的，而是希望引擎执行该用例的“行为”是新的，即现有种子用例都无法触发引擎的该项行为。本文设计并实现了“状态码判重法”来实现新种子用例的选取，“状态码”是将引擎执行结果的四种类别进行细分后得到的，代表着引擎执行该用例时的行为。通过将新的用例的状态码与种子池现有种子用例的状态码进行对比，便可以确定新用例是否触发了引擎新的行为，据此来判断该用例是否是“新的”。本文共设定了九种不同的状态，如表 7 所示。

表 7 引擎执行结果的九种状态

状态码	英文名称	中文名称	状态描述
1	Pass	通过	引擎正常执行完该用例，未抛出任何异常
2	Timeout	超时	引擎执行该用例超时
3	SyntaxError	语法错误	引擎执行时抛出语法错误
4	RangeError	范围错误	引擎执行时抛出范围错误
5	ReferenceError	引用错误	引擎执行时抛出引用错误
6	TypeError	类型错误	引擎执行时抛出类型错误
7	URIError	标识符错误	引擎执行时抛出标识符错误
8	OtherErrors	其他错误	引擎执行时抛出其他错误（如自定义异常）
9	Crash	崩溃	引擎执行时崩溃

用多个引擎执行同一个用例，并将每个执行结果都按照其行为映射成相应的状态

码，便会得到多个状态码，将其合并后作为该用例最终的状态码。例如，对于同一个用例，JavaScriptCore 引擎执行通过，没有抛出任何异常（对应状态码 1），而其余三个引擎都抛出引用错误（对应状态码 5），则确定该用例最终的状态码便为“1555”。若两个用例的状态码完全相同，则说明这两个用例触发的是引擎相同的行为，那么就认为它们是重复的。

在实现上，本方法会预先将种子池中的所有种子用例执行一遍模糊测试，记录下每个种子用例的状态码。每当出现一个新的可疑用例，便会将该用例的状态码与种子池中用例的状态码进行对比，若已存在该状态码，说明该用例的行为已经在种子池中出现过了，便将该用例视为“旧的”并将其丢弃；若不存在，便将其视为新的有效可疑用例，接着将其作为新的种子用例扩充到种子池中，以供后续使用。

状态码判重法不仅能够确定新种子用例，还能实现可疑用例的过滤。由于模糊测试是一种自动化测试，所以在长期的测试过程中会积累大量需要人工分析的可疑用例，而这些用例中可能会存在许多重复，对其进行人工分析会耗费大量不必要的人工成本。使用状态码判重法可以对可疑用例进行去重，只保留那些出现了新的引擎行为的用例，这样能够有效地减少人工分析的成本，从侧面提高发现引擎缺陷的效率。

4.4.4 人工分析

本文提出的测试方法的最后一步便是对去重后的可疑用例进行人工分析，以确定其是否真的为引擎缺陷。人工分析主要包括用例精简、缺陷定位、标准推演以及编写并提交报告等四个步骤，首先需要根据执行结果对用例进行不断地精简，直到能够定位到触发引擎不一致行为的具体代码；随后查询 ES 标准，定位到该代码对应到标准中的具体条目；之后人工按照标准的内容进行推导，以得到标准规定的预期行为。最后将其实际行为与预期行为进行作对比，若引擎行为不符合标准预期，则说明该引擎在实现上确实存在缺陷，接着便编写缺陷报告，提交给开发商，等待后续确认；若一致，则说明本次属于误报，故跳过。

4.5 本章小结

本章主要对本文所使用基于差分模糊测试的 JavaScript 引擎一致性缺陷检测方法进行了详细的介绍。首先对测试方法的整个流程进行了概括，说明几个主要步骤，随后依次对每个步骤进行了重点介绍。本方法包含生成模型构建、用例产生方法以及差分模糊测试方法设计三个部分。在生成模型构建部分，本文介绍了从收集数据到模型

训练的整个过程；在用例产生方法部分，结合实例说明了本文所使用这两种不同的用例产生方法，即突变生成和续写生成；在差分模糊测试阶段，说明了本文所设计的差分模糊测试系统的原理、实现过程以及设计优势。

第五章 实验评估与分析

第三章和第四章分别对本文的两个核心内容,即基于标准文档分析的测试用例突变方法以及基于差分模糊测试的一致性缺陷检测方法进行了介绍,本文所提出的方法以差分模糊测试为主要指导思想,以测试用例定向突变为辅助手段,能够有效地对 JavaScript 引擎的一致性缺陷进行检测。本文依据该方法设计并实现了名为 ESfunfuzz 的原型系统,并基于多个维度对其进行了实验评估,证明其拥有较强的缺陷检测能力。

5.1 ESfunfuzz 原型系统的设计与实现

5.1.1 系统模块设计

参照第三章与第四章所描述的方法,本文实现了一个名为 ESfunfuzz¹²的原型系统,该系统的模块设计如图 24 所示。

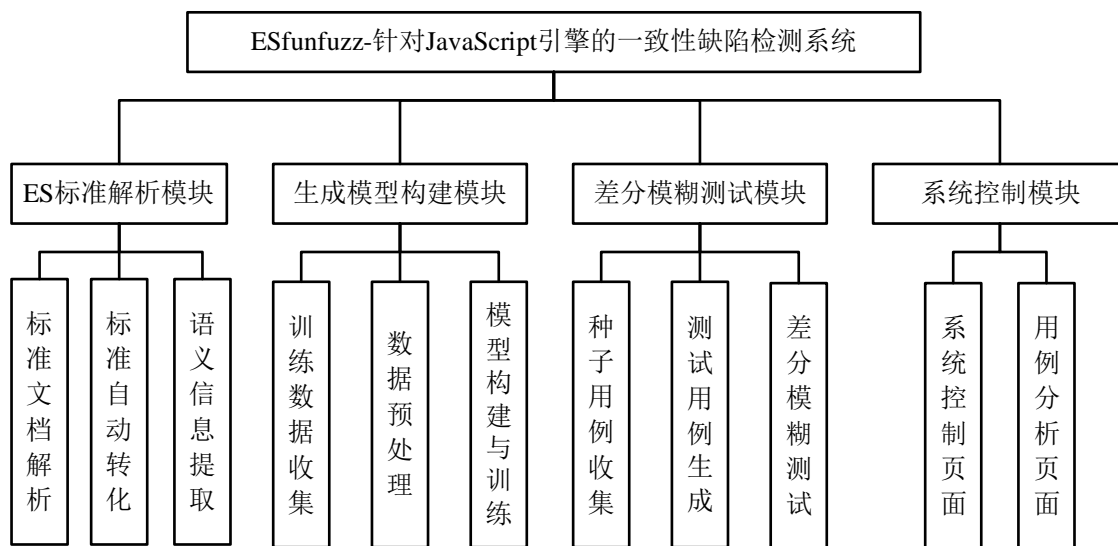


图 24 ESfunfuzz 原型系统模块设计总览

ESfunfuzz 系统包含以下四个主要模块：

(1) ES 标准解析模块

本模块主要负责完成从 ES 标准中提取出有效的语义信息的功能,该信息将在后续的模糊测试过程中指导用例进行定向突变。本模块主要包含三个步骤:首先需要将 ES 标准的 HTML 文档进行解析,以得到 ES 标准的文本;之后按照人工定义的正则规则,将 ES 标准的自然语言文本转化为对应的 JavaScript 代码;最后从 JavaScript

¹² 命名为 ESfunfuzz 是为了向经典的 JavaScript 引擎模糊测试工具 jsfunfuzz 致敬。

代码中解析出有效的语义信息，即与传入参数有关的边界值和特殊值，并据此总结出传入参数的候选值，以指导后续的用例变异。

（2）生成模型构建模块

本模块主要用来完成代码生成模型（即语言模型）的训练过程，训练好的模型将用来基于给定的代码上文自动续写下文，以产生新的测试用例。本模块主要包含三个步骤：首先从 GitHub 平台上收集 JavaScript 文件作为训练数据；随后对其进行数据预处理，主要是去掉过长、过短、包含特殊字符、大量重复的数据，将处理后的数据作为最终的训练数据；最后使用 PyTorch 深度学习框架构建一个基于 LSTM 结构的神经语言模型，并使用上一步得到的数据进行训练，训练好的模型将作为代码生成模型用于后续的用例生成步骤。

（3）差分模糊测试模块

本模块是 ESfunfuzz 系统的核心模块，主要完成运用差分模糊测试的思想对 JavaScript 引擎进行一致性缺陷检测的过程。本模块主要包含三个步骤：首先需要收集到若干有效的测试用例作为种子池；随后从种子池中随机选择一个种子用例，并同时使用语言模型续写以及语义信息突变两种方法基于该种子用例来产生多个新的测试用例；最后按照章节 2.2.3 中描述的差分模糊测试过程执行新的测试用例。这个过程将不断地循环，并记录下所有触发过引擎不一致行为的用例。

（4）系统控制模块

本模块是 ESfunfuzz 系统的控制模块，为了更方便地控制整个模糊测试系统的运行，以及更高效地对可疑用例进行人工分析，本文另外实现了一个 Web 系统。该系统主要包含系统控制页面和用例分析页面两个主要子模块，前者负责对整个系统的开始、结束、自定义配置等进行控制，后者则集成了多个对用例进行人工分析时所需要的便捷操作，以提高人工分析的效率，力图在单位时间内发现尽可能多的引擎缺陷。

5.1.2 系统具体实现

整个 ESfunfuzz 原型系统共包含约 3,000 行 Python 代码、2,000 行 JavaScript 代码、1,500 行 Java 代码以及部分用于编写网页的 HTML、CSS 以及 JavaScript 代码。其中，Python 代码主要负责实现语言模型训练模块和差分模糊测试模块；JavaScript 代码负责实现 ES 标准解析模块；Java 与其他网页代码负责实现系统控制模块（即 Web 系统）。本着公开透明的原则，ESfunfuzz 原型系统所有代码和数据均已在 GitHub 平台上实现

开源¹³，以供参考和审阅。

5.2 实验环境与实验设置

5.2.1 实验环境

硬件方面，本文的实验平台是一台搭载了主频 3.6GHz 的 Intel i7-7820x 处理器和 Ubuntu 18.04（内核版本 4.15）操作系统的高性能服务器，内存为 64GB，图形显示卡为四张 RTX 2080Ti 显卡。

软件方面，本文实验主要的开发语言是 Python、JavaScript 以及 Java，所用到的 Python 解释器版本为 Python v3.7.5，JavaScript 的解释器版本为 Node v12.18，Java 开发工具包版本为 1.8。

5.2.2 实验设置

本文主要对 ESfunfuzz 原型系统进行三个方面的实验：

（1）语言模型性能调优实验：由于 ESfunfuzz 内部使用了神经语言模型来生成测试用例，而神经语言模型的效果又会根据训练数据以及参数的配置而产生上下浮动，因此，为了使得 ESfunfuzz 系统的综合性能达到最佳，首先需要对语言模型进行调优实验，即训练数据的编码级别实验，该实验将对神经语言模型的训练数据采用不同的编码级别，通过比较生成用例的质量来探究最适当的编码级别。

（2）与其他工具的对比实验：目前针对于 JavaScript 引擎进行缺陷检测的研究不在少数，因此，为了验证本文提出的方法的有效性，需要与当前最先进的其他检测工具进行对比。对比实验分为两组，分别是生成用例的质量评估实验以及发现引擎缺陷的能力实验，前者将使用多种工具分别生成一定数量的用例，通过比较其语法正确率和语义正确率来衡量其质量；后者将使用本文提出的系统框架去分别执行各个工具生成的测试用例，通过发现的缺陷数目来衡量各个工具发现引擎缺陷的能力。

（3）真实环境下的缺陷检测实验：想要验证缺陷检测方法的有效性，除了实验环境下验证以外，还必须能够在真实的环境下检测出实际存在的引擎缺陷。因此，最后一部分实验便是真实环境下的缺陷检测实验，其主要包含两个实验：其一是使用 ESfunfuzz 系统对真实的四个主流 JavaScript 引擎进行长期的缺陷检测，以便于获得真实的引擎一致性缺陷；其二是从发现的缺陷实例中挑选几个典型的案例进行分析研

¹³ <https://github.com/ty5491003/ESfunfuzz>

究，探明本文提出的方法的生效机制，以说明本文方法的有效性。

5.3 语言模型调优实验

本文的检测方法基于模糊测试机制，通过生成大量的测试用例来对 JavaScript 引擎缺陷进行自动化地检测，生成用例的质量将直接决定检测方法的效果。本文有两种用例生成方法，其中，基于标准文档分析的突变式方法由于是直接在语法树级别操作，所以一定不会带来语法错误，故不需要调优；而用例续写生成方法由于是使用神经语言模型进行自动续写的，可能会导致生成语法不正确的用例，对模糊测试的效率影响较大。因此，ESfunfuzz 的性能调优就是指对语言模型的性能进行调优。

5.3.1 语言模型性能的衡量标准

要对语言模型性能进行调优，首先需要确定一套适当的质量评估标准，本文结合实际的代码生成场景，将模型生成的测试用例的通过率（Passing Rate）作为衡量模型性能的标准。通过率是指模型生成的若干用例中没有发生“运行时错误”的用例的比例，例如生成 1000 条用例，其中有 800 条在执行时没有发生运行时错误，则本次生成的通过率为 80%。运行时错误按照 ES 标准的规定主要分为语法错误、范围错误、引用错误、类型错误和标识符错误五类。本文将使用 ChakraCore v1.11.24 引擎¹⁴来统计用例的通过率。

语言模型生成的用例的通过率越高，一方面说明该模型对语料库学习得越好，即对 JavaScript 语言的编码规则和语法语义信息理解得越透彻；另一方面说明该模型生成的有效用例越多，进行模糊测试会越高效，在相同的时间内能检测出更多的缺陷。

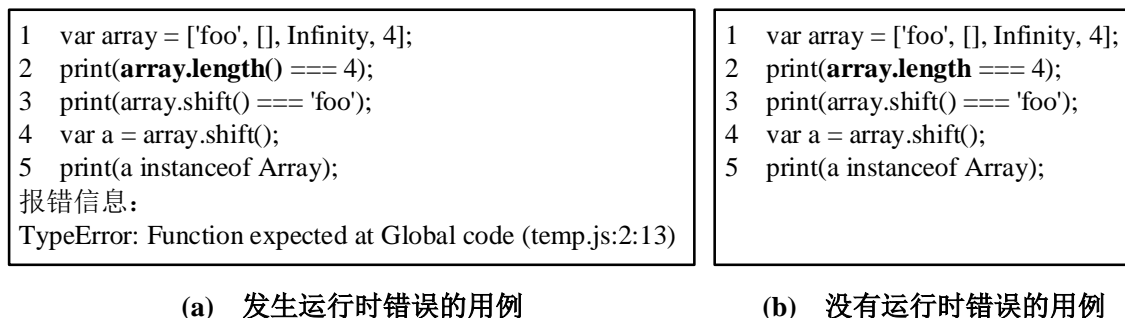


图 25 运行时错误的用例示意

图 25(a)是发生运行时错误的一个用例实例，该用例是由语言模型自动生成的，然而，由于模型对 JavaScript 语义的学习不够充分，将数组 array 的属性 length 误以

¹⁴ <https://github.com/chakra-core/ChakraCore/releases>

为是方法并加以调用，使得该用例在执行时抛出了类型错误（`TypeError`），该错误导致用例的第 3~6 行没有被执行到，这种用例往往难以触发引擎的不一致行为，属于无效用例。与之对比，用例 25(b)是一个没有运行时错误的用例，它对 `length` 属性的用法是正确的，因此不会抛出任何异常，用例的所有语句都会被执行到，它就属于有效用例。显然，我们更希望模型生成的是用例 25(b)而不是 25(a)。

5.3.2 数据编码级别实验

数据的编码级别是对模型效果影响最大的因素之一，编码级别是指在将训练数据编码成向量的过程中，基于怎样的粒度来进行切分，切分后的词统称为 `Token`。常见的编码级别有字符（`Char`）级别、词级别（`Word`）等，本文除了这两种传统的级别以外，还选择了字节对编码（`Byte-pair-encoding`, `BPE`）级别，它是当今自然语言处理领域内流行的 `Subword` 算法的一种具体实现，能够有效地应对词汇表过大以及未登录词¹⁵的问题^[53]。本实验使用 `Esprima` 工具来实现 `Word` 级别编码；使用 `SentencePiece` 工具¹⁶来实现 `BPE` 级别的编码。

本实验的具体实施步骤是：使用章节 4.2.3 中介绍的同一套超参数，分别按上述三种不同编码形式对数据进行编码后训练，每个模型训练 50 个迭代（`Epoch`），每 10 次迭代保存一次模型（故三种编码方式下共得到 15 个模型）；之后分别使用这些模型使用续写生成方法各自产生 10000 条测试用例，最后使用 `ChakraCore` 引擎分别执行并统计这些用例的通过率指标。

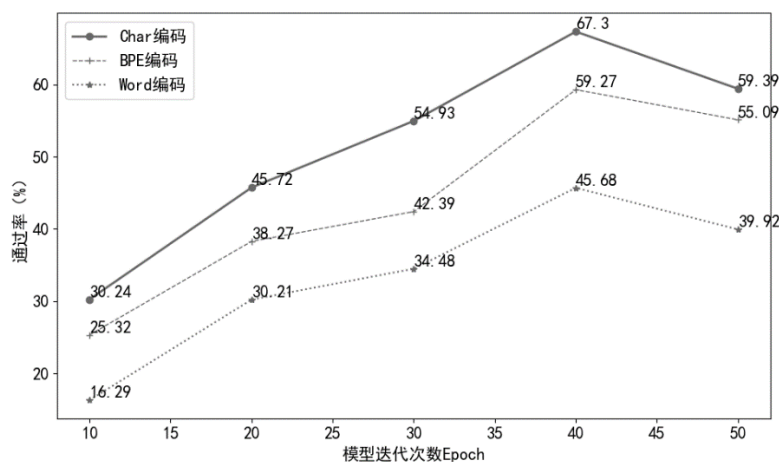


图 26 不同编码级别下生成用例通过率随 Epoch 变化示意图

实验结果如图 26 所示，横轴表示模型的迭代次数 `Epoch`，纵轴表示模型生成用

¹⁵ 未登录词是指当前输入的 `Token` 在词汇表中不存在的情况，它会直接导致模型的泛化性能下降

¹⁶ <https://github.com/google/sentencepiece>

例的通过率，共三条折线，分别表示 Char、Word 和 BPE 三种不同的编码方式，具体请参考图示。从图中可以看出，采用 Char 级别进行编码的模型生成用例的通过率明显优于其他两种编码，在每个 Epoch 下的表现都是最优的；BPE 编码的模型效果居中，Word 编码的效果最差。

通过对 Word 编码的模型生成的用例进行分析，本文发现其通过率低的原因是 Word 编码下难以解决未登录词的问题，即当截取的代码上文中存在着词汇表中没有的 Token 时(比如一个新字符串或者新数值)，就只能将其替换为占位符“<UNK>¹⁷”，并进行后续的生成步骤，这在自然语言处理领域中是通用的做法，因为自然语言中可以根据上下文推测缺失的词的含义。然而，这对代码生成任务来说却是致命的，因为编程语言都有严格的语法限制，只要有一点不满足要求就会直接抛出语法错误，进而导致整个用例都无法执行；而 Char 编码是字符级别的编码，所有常见字符已经全部收录在词汇表中，故不存在未登录词问题；BPE 编码则是通过将词拆分为子词的方式在一定程度上缓解了该问题，因此这两种编码的效果普遍优于 Word 编码。从图中还可以发现，模型的表现普遍随着迭代次数的增加呈现出先上升后下降的趋势，并在 Epoch 为 40 时达到最优，猜测可能是 40 之前模型还未充分对训练数据进行拟合，而 40 之后可能出现了过拟合的原因。

总的来说，生成效果最好的是采用 Char 级别编码在 Epoch 为 40 时的模型，使用该模型生成的用例通过率达到 67.3%，即生成的 10000 个用例中有 6730 条都是没有抛出任何异常的有效用例，这对生成 JavaScript 这种语法严格的编程语言来说属于较高水平，后续的实验也将基于此模型进行。

5.4 与其他工具的对比实验

为了进一步验证本文方法的效果，需要将 ESfunfuzz 与当前最先进的其他模糊测试工具进行对比实验。本实验主要关注两个方面，即生成用例的质量以及缺陷检测的能力。本文选择的对比对象有四个，分别是 Fuzzilli、CodeAlchemist、Montage 和 DIE，对它们的具体介绍参考章节 1.2，它们都是近几年来最先进的针对于 JavaScript 引擎的模糊测试工具，与它们进行对比可以充分说明 ESfunfuzz 的有效性。

¹⁷ <UNK>意为“Unknown Word”，通常会用它替换掉未登录词。

5.4.1 生成用例质量对比

生成用例的质量主要通过两方面来衡量，其一是用例的通过率，通过率越高则意味着测试效率越高；其二是代码覆盖率，代码覆盖率表示测试用例被执行到了多少，覆盖率越高说明用例的利用率越高，就也有可能触发引擎缺陷。覆盖率可以进一步细分为三种类型，即行覆盖率、分支覆盖率和方法覆盖率，分别表示语句行、条件分支和方法被执行到了多少，通过它们能对用例的质量进行一个比较全面的衡量。

本实验的具体实施步骤是：分别配置好 ESfunfuzz 与其他四种工具，基于本文获得的相同的语料库数据，使用它们各自生成 10000 条测试用例，最后使用 ChakraCore 引擎统计生成用例的通过率，使用 Nyc 工具¹⁸统计三种覆盖率，通过四个指标共同衡量生成用例的质量。

统计结果如图 27 所示，横轴为不同的评估标准，纵轴为评估的结果百分比，每个评估标准都有五组结果，分别代表着五种不同的模糊测试工具，具体表示请参考图示。可以观察到，Fuzzilli 工具生成的用例，其通过率、行覆盖率和方法覆盖率上都领先于其他工具；而本文提出的 ESfunfuzz 在分支覆盖率上领先，在其他三个指标上略微落后于 Fuzzilli，综合排名第二；紧随其后的是 DIE 工具，它与本文提出的 ESfunfuzz 在生成用例质量上相近，并无明显差距。最后是 CodeAlchemist 和 Montage 工具，它们生成用例的四个指标与其他三种工具生成的用例差距较为明显。

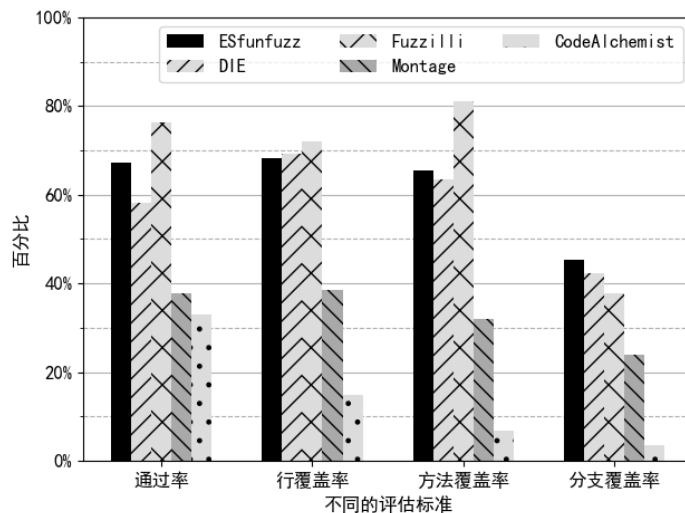


图 27 用例质量评估结果对比

本文分析，Fuzzilli 工具生成的用例质量之所以高，是由其生成用例的方法的特性决定的，它是通过自定义中间语言实现的用例生成和突变，不需要任何种子用例的

¹⁸ <https://www.npmjs.com/package/nyc>

支持，因此不会受种子用例的质量的影响，拥有比较稳定的生成质量；而其他四种工具都需要预先收集的种子用例做支持，种子用例质量将直接影响生成用例的质量，CodeAlchemist 和 Montage 生成的用例质量较低可能就是由于种子用例不适用造成的。

通过这组实验证明，本文提出的 ESfunfuzz 原型系统生成的测试用例本身拥有较高的质量，即使不是几种工具中最高的，但整体也处于上游地位。需要注意的是，对各个模糊测试工具生成的用例本身进行质量评估只是一个参考，仅仅表示该工具生成的用例的利用率高低，其并不能直接证明各个工具的缺陷检测能力。例如，工具 A 生成用例的质量很高，但其可能就是检测不出缺陷来；而工具 B 即使生成用例的质量较低，但却能够有效地触发引擎缺陷，这种情况也是有可能出现的。因此，除了对用例本身的质量进行评估外，还需要对各个工具的缺陷检测能力进行一个评估。

5.4.2 缺陷检测能力对比

评估一个模糊测试工具的缺陷检测能力，最直接且有效的方法就是统计其在一定时间内发现的引擎缺陷的数量^[54]，因此，本文还将通过比较各个工具发现的缺陷个数来衡量 ESfunfuzz 系统的缺陷检测能力。本实验的具体实施步骤是：使用 ESfunfuzz 与其他四种模糊测试工具分别进行为期 72 小时¹⁹的测试过程，并对所有检测出的可疑用例进行人工分析和缺陷提交，分别统计其触发和确认的缺陷个数，个数越多说明其缺陷检测能力越强。尤其需要注意的是，测试时需要将其他工具生成的用例统一放到本文的模糊测试框架下执行，以此来屏蔽各个工具由于语言和实现不同所带来的执行效率上的差异，保证测试的公平性。

实验结果如图 28 所示，横轴为不同的模糊测试工具，纵轴为发现的缺陷数量，共五组结果，每组结果包含两条数据，其中左边的黑色部分表示使用该工具检测并提交的缺陷数量，右边的灰色部分表示提交的缺陷中已被开发人员确认的数量。从图中可以看出，在为期 72 小时的模糊测试过程中，Fuzzilli 工具检测出了最多的缺陷，共 10 个，并且有 8 个已被确认；本文提出的 ESfunfuzz 紧随其后，共检测出了 8 个缺陷，其中 7 个被确认；之后分别是 DIE、CodeAlchemist 和 Montage。

¹⁹ 72 小时仅仅是执行测试的时间，后续的人工分析过程持续了两周。

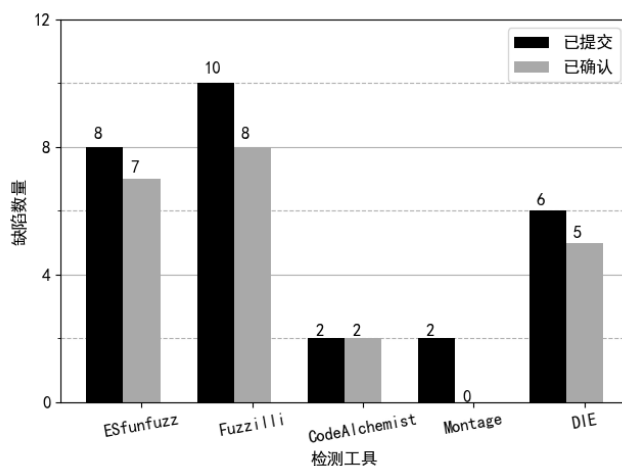


图 28 缺陷检测数量结果对比

这样的实验结果是可以预料的，因为排名靠前的三个工具，即 Fuzzilli、ESfunfuzz 和 DIE 都具有某种特定导向的功能，而并非是完全依靠随机方法来产生测试用例。其中，Fuzzilli 以覆盖率为导向来衡量新生成的用例是否需要被保留，被保留的用例将会持续地进行突变直到覆盖到引擎的新的行为，通过这种方式能够尽可能地覆盖到引擎的更多范围从而检测出更多的缺陷；而 ESfunfuzz 则基于本文提出的基于标准文档分析的定向突变方法以及反馈式地种子池扩充方法，一方面能够定向地对 JavaScript 中的特定目标进行检测，另一方面能够不断地扩充种子池中用例的测试范围，以便检测出更多的缺陷；DIE 工具通过保留种子用例中与触发缺陷相关的重要变量和代码依赖关系，替换其他无关的代码片段来实现对种子用例的变异，从而发现更多的引擎缺陷。而 CodeAlchemist 和 Montage 工具的用例产生方法则是完全随机的，并不具有任何的导向性，这种随机无疑会大大降低用例触发引擎缺陷的可能性。

Fuzzilli 是一种典型的白盒模糊测试工具，需要对引擎源码进行插桩以实时获取引擎覆盖率信息，白盒方法拥有较强的缺陷检测能力，但往往是高度定制的，故通用性较差，不便于迁移。而其余四种工具都是黑盒工具，即将待测目标视作一个黑盒，不会对其进行输入输出外的任何操作，拥有较强的通用性。

通过这组实验，一方面说明了导向性的用例产生方法确实优于随机性的用例产生方法，另一方面说明了本文提出的 ESfunfuzz 原型系统具有较强的缺陷检测能力，其结果仅次于 Fuzzilli，进而证明了本文提出的基于标准文档分析的用例突变方法的有效性。

5.5 真实环境下的缺陷检测实验

本章截止到目前,所安排的实验设置以及实验结果都很好地说明了本文提出的基于差分标准文档分析和差分模糊测试的 JavaScript 一致性缺陷检测方法的有效性。然而,检验一个模糊测试工具是否有效,并不能仅仅局限于实验环境下,也必须在真实环境下的检测任务中取得成果。

因此,笔者将使用本文提出的 ESfunfuzz 工具对章节 2.1.2 中介绍的四个主流 JavaScript 引擎进行一致性缺陷检测,来验证本文方法的有效性。选择它们进行测试的原因很多,一方面,这四个引擎对应的浏览器是目前市场占用率最高的四个,其全球市场占用率总和超过 92%,对其进行测试更有实际意义;另一方面,这四个引擎分别由 Apple、Microsoft、Google 等国际知名厂商开发,是无数优秀程序员智慧的结晶,代表着国际软件开发的最高水平。本文对其进行一致性缺陷检测,并且能发现真实存在的问题,足以证明本文所提出的方法的有效性。

最终,ESfunfuzz 成功检测出各类引擎的一致性缺陷 22 个,其中 19 个得到了开发者确认,具体的检测结果如表 8 所示。其中,缺陷状态为“确认&已修复”的表示该缺陷已经被引擎开发者确认并且修复;缺陷状态为“确认&未修复”的表示该缺陷已经被引擎开发者确认但暂时还未修复;状态为“待确认”的表示该缺陷已经被提交给了开发商,但目前尚未得到回复。

本文检测出的 22 个引擎缺陷中,ChakraCore 引擎占 10 个,JavaScriptCore 引擎占 10 个,V8 引擎 2 个,而 SpiderMonkey 引擎则没有检测出缺陷,笔者猜测可能是由于 SpiderMonkey 引擎的开发商 Mozilla 组织长期致力于模糊测试的研究,其内部开发了多种模糊测试工具(包括但不限于 jsfunfuzz、Dharma、grizzly²⁰等),并使用它们对 SpiderMonkey 引擎进行长期的内部测试,使得该引擎暴露出的问题较少,这也从侧面说明了模糊测试方法的有效性。通过本实验可以证明,ESfunfuzz 系统能够有效地发现真实环境下的 JavaScript 引擎中的一致性缺陷。

²⁰ <https://github.com/MozillaSecurity/grizzly>

表 8 真实环境下的缺陷检测结果²¹

序号	引擎及对应版本	缺陷描述	缺陷状态
1	ChakraCore-v1.11.12	变量重声明未报错	确认&未修复
2	ChakraCore-v1.11.24	this++异常语法未报错	确认&已修复
3	ChakraCore-v1.11.24	if(1)异常语法未报错	确认&未修复
4	ChakraCore-9e2f198	重定义 TypeError 异常	确认&已修复
5	ChakraCore-9e2f198	RegExp.prototype.toString 实现错误	确认&未修复
6	ChakraCore-2308fb0	%TypedArray%.prototype.sort 实现错误	确认&未修复
7	ChakraCore-2308fb0	Proxy 对象拦截 defineProperty 实现错误	确认&未修复
8	ChakraCore-2308fb0	RegExp 与 String 的 match 方法不关联	确认&未修复
9	ChakraCore-2308fb0	Array.prototype.push 自定义 set 方法失效	确认&已修复
10	ChakraCore-2308fb0	const 声明常量后可修改	确认&未修复
11	JavaScriptCore-d940b47	变量重声明未报错	确认&未修复
12	JavaScriptCore-d940b47	对 freezed 对象修改 length 未报错	确认&已修复
13	JavaScriptCore-d940b47	%TypedArray%.prototype.set 实现错误	确认&已修复
14	JavaScriptCore-d940b47	for...of...语法错误未报错	确认&已修复
15	JavaScriptCore-d940b47	已有属性重赋值不调用 set 方法	待确认
16	JavaScriptCore-d940b47	%TypedArray%.prototype.sort 实现错误	确认&已修复
17	JavaScriptCore-d940b47	Proxy 对象拦截 defineProperty 实现错误	确认&未修复
18	JavaScriptCore-d940b47	x 与 this.x 指向不同的错误	确认&未修复
19	JavaScriptCore-d940b47	Array.prototype.push 实现错误	确认&未修复
20	JavaScriptCore-d940b47	重定义属性时未更新属性	确认&未修复
21	V8-d891c59	%TypedArray%.prototype.set 实现错误	待确认
22	V8- e39c701	new 语法实现异常	待确认

5.6 典型案例研究

为了进一步说明本文提出的方法的有效性，本文从检测出的 22 个缺陷中挑选了 3 个典型的案例进行研究，通过对其缺陷原因和触发用例的来源进行分析，探明本文所提出的各个方法的具体生效机制，以证明其有效性。其中，案例一是通过第四章介绍的用例续写生成方法触发的；案例二是通过第三章介绍的用例定向突变方法触发的；而案例三则是通过章节 3.3.5 中介绍的非预期类型变异方法触发的。

²¹ <https://github.com/ty5491003/ESfunfuzz/blob/master/README.md>

5.6.1 案例研究（一）——续写生成方法的有效性验证

图 29 是证明本文所提出续写生成方法有效性的典型案例，其中用例(a)是从 JerryScript 引擎的回归测试套件中收集到的种子用例²²，而用例(b)是根据用例(a)的前两行使用神经网络模型自动续写生成的，续写所依据的代码上文即加粗部分。用例(b)触发了 JavaScriptCore 引擎(d940b47 版本)下修改被冻结的数组对象未报错的缺陷。

```
1 Array.prototype.push(1);
2 Object.freeze(Array.prototype);
3 try {
4   new RegExp().constructor.prototype.exec();
5 } catch ($) {}
```

(a) 种子用例

```
1 Array.prototype.push(1);
2 Object.freeze(Array.prototype);
3 print(Array.prototype.push.call(Array.prototype.push));
4 print(Array.prototype.push.length);
```

(b) 续写生成的触发缺陷的新用例

图 29 续写生成方式产生的用例实例^[55]

`Object.freeze` 是 JavaScript 语法中用来将一个数组对象“冻结”的方法，顾名思义，被冻结后的对象便成为一个不能被修改的“只读”对象^[56]，ES 标准规定当试图修改一个只读对象时引擎需要抛出 `TypeError` 异常。在用例(b)的第 2 行中，将 `Array.prototype` 对象进行了冻结，因此该对象便不能再被修改；然而在第 3 行中又通过 `push` 方法向 `Array.prototype` 对象中添加新的属性，即试图修改该对象的属性，此时按标准引擎理应抛出 `TypeError` 异常。然而在实际的执行时，除 JavaScriptCore 引擎之外的所有引擎都按标准抛出了该项异常，只有 JavaScriptCore 引擎既没有修改成功，也没有抛出任何异常，这种引擎执行不一致的行为立即被差分测试系统捕捉到，将其标记为可疑用例，并在后续的人工分析中被笔者确定为 JavaScriptCore 引擎的实现缺陷。

本用例很好地说明了本文设计的基于神经语言模型的测试用例续写生成方法的有效性，所有待测引擎在执行原始的种子用例(a)时全部执行通过，没有触发任何引擎的不一致行为。然而当它们执行根据用例(a)生成的新用例(b)时便立即暴露出了不一

22

<https://github.com/jerryscript-project/jerryscript/blob/8edf8d6eea4327dd83b7fabddcae4ea23bf98fb9/tests/jerry/regression-test-issue-1079.js>

致行为，并敏锐地被捕捉到，说明本方法能够触发种子用例触发不了的引擎缺陷。

尤其值得注意的是，从生成新用例到差分模糊测试再到捕捉不一致行为，这整个流程都是完全自动化的，期间不需要任何的人工参与，而仅仅在最后的可疑用例分析阶段需要人为介入，这也说明了本文的方法是非常高效的。

5.6.2 案例研究（二）——定向突变方法的有效性验证

<pre> 1 "TypedArray.prototype.sort": [{ 2 3 }, { 4 "name": "comparefn", 5 "type": "", 6 "conditions": ["comparefn !== undefined"] 7 "scopes": [], 8 "values": ["undefined"] 9 }] </pre>	<pre> 1 a = 10; 2 b = new Uint8Array(a); 3 function c() { 4 d; 5 } 6 b.sort(c); // TypedArray.prototype.sort </pre> <p style="text-align: center;">(b) 种子用例（精简后）</p> <pre> 1 a = 10; 2 b = new Uint8Array(a); 3 b.sort(undefined); </pre> <p style="text-align: center;">(c) 定向突变生成的新用例（精简后）</p>
(a) 从标准中解析出的语义信息	(c) 定向突变生成的新用例（精简后）

图 30 定向突变方法产生的用例^[57]

图 30 是证明本文所提出定向突变方法有效性的典型案例，图 30(a)是从 ES 标准中解析出的关于 `TypedArray.prototype.sort` 方法的相关语义信息，其核心内容是对 `comparefn` 的参数给出了候选值“`undefined`”；用例(b)是从 `JerryScript` 引擎的回归测试套件中获取到的种子用例²³；而用例(c)是将种子用例根据语义信息经过定向突变后产生的新用例，该用例触发了 `ChakraCore` 引擎（1.11.19 版本）在 `TypedArray.prototype.sort` 方法的实现上缺失了强制类型转换步骤的缺陷。

`TypedArray.prototype.sort` 是用来将一个 `TypedArray` 数组对象进行排序的方法，其接收一个参数 `comparefn`，表示排序的具体实现方法。`ESfunfuzz` 通过对 ES 标准进行分析得到了 `comparefn` 可能会有特殊值“`undefined`”的信息，利用这一信息将种子用例(b)突变为用例(c)并成功地检测出了 `ChakraCore` 引擎的缺陷，此用例很好地说明了基于标准文档分析的定向突变方法的有效性。

²³ <https://github.com/jerryscript-project/jerry/blob/8edf8d6eea4327dd83b7fabddcae4ea23bf98fb9/tests/jerry/es.next/regression-test-issue-3975.js>

5.6.3 案例研究（三）——非预期类型变异生成方法的有效性验证

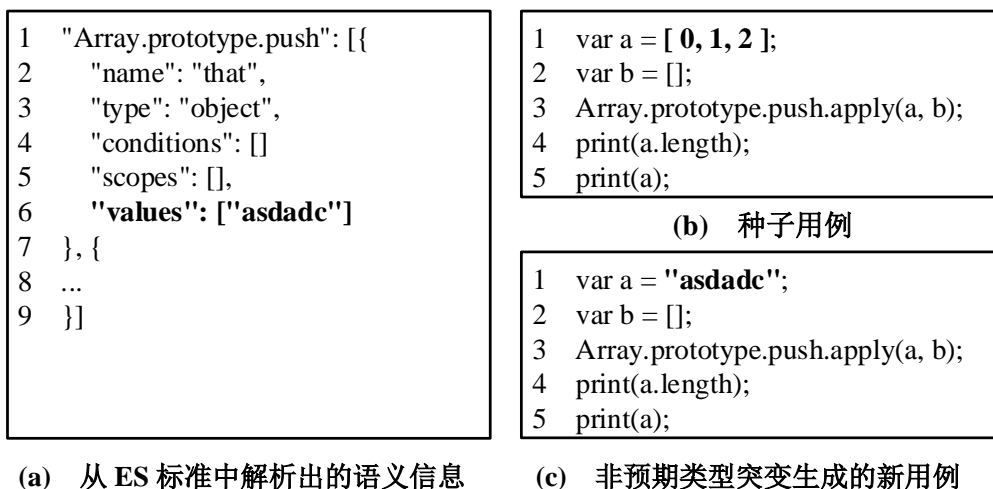


图 31 非预期类型突变生成方式产生的用例实例^[58]

图 31 是证明本文提出的非预期类型变异方法有效性的典型案例，31(a)是从 ES 标准中解析出的关于 `Array.prototype.push` 方法的相关语义信息；用例(b)是原始的种子用例；用例(c)是将种子用例根据语义信息经过非预期类型突变后产生的新用例，该用例触发了 JavaScriptCore 引擎（d940b47 版本）在该方法的实现上缺乏将传入参数强制转化为对象的步骤的缺陷。

`Array.prototype.push` 是 JavaScript 语法中用来向一个数组对象中添加元素的方法，从图 31(a)中可以看出，从 ES 标准中并没有提取出关于该方法的任何条件语句（即 `conditions` 字段都为空），因此获取不到该方法传入参数的候选值，也就难以为其定向突变提供指导。在语义信息指导方法失效的情况下，参数类型推断和非预期类型变异方法发挥了作用：根据 ES 标准，参数类型推断方法准确地推断出了 `push` 方法的第一个参数类型是 `Object`（即图 31(a)第 3 行），因此，非预期类型变异方法将从 `Object` 以外的几种类型中随机选择并生成一个值。此处正好选中了字符串生成器，并生成了一个随机字符串“asdadc”，添加到了候选值列表中（即图 31(a)第 6 行）。根据这唯一的候选值，将用例(a)定向突变成了新用例(c)，从而成功地检测出了 JavaScriptCore 引擎的缺陷。此用例有力地说明了非预期类型变异方法的有效性。

5.7 本章小结

本章主要对两部分内容进行了介绍，其一是对基于本文提出方法的缺陷检测系统原型 ESfunfuzz 的设计与实现；其二是对 ESfunfuzz 以及其他主流的模糊测试工具的对比实验。实验部分是本章的重点，主要包含三个方面：首先为了使 ESfunfuzz 原型

系统的性能达到最优，进行了语言模型的性能调优实验，通过该组实验确定了此场景下最优的数据处理形式和参数配置；其次是 ESfunfuzz 与其他工具的对比实验，分别进行了生成用例质量的对比以及缺陷检测能力的对比，通过此实验说明了本文提出的缺陷检测方法拥有较优秀的缺陷检测能力；最后是真实环境下的缺陷检测实验，使用 ESfunfuzz 对四个主流引擎进行了实际的测试，有效地检测出了 22 个引擎缺陷，并通过对典型案例进行研究分析，探明了本文方法的生效机制，进一步地说明了基于标准文档分析的缺陷检测方法的有效性。

总结与展望

总结

JavaScript 是一种流行的、平台无关的编程语言，为了保证 JavaScript 程序在不同平台上的互操作性，JavaScript 引擎的实现必须符合 ES 标准。然而，标准的频繁变动使得引擎开发者往往难以做出及时的更新，从而使 JavaScript 引擎出现不符合标准的行为，即一致性缺陷。为了有效地对一致性缺陷进行检测，本文提出了一种基于标准文档分析和差分模糊测试的自动化缺陷检测方法。

本文主要包含以下几个方面的研究工作：

(1) 本文通过对 ES 标准文档进行研究分析，发现其对于 JavaScript 引擎的一致性缺陷检测具有重要指导作用，从而创新地设计了一种基于标准文档分析的测试用例突变方法。该方法能够从标准文档中自动地解析出有效的语义信息，来指导测试用例进行定向地突变，突变后的用例能够精准地覆盖到引擎的更多分支，从而能够触发更多的引擎一致性缺陷，提高了检测效率。

(2) 本文对软件测试领域的常见技术（尤其是差分测试与模糊测试机制）进行了研究，分析其优势与不足，最终设计了一种基于差分模糊测试机制的自动化测试方法。该方法基于自动生成的测试用例，能够持续地对 JavaScript 引擎进行差分模糊测试，以检测引擎的一致性缺陷。同时，本文对现有的测试用例生成方法进行了广泛地研究，对于突变式和生成式这两种主流的用例生成方法进行了研究分析，最终设计并实现了两种有效的用例产生方法：基于神经语言模型的测试用例续写生成方法和基于标准文档分析的用例突变生成方法，使用这两种方法能够产生有效触发引擎缺陷的测试用例。

(3) 本文依据此方法设计并实现了一个缺陷检测原型系统 ESfunfuzz，并使用它对当前四个主流的 JavaScript 引擎进行了长期的真实环境下的缺陷检测，最终检测出各类引擎的一致性缺陷共 22 个，其中 19 个已被引擎开发商确认，这说明了本文提出的方法是确实可行的。同时，本文还将 ESfunfuzz 与如今最先进的几种模糊测试工具进行了对比实验，进一步说明了本文方法的实用性和有效性。

展望

本文所提出的基于标准文档分析的 JavaScript 引擎缺陷检测方法能够有效地对 JavaScript 引擎中的一致性缺陷进行检测，并已经在真实环境下的验证了方法的有效性，取得了不错的结果。然而，本方法仍然存在一些可以改进的地方，主要包含以下几个方面：

(1) 本文采用差分模糊测试机制来对 JavaScript 引擎的一致性缺陷进行检测。然而，差分测试这一机制本身便存在一个固有缺陷——在差分测试过程中，只有当多组执行结果中呈现出不一致的行为时，该用例才会被注意到。然而，假设对于某个 JavaScript 的语言特性，所有的待测引擎全都实现错误，它们的执行结果会呈现出错误地一致。由于差分测试机制无从得知执行结果的正确与否，只能识别多个结果是否一致，所以这个一致的结果会导致该用例被识别为通过，从而遗漏掉该缺陷。尽管所有引擎同时实现错误的可能性极小，但这也是客观存在的问题，因此后续需要考虑如何对这类问题进行检测。

(2) 本文使用了从 ECMAScript-262 标准文档中解析出的语义信息来指导测试用例的突变，能够有效地提升用例触发引擎缺陷的能力。然而，当前的标准到代码的转化算法是通过人工定义的正则表达式规则来实现的，这种方法虽然实现原理简单，但需要较多的人工介入，并且由于规则都是针对 JavaScript 语言所定制的，所以普适性较差，难以迁移到别的语言上。目前，国内外已有相关研究工作通过深度学习方法实现了从自然语言到代码的转换任务，因此，笔者后续也准备尝试借助深度学习方法来对这一点做出相应的改进。

(3) 本文所提出的缺陷检测方法在最终阶段，即对可疑用例的分析和确认步骤上仍然需要人工参与，人工分析虽然准确但毫无疑问地会拖慢整体的测试效率，增加测试成本。因此，假如通过某种方法能够实现自动地对可疑用例进行分析和判断，那么就能实现整个检测方法完全自动化，形成闭环，这样无疑会给整体的测试过程带来巨大的效率提升。后续考虑在测试用例自动化分析上做出优化。

参考文献

- [1] JavaScript Wikipedia[DB/OL]. <https://en.wikipedia.org/wiki/JavaScript>
- [2] ECMA-262 Specification[EB/OL].
<https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>
- [3] ECMA-262 5.1 Object.keys[EB/OL]. <https://262.ecma-international.org/5.1/#sec-15.2.3.14>
- [4] ECMA-262 6.0 Object.keys[EB/OL]. <https://262.ecma-international.org/6.0/#sec-object.keys>
- [5] JavaScriptCore Bug Report1[DB/OL]. https://bugs.webkit.org/show_bug.cgi?id=200190
- [6] ChakraCore Bug Report1[DB/OL]. <https://github.com/chakra-core/ChakraCore/issues/6553>
- [7] Test-262 TestSuite[DB/OL]. <https://github.com/tc39/test262>
- [8] ChakraCore Bug Statistics[DB/OL]. <https://github.com/chakra-core/ChakraCore/issues>
- [9] jsfunfuzz[DB/OL]. <https://github.com/MozillaSecurity/funfuzz>
- [10] Holler C, Herzig K, Zeller A. Fuzzing with code fragments[C]. 21st USENIX Security Symposium. 2012: 445-458.
- [11] Veggalam S, Rawat S, Haller I, et al. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming[C]. European Symposium on Research in Computer Security. Springer, Cham, 2016: 581-601.
- [12] Samuel Groß. FuzzIL: Coverage guided fuzzing for JavaScript engines[D]. Master's thesis, Karlsruhe Institute of Technology. 2018.
- [13] Han H S, Oh D H, Cha S K. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines[C]. NDSS. 2019.
- [14] Dinh S T, Cho H, Martin K, et al. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases[J]. 2021.
- [15] 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.
- [16] Park S, Xu W, Yun I, et al. Fuzzing JavaScript Engines with Aspect-preserving Mutation[C]. 2020 IEEE Symposium on Security and Privacy (SP). IEEE, 2020: 1629-1642.
- [17] 万东. 跨平台解释器的研究与设计[J]. 电子设计工程, 2015, 23(12): 8-10.
- [18] Client Language Statistics[EB/OL].
https://w3techs.com/technologies/history_overview/client_side_language/all
- [19] JavaScript Engine Wikipedia[DB/OL]. https://en.wikipedia.org/wiki/JavaScript_engine
- [20] Browser Statistics[EB/OL]. <https://gs.statcounter.com/browser-market-share/desktop/worldwide>
- [21] JavaScriptCore Engine[DB/OL]. <https://github.com/WebKit/WebKit>
- [22] ChakraCore Engine[DB/OL]. <https://github.com/chakra-core/ChakraCore>
- [23] V8 Engine[EB/OL]. <https://v8.dev/>
- [24] SpiderMonkey Engine[DB/OL]. <https://mozilla-spidermonkey.github.io/>
- [25] NodeJS[EB/OL]. <https://nodejs.org/en/>

-
- [26] Hermes Engine[EB/OL]. <https://hermesengine.dev/>
- [27] JerryScript Engine[DB/OL]. <https://github.com/jerryscript-project/jerryscript>
- [28] Rhino Engine[DB/OL]. <https://github.com/mozilla/rhino>
- [29] Nashorn Engine[EB/OL]. <http://openjdk.java.net/projects/nashorn/>
- [30] GraalJS Engine[DB/OL]. <https://github.com/oracle/graaljs>
- [31] JerryScript Bug Report1[DB/OL]. <https://github.com/jerryscript-project/jerryscript/issues/3325>
- [32] JerryScript Bug Report2[DB/OL]. <https://github.com/jerryscript-project/jerryscript/issues/3380>
- [33] Rhino Bug Report1[DB/OL]. <https://github.com/mozilla/rhino/issues/587>
- [34] JerryScript Bug Report3[DB/OL]. <https://github.com/jerryscript-project/jerryscript/issues/3229>
- [35] 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.
- [36] Dharma[DB/OL]. <https://github.com/MozillaSecurity/dharma>
- [37] Godefroid P. Random testing for security: blackbox vs. whitebox fuzzing[C]. Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE International Conference on Automated Software Engineering (ASE 2007). 2007: 1-1.
- [38] Ganesh V, Leek T, Rinard M. Taint-based directed whitebox fuzzing[C]. 2009 IEEE 31st International Conference on Software Engineering. IEEE, 2009: 474-484.
- [39] Manès V J M, Han H S, Han C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2019.
- [40] DeMott J, Enbody R, Punch W F. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing[J]. BlackHat and Defcon, 2007.
- [41] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-aware Evolutionary Fuzzing[C]. NDSS. 2017, 17: 1-14.
- [42] McKeeman W M. Differential testing for software[J]. Digital Technical Journal, 1998, 10(1): 100-107.
- [43] Song F, Croft W B. A general language model for information retrieval[C]. Proceedings of the eighth international conference on Information and knowledge management. 1999: 316-321.
- [44] Katz S. Estimation of probabilities from sparse data for the language model component of a speech recognizer[J]. IEEE transactions on acoustics, speech, and signal processing, 1987, 35(3): 400-401.
- [45] 蒋学. 基于 UI 的应用异常行为分析的研究与实现[D]. 北京邮电大学, 2018.
- [46] Bengio Y, Ducharme R, Vincent P, et al. A neural probabilistic language model[J]. The journal of machine learning research, 2003, 3: 1137-1155.
- [47] Mikolov T, Karafiát M, Burget L, et al. Recurrent neural network based language model[C]. Eleventh annual conference of the international speech communication association. 2010.
- [48] 徐萍, 吴超, 胡峰俊等. 基于迁移学习的个性化循环神经网络语言模型[J]. 南京理工大学学报, 2018, 42(04): 401-408.
- [49] Sundermeyer M, Schlüter R, Ney H. LSTM neural networks for language modeling[C]. Thirteenth annual conference of the international speech communication association. 2012.

- [50] AFL[EB/OL]. <http://lcamtuf.coredump.cx/afl/>
- [51] 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.
- [52] Linux Signal[EB/OL]. <https://man7.org/linux/man-pages/man7/signal.7.html>
- [53] Sennrich R, Haddow B, Birch A. Neural Machine Translation of Rare Words with Subword Units[C]. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics. 2016: 1715-1725.
- [54] Klees G, Ruef A, Cooper B, et al. Evaluating fuzz testing[C]. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018: 2123-2138.
- [55] Case Study 1[DB/OL]. https://bugs.webkit.org/show_bug.cgi?id=221177
- [56] MDN Object.freeze[EB/OL].
https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze
- [57] Case Study 2[DB/OL]. <https://github.com/chakra-core/ChakraCore/issues/6503>
- [58] Case Study 3[DB/OL]. https://bugs.webkit.org/show_bug.cgi?id=201910

攻读硕士学位期间取得的科研成果

1. 申请（授权）专利

- [1] 汤战勇, 田洋, 弋雯, 瞿兴等. 一种基于种子用例突变的反馈式 JS 引擎模糊测试方法及装置: 中国, 202110308117.6[P]. 2021-03-23. (受理)
- [2] 房鼎益, 曹帅, 叶贵鑫, 田洋等. 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]. 2020-12-09. (受理)
- [3] 叶贵鑫, 弋雯, 王焕廷, 田洋等. 一种基于标准文档分析的 JS 引擎模糊测试方法: 中国, 202011450408.0 [P]. 2020-03-20. (受理)

2. 参与科研项目及科研获奖

- [1] 国家自然科学基金项目, 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, 项目号 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, 项目号 2020KWZ-013.
- [3] 陕西省国际合作项目, 无线感知视域下以书法为代表的非物质文化遗产活化、传承与创新研究, 项目号 2021KW-15.
- [4] 陕西省国际合作项目, 面向漏洞检测技术的软件源代码特征表征关键技术研究, 项目号 2021KW-04.

致谢

时光荏苒，转眼间我已经在西北大学度过了七个年头。在这段人生最宝贵的时光里，我经历了各种各样的事，结识了各种各样的人，最终从一个懵懂无知的青涩少年成长为了即将步入职场、即将对国家和社会真正做出贡献的青年人。借此机会，我想向一路上伴我同行的人们表达感谢。

首先，我要感谢我的导师汤战勇教授，和您第一次相遇是在学校的院系宣讲会上，您幽默的口才和真诚的态度打动了我，后来我也选择加入了由您主导的网络安全组，事实证明我的选择没错，在这里我度过了非常紧张又充实的三年半的生活。每一次会议，您的意见总是那么一针见血，让我受益良多，感谢您对我的严格要求使我成为了一个更加优秀的人。其次，我要感谢房鼎益教授，感谢您对安全组实验室的支持，为我们提供了如此优秀的科研环境，无论是亲自替我们把关论文还是坚持在讲台上授课，您认真负责的工作态度都令我折服。之后，我要感谢实验室的学长兼讲师叶贵鑫博士，我从进入实验室起就一直在您的小组里参与科研工作，一路上遇到了许多困难的问题，您都能一一解答，并为我指出之后的科研方向。在您的带领下，组里的研究工作不断再创新高，感谢您悉心的指导。我还要感谢与我一同奋战过的组员瞿兴、弋雯、王媛、李豪斌、李小伟等，感谢你们为本工作提供的支持与帮助；感谢已经毕业的曹帅学长，感谢你在我求职的路上为我提供的指导和帮助；我还要感谢范天赐、范子茜、冯晖、李朋、贺怡、王焕廷、薛永康、姚厚友、张梦欢、张宇翔等十位同学，你们的陪伴和鼓励让我在实验室的时光变得更加丰富多彩，与你们相识是我最宝贵的财富。随后我还要感谢我的室友权大玮和李阳阳，感谢你们的陪伴让我在学习之余能够放松身心，养好精神。我还要感谢我的女朋友穆丽，在与你相遇相知的八年时光里，你的支持和鼓励是对我而言最好的安慰剂，感谢你对我生活上的帮助以及情感上的陪伴。最后我要感谢我的父母，感谢你们对我十年如一日的养育和支持，让我可以没有后顾之忧地完成学业。

最后，感谢国家自然科学基金 61972314 号项目、陕西省重点研发计划 2020KWZ-013 以及陕西省国际合作项目 2021KW-15 和 2021KW-04 对本文工作的支持。祝愿西北大学越来越好，祝愿几位老师身体健康，祝愿实验室即将毕业的大家未来工作顺利，前程似锦！