面向JavaScript引擎报错机制的类别导向模糊测试方法

卢凌, 周志德, 任志磊, 江贺

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.221200166

面向 JavaScript 引擎报错机制的类别导向模糊测试方法

卢　凌[1]　周志德[1]　任志磊[1,2]　江　贺[1]
1 大连理工大学软件学院　辽宁 大连 116620
2 南京航空航天大学高安全系统的软件开发与验证技术工业和信息化部重点实验室　南京 210016
(lling@mail.dlut.edu.cn)

摘　要　报错机制是 JavaScript 引擎必不可少的一部分。面对错误的程序,JavaScript 引擎报错机制应输出合理的错误信息,指出错误的原因和位置,帮助开发人员修复错误。然而,JavaScript 引擎报错机制中存在会阻碍开发人员修复错误的缺陷。文中提出了首个面向 JavaScript 引擎报错机制的类别导向模糊测试方法 CAFJER。给定一个种子程序,CAFJER 首先为其选择一个目标类别的错误信息,并进行动态分析得到其上下文信息。其次,CAFJER 根据种子程序的上下文信息生成能触发目标类别错误信息的测试用例。然后,CAFJER 将生成的测试用例输入不同 JavaScript 引擎中进行差分测试。若输出的错误信息间有所差异,则说明其中可能存在缺陷。最后,CAFJER 自动过滤重复的和无效的测试用例,有效减少了人工的参与。为了验证 CAFJER 的有效性,将 CAFJER 与目前先进的相似方法 JEST 和 DIPROM 进行比较,实验结果表明,CAFJER 在 JavaScript 引擎报错机制中发现的独特缺陷数分别是 JEST 和 DIPROM 的 2.17 倍和 26 倍。在为期 3 个月的实验中,CAFJER 还向开发者提交了 17 个缺陷报告,其中 7 个已被确认。

关键词:JavaScript;报错机制;错误信息;差分测试;程序变异
中图法分类号　TP311

Category-directed Fuzzing Test Method for Error Reporting Mechanism in JavaScript Engines

LU Ling[1],ZHOU Zhide[1],REN Zhilei[1,2] and JIANG He[1]
1 School of Software Engineering,Dalian University of Technology,Dalian,Liaoning 116620,China
2 Key Laboratory of Safety-Critical Software Ministry of Industry and Information Technology,Nanjing University of Aeronautics and Astronautics,Nanjing 210016,China

Abstract　Error reporting mechanism is an indispensable part of JavaScript engines. For programs with errors,the error reporting mechanism of JavaScript engines should output reasonable error message,point out location and cause of the error,help developers to repair the program. However,there are defects in the JavaScript engine error reporting mechanism that will prevent developers from repairing errors. In this paper,the first category directed fuzzy testing method for JavaScript engine error reporting mechanism called CAFJER is proposed. For a given seed program,CAFJER first selects an error message of the target category for it and dynamically analyzes it to obtain its context information. Secondly,CAFJER generates test cases that can trigger target category error information according to the context information of the seed program. Thirdly,CAFJER inputs the generated test cases into different JavaScript engines for differential testing. If there are differences between error messages thrown by JavaScript engines,it indicates that there may be a defect. Finally,CAFJER automatically filters repeated and invalid test cases,effectively reducing manual participation. In order to verify the effectiveness of CAFJER,it is compared with the current advanced similar methods JEST and DIPROM. Experimental results show that the unique defects found by CAFJER in the JavaScript engine error reporting mechanism is 2.17 times and 26.00 times that of JEST and DIPROM respectively. During the three-month experiment,CAFJER also submitted 17 defect reports to developers and 7 of which have been confirmed.

Keywords　JavaScript,Error reporting mechanism,Error message,Differential test,Program mutation

1　引言

JavaScript 是世界上最受欢迎的编程语言之一,被广泛应用于客户端编程和服务器端编程[1]。GitHub 年度报告显示,JavaScript 从 2014 年开始成为 GitHub 上最受欢迎的编程语言并持续至今[2]。JavaScript 语言开发过程中不可避免地会出现错误的 JavaScript 程序。面对错误的程序,理想情况下,JavaScript 引擎报错机制应输出合理的错误信息,指出错误的原因和位置,帮助开发人员修复程序。但 JavaScript 引擎报错机制中存在缺陷,输出的错误信息可能会指向未出错的代码位置或给出令人疑惑的解释说明。例如,编译代码“var enum;”时,JavaScript 引擎应提示开发人员不能将关键词

到稿日期:2022-12-28　返修日期:2023-03-30
通信作者:江贺(jianghe@dlut.edu.cn)

"enum"声明为变量名(SyntaxError: Cannot use the keyword "enum" as a variable name)。但此时 JavaScript 引擎 SpiderMonkey[3]认为代码出错的原因是缺少变量名(SyntaxError: missing variable name)[4]。这一错误的提示会使开发人员陷入迷茫,阻碍开发进展。

现有大量有关 JavaScript 的研究[5-13],它们致力于生成正确的测试用例,覆盖尽可能多的语法规则,对 JavaScript 引擎编译机制进行检测。但差分测试时它们抛弃了所有错误的测试用例,忽视了 JavaScript 引擎报错机制的重要性。另外,各 JavaScript 引擎报错机制输出的错误信息由开发人员按照个人的编程经验编写,没有统一的格式。因此,对 JavaScript 引擎报错机制的检测主要面临以下两个挑战:

挑战 1 如何使测试用例覆盖尽可能多的错误类别?面对不同错误程序,JavaScript 引擎报错机制会输出各类错误信息。为实现对报错机制的全面检测,我们需要生成能触发各类错误信息的测试用例。但 JavaScript 官方文档 ES12[14]和各 JavaScript 引擎的说明文档都忽视了错误信息的分类问题,只有 JavaScript 社区尝试对错误信息进行分类。但社区文档没能发现所有的错误类别,也没能明确给出各类错误的触发条件。

挑战 2 如何辨别错误信息间的差异?我们可以将同一错误程序输入不同 JavaScript 引擎,比较报错机制输出的错误信息,若其中存在差异,则说明可能发现了一个报错机制中的缺陷。检测 JavaScript 引擎编译机制时,我们可以直接比较编译机制的输出结果是否完全一致。但错误信息由 JavaScript 引擎开发人员按照个人理解编写,没有统一的规范。面对同一错误程序,各 JavaScript 引擎会抛出表达格式不同的错误信息,我们难以直接比较输出的错误信息间是否存在差异。

为解决上述挑战,本文提出了 CAFJER(CAtegory-directed Fuzzing for JavaScript ERror message),一种类别导向的模糊测试方法。我们从 JavaScript 社区的文档和差分测试输出的错误信息中总结得到了共 61 类错误信息。CAFJER 依据各类错误信息的触发条件实现了增改 API、插入函数模板以及更改符号 3 种类别导向的变异方法。这些方法生成的测试用例能触发选定的错误信息类别,实现了对全部 61 类错误信息的覆盖,从而解决了挑战 1。此外,我们还发现各 JavaScript 引擎的各类错误信息都有独特的表达格式,并为每个 JavaScript 引擎的每类错误信息都设计了对应的正则表达式。正则表达式中固定的普通字符对错误信息中独特的表达格式进行匹配,从而识别错误信息对应的类别。正则表达式中的元字符对应可变的错误位置,提取出报错机制抛出的错误位置。利用正则表达式,CAFJER 将错误信息简化为对应的错误类别和错误位置。差分测试中,CAFJER 直接比较错误信息的错误类别和错误位置是否完全相同,若其中存在差异,则说明可能发现了一个缺陷,从而解决了挑战 2。

我们将 CAFJER 部署在 3 个 JavaScript 引擎 V8[15]、JSC[16]和 SpiderMonkey[3]上。为验证 CAFJER 的有效性,将 CAFJER 与目前先进的相似方法 JEST[10]和 DIPROM[17]进行对比。在长达 24 h 的实验中,CAFJER、JEST、DIPROM 分别发现了 5.2 个、2.4 个、0.2 个独特的缺陷,CAFJER 能更有效地发现 JavaScript 引擎报错机制中更多的缺陷。在长达 3 个月的实验中,我们还向开发者提交了 17 个缺陷报告,其中 7 个已被确认。

本文的主要贡献如下:

1)将各 JavaScript 引擎报错机制输出的错误信息分为 61 类,并总结了各类错误信息的触发条件。

2)提出了一种类别导向的 JavaScript 程序变异方法,它生成的测试用例能触发选定类别的错误信息,实现了对全部 61 类错误信息的覆盖。与 JEST 和 DIPROM 相比,CAFJER 的变异方法能更有效地发现 JavaScript 引擎报错机制中更多的缺陷。

3)编写了 JavaScript 引擎报错机制缺陷的测试方法 CAFJER,对真实世界中 JavaScript 引擎的报错机制进行了检测。在长达 3 个月的实验中,我们共向开发者提交了 17 个缺陷报告,其中 7 个已被确认。

2 相关工作

2.1 模糊测试方法

CAFJER 采用模糊测试的方法对 JavaScript 引擎报错机制缺陷进行检测。模糊测试是 JavaScript 引擎缺陷检测的常用方法,是一种自动化的错误检测技术[18-21],能快速生成大量测试用例,也常被用于编译器缺陷的检测工作中[22-24]。模糊测试方法有两大类,分别是基于变异的模糊测试方法和基于生成的模糊测试方法。

1)基于变异的模糊测试方法

基于变异的模糊测试方法通过修改种子程序生成测试用例。AFL[25]是最著名的基于变异的模糊测试方法,实现了多种覆盖率导向的变异策略。在 JavaScript 引擎缺陷检测领域,Superion[5]、Cerebro[6]和 DIE[7]拓展了 AFL 的研究。Superion 利用基于语法感知的变异策略对种子程序的抽象语法树进行修剪和变异操作,提高了测试用例的正确率;Cerebro 依据有效性对种子程序进行排序,优先选择发现缺陷潜力大的种子程序;DIE 变异时保留了抽象语法树各结点的参数类型和种子程序中的函数结构,使测试效率得到进一步提高。除此之外,IFuzzer[8]使用遗传算法优化种子程序抽象语法树的突变策略,SoFi[9]利用动态反射技术挖掘种子程序中的函数信息。但现有的模糊测试方法都抛弃了生成的错误测试用例,本文提出的 CAFJER 也是一种基于变异的模糊测试方法。给定一个种子程序和目标类别的错误信息,CAFJER 利用种子程序的上下文信息生成错误代码,使输出的测试用例能触发目标类别的错误信息。

2)基于生成的模糊测试方法

基于生成的模糊测试方法依据开发人员手动编写的上下文无关语法,从零开始生成测试用例。现有研究工作中,JEST[10]利用信息检索技术解析 JavaScript 语法规则,依此生成测试用例。CodeAlchemist[11]将 JavaScript 程序拆分成代码块,分析各代码块的上下文需求,利用代码块拼出测试用例。Montage[12]利用 JavaScript 程序训练神经网络,使神经网络输出测试用例。COMFORT[13]解析 ES12 中关于 API 和

对象的定义规则，生成 API 和对象的调用语句。

2.2　编译器警告信息缺陷的检测工作

编译器警告信息缺陷的检测工作与 JavaScript 引擎报错机制缺陷的检测工作极为相似。Epiphron[26] 是首个编译器警告信息缺陷的检测方法，它利用 C 语言语法随机生成测试用例，筛选出其中能触发编译器警告信息的测试用例，并将它们输入不同编译器中进行差分测试。若各编译器输出的警告信息间存在差异，则说明可能发现了缺陷。DIPROM[17] 在变异方法方面取得了突破，依据 C 语言的特性，设计了多种对种子程序进行修剪和插入操作的变异策略，提高了测试的有效性。但这两种方法都不能确保生成的测试用例会触发警告信息。在 JavaScript 引擎报错机制缺陷检测领域，CAFJER 依据各类错误信息的触发条件设计变异方法，确保生成的测试用例能使 JavaScript 引擎抛出目标类别的错误信息。

3　JavaScript 引擎报错机制

本章将介绍有关 JavaScript 引擎报错机制的信息，包括 JavaScript 引擎报错机制缺陷的危害，JavaScript 错误信息的构成，JavaScript 错误信息的分类以及 JavaScript 报错机制缺陷的分类。

3.1　JavaScript 引擎报错机制缺陷的危害

JavaScript 语言受到了开发人员的青睐。开发者调查分析公司 SlashData 的统计数据显示，全球有超过 174 万活跃的 JavaScript 开发人员[27]。GitHub 的年度报告表明，JavaScript 连续 8 年都是 GitHub 上最受欢迎的编程语言。

JavaScript 开发人员难免会写出错误的程序。面对编译失败的 JavaScript 程序，开发人员急需 JavaScript 引擎报错机制的帮助。理想情况下，JavaScript 引擎报错机制能处理各类异常的输入，抛出合适的错误信息，指出错误的原因和位置，帮助开发人员解决问题。但存在缺陷的报错机制可能会抛出异常的错误信息，例如指向未发生错误的位置或给出令人困惑的解释说明。报错机制中的缺陷会阻碍错误的修复，从而耗费开发人员更多的时间和精力。

同时，一款有竞争力的 JavaScript 引擎也必须具备强健壮性，充分考虑各种异常情况，妥善处理各类意料之外的输入。报错机制中的缺陷会影响开发人员对 JavaScript 引擎的评价。面对错误的 JavaScript 程序，JavaScript 引擎报错机制应该抛出合适的错误信息以帮助开发人员修复程序，不能崩溃闪退或超时无回应，也不能仅仅提示编译错误。

3.2　JavaScript 错误信息的构成

JavaScript 引擎报错机制抛出的错误信息由 3 部分构成，分别是错误类型、错误说明和错误位置。

1）错误类型

错误类型是错误信息中最先输出的部分。错误类型共有 7 类，分别是 RangeError，ReferenceError，SyntaxError，TypeError，URI-Error，InternalError 和 EvalError。错误类型会告知开发人员产生错误的根本原因，例如类型不匹配或超出范围等。ES12 详细描述了这 7 种错误类型的触发条件，下面是各错误类型的介绍。

（1）RangeError 表示被调用的值不在允许的范围内。例如，把数组的长度设置成负数时，JavaScript 引擎报错机制就会抛出 RangeError。

（2）ReferenceError 表示检测到一个无效的引用。例如，调用一个未被初始化的变量时，JavaScript 引擎报错机制就会抛出 ReferenceError。

（3）SyntaxError 表示 JavaScript 引擎解析程序失败。被破坏的程序结构会触发 SyntaxError。例如，程序中只有左括号而没有右括号，出现了括号不匹配的错误时，JavaScript 引擎报错机制就会抛出 SyntaxError。

（4）TypeError 表示类型错误，输入不符合预期的数据类型或调用无效的方法会触发 TypeError。例如，试图删除只读变量时，JavaScript 引擎报错机制就会抛出 TypeError。

（5）URIError 表示全局 URI 处理函数的使用与定义相违背。例如，URI 的编码和解码出现问题时，JavaScript 引擎报错机制就会抛出 URIError。

（6）InternalError 表示内部错误，JavaScript 引擎的工作负载突然激增达到上限时会触发 InternalError。例如，编译死循环代码时，JavaScript 引擎报错机制就会抛出 InternalError。

（7）EvalError 表示发生了异常的 eval()函数调用。ES12 中，EvalError 已不再被拓展，只对以前的版本进行了兼容。

2）错误说明

错误说明是对错误的几句描述，用于解释错误产生的原因，给出了解决错误的建议，一部分错误说明中也包含了错误位置。JavaScript 引擎报错机制在错误类型后输出错误说明。ES12 没有对错误说明进行任何规定。错误说明由各 JavaScript引擎开发人员按照个人理解编写。面对同一错误程序，不同 JavaScript 引擎会输出不同的错误说明。

错误说明是错误信息分类的主要依据。同一 JavaScript 引擎中，我们认为同类的错误说明拥有相同的错误原因，对应相同的错误类型，有至少半数以上相同的连续字符。不同 JavaScript 引擎中，我们只根据错误原因对错误说明进行归类。依据错误说明，我们总结得到了 61 类错误信息。

3）错误位置

错误位置由发生错误的代码行号以及引发错误的代码片段组成，一般在错误类别和错误说明的下一行输出。根据 JavaScript 引擎和错误类别的不同，错误位置的详细程度不一。详细的错误位置会精确指向引发错误的 API、符号或函数调用，粗略的错误位置则会输出引发错误的一整行代码。错误位置也是检测 JavaScript 引擎报错机制缺陷的有效手段。对于同一个测试用例，如果各 JavaScript 引擎抛出的错误位置有所差异，那么其中可能存在缺陷。

3.3　JavaScript 错误信息的分类

为实现类别导向的模糊测试方法，首先需要对报错机制输出的错误信息进行分类。

JavaScript 官方文档 ES12 和各 JavaScript 引擎的说明文档都忽视了错误信息的分类问题，只有 JavaScript 社区尝试对错误信息进行分类。但社区不能保证分类的正确性，社区文档记录的各类错误信息中存在错误。我们在最新版本的 JavaScript 引擎中尝试复现社区文档中的各类错误信息。

社区文档共记载了 73 类错误信息，其中 18 类不能在 JavaScript 引擎中成功复现。从社区文档中，我们共得到了 55 类有效的错误信息。

同时，社区文档的分类中存在缺漏。为了寻找缺失的错误信息类别，我们先为已发现的各类错误信息构建对应的正则表达式。同一 JavaScript 引擎的同类错误信息中的错误类型和绝大部分错误说明始终固定不变，错误位置会随着输入程序产生变化。为特定类别错误信息构建正则表达式时，我们用正则表达式中固定的普通字符匹配错误信息中的错误类型和错误说明的不变字段，用正则表达式的元字符匹配错误说明的灵活字段和错误位置。不同 JavaScript 引擎间，同类错误信息中错误说明的固定字段也有差异。我们为每个 JavaScript 引擎的每类错误信息都构建了独特的正则表达式，它们能且仅能匹配对应 JavaScript 引擎的对应错误信息类别。差分测试中，我们利用正则表达式对输出的错误信息进行匹配，如果匹配失败，则说明可能发现了新的错误信息类别。依据错误原因是否相同、错误类型是否相同以及错误说明中是否存在半数以上相同的连续字符这 3 个指标，我们将新发现的错误信息和现有的各类错误信息进行比较。如果现有的某类错误信息与新发现的错误信息之间能满足以上至少两个指标，就将新发现的错误信息作为它的补充，否则将新发现的错误信息作为新的错误信息类别进行研究。差分测试中，我们共发现了 6 类新的错误信息。

通过研究，我们在社区文档的记录和差分测试输出的错误信息中共总结得到了 61 类错误信息。

3.4 JavaScript 报错机制缺陷的分类

JavaScript 引擎报错机制的缺陷可分为两类，分别是错误类别不正确和错误位置不正确。

错误类别不正确指报错机制输出的错误信息类别与真正需要的错误信息类别不一致，输出的错误信息对错误产生原因的描述有误。开发人员遇到错误类别不正确的错误信息时，会轻信 JavaScript 引擎报错机制抛出的错误信息，先入为主地误解错误产生的原因，难以快速修复错误，使开发工作陷入僵局。差分测试中，如果同一测试用例使不同 JavaScript 引擎抛出不同类别的错误信息，即不同 JavaScript 引擎认为引发错误的原因不同，那么很有可能发现了错误类别不正确的缺陷。

错误位置不正确指报错机制输出的错误位置与真正引发错误的代码位置不一致，输出的错误位置指向了未引发错误的代码行号或语句。开发人员遇到错误位置不正确的错误信息时，会浪费大量的时间精力研究正确的代码。差分测试中，如果同一测试用例使不同 JavaScript 引擎抛出不同的错误位置，那么有可能发现了错误位置不正确的缺陷。

4 CAFJER

4.1 总体流程

CAFJER 利用类别导向的模糊测试方法实现对各 JavaScript引擎报错机制缺陷的检测。CAFJER 的总体框架如图 1 所示。给定一个种子程序，CAFJER 首先对其进行动态分析，得到种子程序的上下文信息，并为种子程序选择一个目标类别错误的信息。其次，CAFJER 将带上下文信息的种子程序和目标类别错误信息输入变异器，生成能触发 JavaScript 引擎目标类别错误信息的测试用例。然后，CAFJER 将生成的测试用例输入不同 JavaScript 引擎中进行差分测试，对比抛出的错误信息，判断其中是否存在差异。最后，CAFJER 过滤掉重复的和无效的测试用例，输出缺陷报告。

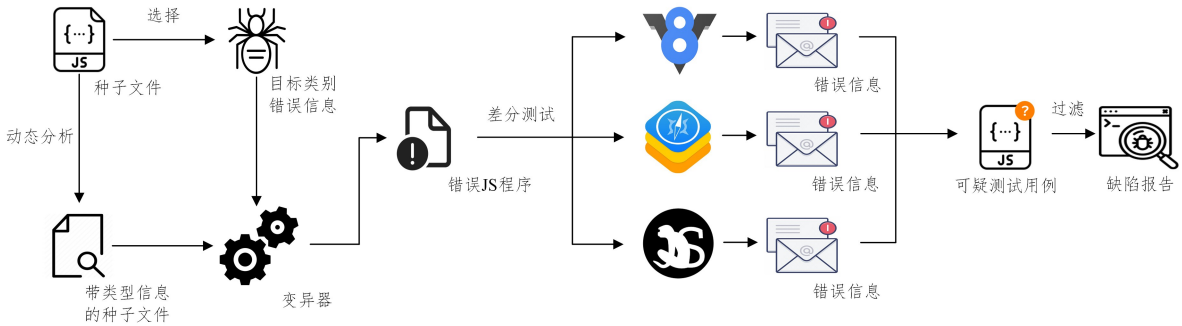


图 1 CAFJER 总体框架示意图

Fig. 1 Overall framework diagram of CAFJER

4.2 动态分析

CAFJER 用 esprima[28] 和 escodegen[29] 对种子程序进行动态分析，得到种子程序的上下文信息。esprima 和 escodegen 是实现 JavaScript 程序与抽象语法树之间转换的工具，上下文信息中包含了参数信息、API 信息和函数信息。

分析参数信息时，CAFJER 将种子程序转化为抽象语法树，从抽象语法树的结点中读取所有涉及的参数名，在种子程序的每一行代码间用 typeof 函数调用所有参数，记录下各个参数在各个位置对应的类型。分析 API 信息时，CAFJER 从抽象语法树的各个结点中得到种子程序涉及的所有 API，记录下各 API 调用的参数。分析函数信息时，CAFJER 在抽象语法树中找到所有函数声明语句，得到种子程序中所有的函数名。根据函数名找到对应的函数调用语句，利用 typeof 函数，CAFJER 得到并记录下函数调用的各个参数的类型以及函数返回值的类型。

4.3 选择目标类别错误信息

各类错误信息复杂程度不同，发现报错机制缺陷的潜力也不同[30]。CAFJER 记录了各类错误信息被选择的次数和生成的可疑测试用例数，可疑测试用例指差分测试中使 JavaScript 引擎抛出不一致错误信息的测试用例[31-32]。生成的可疑测试用例数在被选择的总次数中占比越高，CAFJER 就认为其发现缺陷的潜力越大。按照发现缺陷潜力的大小，

CAFJER 将错误信息类别进行排序，优先选择排序靠前且发现缺陷潜力大的错误信息作为目标类别错误信息。

CAFJER 采用梅特罗波利斯-黑斯廷斯算法（Metropolis-Hastings algorithm，MH 算法）指导目标类别错误信息的选择[33]。MH 算法是一种马尔可夫方法，用于在难以直接采样的概率分布中抽取随机样本序列。给定样本的建议分布，MH 算法将当前状态随机转化为下一个状态。转换过程中，MH 算法引入接受概率 p 来判断是否执行状态转换。根据现有的研究经验[14,26]，我们将建议分布设置为几何分布，即伯努利试验获得一次成功所需次数 X 的概率分布。设每次选择成功的概率为接受概率 p，则第 k 次选择首次成功的概率为 $Pr(X=k)=(1-p)^{k-1}p$。

选择目标类别错误信息时，CAFJER 先记下当前目标错误信息的类别，假设它在排序中位列第 a 位，记为 E_a。随机选择另一个位列第 b 位的错误信息类别 E_b。若 a 大于 b 即 E_b 发现缺陷的潜力更大，*CAFJER* 直接将 E_b 作为下一个目标错误信息类别。若 a 小于 b，根据 a，b 的值和接受概率 p，CAFJER 以 $(1-p)^{b-a}$ 的概率接受 E_b。若 E_b 未被接受，则 CAFJER 重新选择一个错误信息类别并重复上述过程。CAFJER 接受新目标错误信息类别 E_b 的概率总结如下：

$$Pb(E_b \mid E_a)=\min(1,(1-p)^{b-a}) \tag{1}$$

本文利用以下 3 个条件对接受概率 p 进行约束：

$$0.95\leqslant\sum_{k=1}^{61}Pr(X=k)\leqslant 1 \tag{2}$$

$$\varepsilon<(1-p)^{61-1}p \tag{3}$$

$$p>\frac{1}{61} \tag{4}$$

其中，ε 是一个非常小的实数（例如 0.001）。第一个条件确保累积概率接近 1；第二个条件确保潜力最小的错误信息类别仍有被选择的概率；第三个条件确保潜力最大的错误类别被选择的概率大于随机选择时被选择的概率。经过计算，我们将接受概率 p 设置为 0.06。

4.4　类别导向的变异方法

本节利用 API 模板、函数模板以及符号信息实现了增改 API、插入函数模板以及更改符号 3 种类别导向的变异方法。

4.4.1　增改 API

我们收集能触发错误信息的 API 和相应的参数要求，来制作 API 模板，实现增改 API 的变异方法。我们在 40 类错误信息上构建了 432 个 API 模板。

每个 API 模板都对应一类错误信息。图 2 给出了一个 API 模板的示例。API 模板的第一部分是目标类别错误信息。图 2 中 API 模板的目标类别错误信息是“RangeError: toString() radix argument must be between 2 and 36”。该错误信息提示我们“toString()”调用的参数值必须在 2～36 之间，超范围的参数会触发此类错误信息。API 模板的第二部分是触发目标类别错误信息的 API 语句，图中示例的 API 语句为“Number.prototype.toString(*v*1)”，其中 *v*1 是一个待填充的参数。API 模板的第三部分是参数要求，对 API 语句中待填充的参数进行约束，以确保生成的 API 语句能触发目标类别的错误信息。示例中只有一个待填充的参数 *v*1，API 模板要求 *v*1 是一个小于 2 或大于 36 的 Number 型数据。例如，我们可以将 1 赋值给 *v*1，得到满足条件的语句“Number.prototype.toString(1)”。它使“toString()”调用的参数值小于 2，能触发目标类别错误信息“RangeError: toString() radix argument must be between 2 and 36”。

目标类别错误信息：	RangeError: toString() radix argument must be between 2 and 36
API 语句：	Number.prototype.toString(v1)；
参数要求：	
v1：	
类型：	Number
范围：	小于 2 或大于 36

图 2　API 模板示例图

Fig. 2　Example of API template

利用 API 模板进行变异时，我们以等可能的概率随机选择增加或修改的变异策略。如算法 1 所示，对给定的种子程序 P 和目标类别错误信息 E_{ID} 执行增加 API 的变异方法时，CAFJER 首先用 esprima 和 escodegen 对种子程序进行动态分析，得到种子程序中变量的名称及类型信息 V_P，以及函数的名称及输入输出的参数类型 F_P（line1）。CAFJER 根据目标类别错误信息 E_{ID} 从 API 模板库中随机选择一个对应的 API 模板 T_{API}（line2），读取 T_{API} 中对每个参数的要求 V_{req}，利用 V_P 和 F_P 生成满足要求的参数，将它们放入列表 V 中（line3，4）。例如，API 模板需要一个 String 类型的参数，CAFJER 既可以直接生成一个 String 类型的字符串，也可以调用类型为 String 的变量，还可以调用返回值类型为 String 的函数或 API。之后，CAFJER 把生成的参数填入 API 语句中对应的待填充位置，将变异后的语句 API_p 插入种子程序中的随机位置（line5，6）。最后，检测目标类别错误信息是否需要使用严格模式，是否需要添加语句“use strict;”，并输出变异后的测试用例（line7，8）。

算法 1　增加一个 API

输入：种子程序 P，目标类别错误信息 E_{ID}

输出：变异后的程序 P_{new}

1. V_P, F_P = getContext(P)；
2. T_{API} = getAPITemplate(E_{ID})；
3. for each V_{req} in T_{API}：
4. 　V = V.addVar(getVar(V_{req}, V_P, F_P))；
5. API_p = combineAPI(T_{API}, V)；
6. P_{API} = P.add(API_p)；
7. P_{new} = isStrictMode(P_{API}, E_{ID})；
8. return P_{new}.

严格模式是一种具有限制性的 JavaScript 编译模式，对编译过程作出了额外的规定，使 JavaScript 引擎以更严格的条件编译程序，违反这些规定会触发仅在严格模式下出现的错误信息。例如，严格模式下 Number 型的变量不允许出现前导零，违反此规定会使 JavaScript 引擎抛出仅在严格模式下出现的错误信息“SyntaxError: 0-prefixed octal literals and octal escape seq are deprecated”。我们记录了仅在严格模式下出现的错误信息类别，当它们作为目标类别错误信息时，我们在程序第一行额外添加语句“use strict;”，使编译过程进入严格模式。

执行修改 API 的变异策略时，面对种子程序 P 和目标类别错误信息 E_{ID}，我们在动态分析时额外提取种子程序 P 涉及的所有 API。如果 E_{ID} 对应的某个 API 模板与种子程序 P

调用了相同的 API，则用这个 API 模板生成 API 语句替换种子程序 P 中相同的 API 语句。如果未发现相同的 API 语句，则放弃修改 API 的变异策略执行插入 API 的变异策略。

4.4.2 插入函数模板

部分复杂的错误信息难以仅靠一句 API 语句触发。我们采用插入函数模板的变异方法去触发增改 API 无法触发的错误信息类别。我们在另外 16 类错误信息上构建了 34 个函数模板。

类似于 API 模板，函数模板的第一部分是目标类别的错误信息，第三部分是参数要求。第二部分函数语句是能满足目标类别错误信息触发条件的代码块。图 3 给出了一个函数模板的示例。它对应的目标错误信息是“InternalError: too much recursion”。此类错误信息提示用户程序的迭代次数达到上限，会被存在死循环的程序触发。在函数模板的函数语句中，我们首先构建一个 JavaScript 函数 Template，在其中插入种子文件的代码片段用于丰富函数 Template 的上下文语境。之后，在函数 Template 中编写满足目标类别错误信息触发条件的代码块。示例中，此代码块具体表现为一个 if 语句，示例函数模板规定参数 v1 的类型为 Boolean，值为 true，保证 if 语句中调用函数 Template 自身的语句被反复执行，使程序出现死循环，从而触发目标类别错误信息“InternalError: too much recursion”。

```
目标类别错误信息：        InternalError: too much recursion
函数语句：                function Template(){
//种子文件的代码片段
                              if(v1){
                                Template();
                              }
                          }
参数要求：
v1：
  类型：                  Boolean
  范围：                  等于 true
```

图 3 函数模板示例图

Fig. 3 Example of function template

给定种子程序和目标类别错误信息，执行插入函数模板的变异方法时，第一步，CAFJER 根据目标类别错误信息选择一个函数模板，并将种子程序的代码片段插入函数 Template 中，丰富其上下文语境。第二步，CAFJER 利用 esprima 和 escodegen 进行动态分析，得到种子程序中的参数信息和函数信息。第三步，CAFJER 根据种子程序的参数信息和函数信息结合模板中的参数要求生成满足条件的参数，替换函数语句中待填充的参数，并在种子程序中的随机位置调用变异后的函数 Template。最后，检查目标类别错误信息是否需要严格模式，是否需要添加语句“use strict”，并输出变异后能触发目标类别错误信息的测试用例。

4.4.3 更改符号

剩下的 5 类错误信息与程序中的符号有关，难以被 API 模板或函数模板触发。

本文分析剩下的 5 类错误信息，得到每类错误信息对应的符号。例如，括号不匹配的错误信息对应着左小括号、右小括号、左中括号、右中括号、左大括号、右大括号。我们在种子程序中随机选择一个目标类别错误信息对应的符号，将它替换为其他随机符号，实现更改符号的变异方法。

4.5 差分测试

差分测试[34-36]是检测 JavaScript 引擎缺陷的常用方法。差分测试将同一测试用例输入不同 JavaScript 引擎，以比较各 JavaScript 引擎输出的结果是否相同，若其中存在差异，则说明可能发现了缺陷。

CAFJER 将变异后的测试用例输入不同 JavaScript 引擎进行差分测试，以比较各 JavaScript 引擎报错机制抛出的错误信息是否相同。但报错机制输出的错误信息是口语化的，难以直接进行比较。为此，我们可以将问题转化为错误信息对应的错误信息类别与错误位置是否相同[35]。我们为各 JavaScript 引擎的各类错误信息都构建了对应的正则表达式，用正则表达式固定的普通字符匹配错误信息中错误说明的不变字段和错误类型，用正则表达式的元字符匹配错误说明的灵活字段和错误位置，实现错误信息类别和正则表达式之间的对应关系。CAFJER 利用正则表达式匹配得到各错误信息对应的错误信息类别和错误位置。根据错误信息类别和错误位置的异同，我们发现了 3 类差分测试的结果。

1）错误信息类别和错误位置都相同。这说明各 JavaScript引擎报错机制输出的错误信息内容一致，其中不存在缺陷。

2）错误信息类别不同，错误位置相同。错误信息类别不同说明各 JavaScript 引擎报错机制认为触发错误的根本原因不同，此类可疑测试用例中很可能存在缺陷。

3）错误信息类别相同，错误位置不同。错误位置不同说明各 JavaScript 引擎报错机制认为触发错误的代码位置不同，此类可疑测试用例中可能存在缺陷。

实验过程中，CAFJER 没有发现会使各 JavaScript 引擎报错机制抛出错误信息类别和错误位置都不同的测试用例。

4.6 过滤无效测试用例

差分测试直接比较各 JavaScript 引擎报错机制输出的错误信息类别和错误位置是否完全相同，但实际上错误信息类别不同或错误位置不同并不意味着发现了新的缺陷。各 JavaScript 引擎中一些标准的设置以及错误位置的精度有所差异，这导致差分测试直接比较的方法会输出大量无效的不存在缺陷的可疑测试用例。CAFJER 根据可疑测试用例触发的错误信息以及错误位置指向的错误代码过滤以下 4 类无效的可疑测试用例。

1）已知的缺陷

CAFJER 发现的缺陷被提交后，开发人员不可能第一时间进行修复。被确认的缺陷依旧存在于 JavaScript 引擎报错机制中，能够再次被发现。CAFJER 根据错误信息和错误代码判断可疑测试用例中是否存在已知缺陷。例如，把关键词作为变量名进行声明时，SpiderMonkey 会提示缺少变量名而不是提示不能将关键词声明为变量名，这是一个已知的缺陷。过滤时，CAFJER 先确认缺少变量名这一错误信息能由已知缺陷触发，之后在错误代码中找到用关键词作为变量名的声明语句，从而判断此测试用例中存在已知缺陷，将其进行过滤。

2）对应多个类别的错误信息类别

JavaScript 引擎报错机制抛出的错误信息没有统一的

格式，并不是每个 JavaScript 引擎都有全部 61 类错误信息，个别 JavaScript 引擎的某类错误信息能对应其他引擎的几类错误信息。例如 V8 中的错误信息“Unexpected token”，它表示发现了预期之外的符号。其他 JavaScript 引擎中，这类错误信息被细分为预期之外的括号、预期之外的冒号以及预期之外的引号等多类错误信息。V8 中的错误信息“Unexpected token”能对应其他引擎中的多类错误信息。但差分测试直接比较的方法仍会将其作为可疑测试用例输出。CAFJER 记录了所有一对多的错误信息类别及相应的 JavaScript 引擎，将其与差分测试输出的可疑测试用例进行匹配，过滤此类无效的测试用例。

3）标准设置的差异

各 JavaScript 引擎标准设置的差异也会导致差分测试输出无效的可疑测试用例。过滤此类可疑测试用例时，CAFJER 检测错误位置指向的代码片段是否涉及有差异的标准。例如，JavaScript 语言的 Promise 对象能执行异步程序。但 JavaScript 引擎 JSC 不同于 V8 和 SpiderMonkey，不会抛出异步程序所触发的错误信息。如果 CAFJER 发现可疑测试用例的报错位置出现在 Promise 对象执行的异步程序中，且 JSC 没有抛出错误信息，那么 CAFJER 就认为这是一个标准设置的差异导致的无效测试用例，并将其进行过滤。

4）错误位置精度的差异

根据 JavaScript 引擎以及错误信息类别的不同，报错机制抛出错误位置的详细程度也不同。详细的错误位置会精确指向引发错误的 API、符号或函数调用，粗略的错误位置会包含一整行代码。例如，假设语句“a.b()”中包含一个错误，精度高的报错位置可能指向“b()”，精度低的报错位置可能指向“a.b()”。差分测试比较时认为它们不完全相同，会将其作为可疑测试用例输出，但实际上其中不存在缺陷。CAFJER 将检测所有错误信息类别相同但错误位置不同的可疑测试用例，若其中的错误位置间存在包含关系，则判定错误位置精度的差异使差分测试产生了误报，并过滤此类无效的测试用例。

5　实验分析

本章主要通过实验来验证 CAFJER 检测 JavaScript 引擎报错机制缺陷时的有效性。

5.1　硬件信息和目标 JavaScript 引擎

本文的实验部署在一台操作系统为 Ubuntu20.04 Linux，处理器为 Intel(R) Core(TM) i7-10700 CPU @2.90GHz，内存为 32GB 的计算机上。我们选用 V8，JSC 和 SpiderMonkey 这 3 个 JavaScript 引擎作为实验对象，它们都是运营多年的成熟 JavaScript 引擎，拥有活跃的社区。它们的开发人员认可并重视报错机制中的缺陷，bug 仓库中能找到此类的缺陷报告。

5.2　研究问题

为了验证 CAFJER 检测 JavaScript 引擎报错机制缺陷时的有效性，我们设计了以下 3 个实验进行研究。

1）RQ1：与现有先进的方法相比，CAFJER 发现 JavaScript 引擎报错机制缺陷的能力如何？

2）RQ2：CAFJER 中类别导向的变异方法对发现 JavaScript 引擎报错机制缺陷的贡献有多大？

3）RQ3：CAFJER 发现真实世界 JavaScript 引擎报错机制缺陷的能力如何？

RQ1 中我们将 CAFJER 与目前先进的相似方法 JEST[10] 和 DIPROM[17] 进行了比较，验证了 CAFJER 的有效性。RQ2 中我们编写了 CAFJER 的变体 CAFJER_r，将其中类别导向的模糊测试方法改写为随机变异方法，比较 CAFJER 和 CAFJER_r 的性能，验证 CAFJER 类别导向模糊测试方法的有效性。RQ3 中我们列举了真实世界中 CAFJER 发现并被确认的缺陷，验证了 CAFJER 在真实世界中发现 JavaScript 引擎报错机制缺陷的能力。

5.3　与现有先进方法的比较

当前没有其他检测 JavaScript 引擎报错机制的有效方法。JEST 是目前先进的检测 JavaScript 引擎编译机制缺陷的方法，它解析 ES12 的语法规则，依此生成测试用例。实验中我们只取 JEST 生成的能触发错误信息的测试用例进行研究。DIPROM 是目前先进的编译器警告信息缺陷的检测方法，通过对种子程序抽象语法树进行修剪和插入操作来生成测试用例，我们将它改写应用于 JavaScript 引擎报错机制缺陷的检测工作中。本节用 JEST 和 DIPROM 作为对照组，验证 CAFJER 检测 JavaScript 引擎报错机制缺陷时的有效性。

我们根据可疑测试用例数、可疑测试用例比例、独特缺陷数、P 值（P-value）、效应量（effect size）这几个指标对 CAFJER，JEST 和 DIPROM 进行对比。可疑测试用例数指差分测试发现输出的错误信息不一致的测试用例数；可疑测试用例比例指可疑测试用例数在总变异次数中的占比；独特缺陷数指一轮测试中发现的独特缺陷个数。

为了减小偶然性对实验的影响，我们共进行了 5 轮实验，每轮实验持续 24h，将得到的数据取平均数，实验结果如表 1 所列。

表 1　CAFJER 与现有方法的性能比较表

Table 1　Performance comparison between CAFJER and existing methods

	CAFJER	JEST	DIPROM	CAFJER_r
可疑测试用例数	34699.4	26412.0	6105.8	20723.6
可疑测试用例比例/%	40.98	23.40	12.10	26.46
独特缺陷数	5.2	2.4	0.2	0.6
P-value	—	小于 0.01	小于 0.01	小于 0.01
效应量	—	0.953	0.987	0.982

表 1 中的数据显示，一轮实验中，CAFJER 平均发现了 34699.4 个可疑测试用例，而 JEST 和 DIPROM 平均只发现了 26412.0 个以及 6105.8 个可疑测试用例，CAFJER 多发现了 31.38% 和 458.30% 的可疑测试用例数。CAFJER 发现可疑测试用例的比例高达 40.98%，显著优于 JEST 的 23.40% 和 DIPROM 的 12.10%。由于差分测试的输出中存在 4 类无效的可疑测试用例，因此 CAFJER 在一轮实验中平均只发现了 5.2 个独特缺陷，而 JEST 和 DIPROM 在一轮实验中平均只发现了 2.4 个以及 0.2 个独特缺陷，CAFJER 发现的独特缺陷数分别是 JEST 和 DIPROM 的 2.17 倍以及 26.00 倍。实验计算了 JEST 和 DIPROM 相对于 CAFJER 的 P 值和效应量。表 1 列出了 JEST 和 DIPROM 的 P 值都小于 0.01，说明 CAFJER 的表现优于 JEST 和 DIPROM。在 CAFJER 与对照组的比较中，若效应量大于 0.5 则说明 CAFJER 更有效，若效应量等于 0.5 则说明 CAFJER 和对照组一样有效，

若效应量小于 0.5 则说明对照组更有效。表 1 中 JEST 和 DIPROM 的效应量都大于 0.9，证明 CAFJER 检测 JavaScript 引擎报错机制缺陷的能力显著优于 JEST 和 DIPROM。

在两个对照组中，JEST 能够快速地生成更多的测试用例，但输出可疑测试用例的比例不高，发现报错机制缺陷的能力不强，找到的独特缺陷数也远少于 CAFJER。另一个对照组 DIPROM 的表现最差，说明 DIPROM 对种子程序抽象语法树进行修剪和插入操作的变异方法不适用于 JavaScript 引擎报错机制缺陷的检测。

基于以上的数据和分析，我们可以对 RQ1 做出解答，与现有的先进模糊测试方法相比，CAFJER 检测 JavaScript 引擎报错机制的效果更好。

5.4 类别导向变异方法的有效性

为验证 CAFJER 类别导向变异方法的有效性，本文编写了 CAFJER 的变体 CAFJER_r 来进行对比实验。CAFJER_r 将 CAFJER 中类别导向的变异方法修改为随机变异的方法，随机变异的方法随机选择种子程序中的参数、API 或符号进行增删改操作。

表 1 所列的数据显示，CAFJER 平均发现了 34 699.4 个可疑测试用例，CAFJER_r 平均发现了 20 723.6 个可疑测试用例，CAFJER 比 CAFJER_r 多发现了 67.44%的可疑测试用例。CAFJER 发现可疑测试用例的比例为 40.98%，比 CAFJER_r 发现可疑测试用例的比例 26.46%高 14.52%。每轮实验中，CAFJER 平均发现了 5.2 个独特缺陷，CAFJER_r 平均只发现了 0.6 个独特缺陷，CAFJER 发现的独特缺陷数比 CAFJER_r 多 7.67 倍。因此，我们认为 CAFJER 发现报错机制缺陷的能力优于 CAFJER_r。

表 1 显示，CAFJER_r 的 P 值小于 0.01，说明 CAFJER 的表现优于 CAFJER_r。同时，CAFJER_r 的效应量大于 0.9，也证明了 CAFJER 检测 JavaScript 引擎报错机制缺陷时更有效。

实验结果表明，CAFJER 的性能显著优于 CAFJER_r。基于以上的数据和分析，我们可以对 RQ2 做出解答，类别导向的变异方法在 CAFJER 中起到了至关重要的作用，有效提高了 CAFJER 检测 JavaScript 引擎报错机制缺陷的能力。

5.5 被确认的缺陷

在 3 个月的实验过程中，我们共向开发者提交了 17 个缺陷报告，其中 7 个已被确认。被确认的缺陷中，2 个在 V8 中，1 个在 JSC 中，4 个在 SpiderMonkey 中。表 2 列出了所有 CAFJER 提交后被确认的缺陷。

优先级 P1 表示缺陷最受开发者重视，P5 表示缺陷的影响最小。CAFJER 发现的缺陷中 3 个优先级为 P2，3 个优先级为 P3，1 个优先级为 P5，说明 CAFJER 发现的缺陷得到了各 JavaScript 引擎开发者的重视。

表 2 被确认的缺陷

Table 2 Confirmed defects

	ID	优先级	缺陷类别	引擎
1	12918	P3	错误类别不正确	V8
2	13187	P2	错误位置不正确	V8
3	244018	P2	错误类别不正确	JSC
4	1771690	P3	错误类别不正确	SpiderMonkey
5	1775215	P3	错误类别不正确	SpiderMonkey
6	1780916	P2	错误类别不正确	SpiderMonkey
7	1782849	P5	错误位置不正确	SpiderMonkey

CAFJER 发现的缺陷中，有 5 个缺陷抛出了不正确的错误类别，有 2 个缺陷抛出了不正确的错误位置。本文认为，报错机制产生缺陷的根本原因是 ES12 没有对错误信息进行详细定义。ES12 忽视了错误信息类别，使错误信息类别缺少官方的统一规范，只能由开发人员按照个人理解编写。受限于开发人员自身的编程经验，各类错误信息中较容易出现缺陷，因此 CAFJER 发现了 5 个错误类别不正确的缺陷。而 ES12 规定了 JavaScript 正确的语法规则，各 JavaScript 引擎开发人员能根据规定确定错误位置。但实际 JavaScript 程序中存在众多复杂的特殊情况，开发人员难免有所疏漏，CAFJER 仍发现了 2 个错误位置不正确的缺陷。基于这个发现，本文认为如果 ES12 对错误信息类别进行规范统一，使各 JavaScript 引擎开发人员在编写错误说明时有据可依，就能有效减少报错机制中的缺陷数量。

基于以上的数据和分析，我们可以对 RQ3 做出解答，CAFJER 能够有效地发现真实世界 JavaScript 引擎报错机制中的缺陷。

结束语 为了检测 JavaScript 引擎报错机制中的缺陷，本文提出了类别导向的模糊测试方法 CAFJER。我们分析 JavaScript 社区中的文档和差分测试输出的错误信息，将 JavaScript 错误信息分为 61 类，并设计了能触发每类错误信息的变异方法。给定一个种子程序，CAFJER 首先对其进行动态分析，为其选择一个目标类别的错误信息。其次，CAFJER 使用类别导向的变异方法生成测试用例。接着，CAFJER 将生成的测试用例输入不同 JavaScript 引擎进行差分测试，比较抛出的错误信息间是否存在差异。最后，CAFJER 自动过滤重复的和无效的测试用例，输出缺陷报告。

我们将目前最先进的编译器警告信息缺陷的检测方法 DIPROM 和目前最先进的 JavaScript 引擎编译机制缺陷的检测方法 JEST 加以改写，并将其应用到 JavaScript 引擎报错机制缺陷的检测工作中。与 DIPROM 和 JEST 相比，CAFJER 多输出了 458.30%和 31.38%的可疑测试用例数，多发现了 26.0 倍和 2.17 倍的独特缺陷数。CAFJER 发现 JavaScript 引擎报错机制缺陷的性能显著优于现有方法。真实世界中，CAFJER 在 3 个月内提交了 17 个缺陷报告，其中 7 个已被确认。

下一步的工作中，我们计划创建更多的 API 模板和函数模板，并尝试变异错误的 JavaScript 程序使其触发不同的错误信息类别。

参 考 文 献

[1] RYAN DAHL. Node. js [EB/OL]. (2022) [2022-12-21]. https://nodejs. org/.

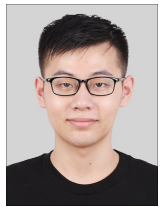
[2] GITHUB. Github Annual Report in 2021[EB/OL]. (2022) [2022-12-21]. https://octovers-e. github. com/2022/top-programming-languages.

[3] MOZILLA FOUNDATION. Spidermonkey [EB/OL]. (2022) [2022-12-21]. https://spidermonkey. dev/.

[4] SPIDERMONKEY. SpiderMonkey bug # 1775215. [EB/OL]. (2022-6) [2022-12-21]. https://bugzilla. -mozilla. org/show_bug. cgi?id=1775215.

[5] WANG J, CHEN B, WEI L, et al. Superion: Grammar-Aware

Greybox Fuzzing[C]// 2019 IEEE/ACM 41st International Conference on Software Engineering(ICSE). ACM，2019.
[6] LI Y，XUE Y，CHEN H，et al. Cerebro：context-aware adaptive fuzzing for effective vulnerability detection[C]// 2019 27th ACM Joint Meeting. ACM，2019.
[7] 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.
[8] MATHIS B，GOPINATH R，ZELLER A. Learning input tokens for effective fuzzing[C]// 29th ACM SIGSOFT International Symposium on Software Testing and Analysis(ISSTA'20). ACM，2020.
[9] HE X，XIE X，LI Y，et al. SoFi：Reflection-Augmented Fuzzing for JavaScript Engines[C]// Association for Computing Machinery. 2021.
[10] PARK J，AN S，YOUN D，et al. Jest：N+ 1-version differential testing of both javascript engines and specification[C]// 2021 IEEE/ACM 43rd International Conference on Software Engineering(ICSE). IEEE，2021：13-24.
[11] HAN H S，OH D H，CHA S K. Codealchemist：Semantics-Aware code Generation to Find Vulnerabilities in Javascript Engines[C]// The 2019 Annual Network and Distributed System Security Symposium. 2019.
[12] LEE S，HAN H S，CHA S K，et al. Montage：A Neural Network Language Model-Guided JavaScript Engine Fuzzer[J]. arXiv：2001. 04107，2020.
[13] YE G，TANG Z，TAN S H，et al. Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing：，10. 1145/3453483. 345405-4[P]. 2021.
[14] ECMA-262. The specification of JavaScript language [S/OL]. https：//tc39. es/ecma262/，2021.
[15] Google. V8[EB/OL]. (2022) [2022-12-21]. https：//v8. dev/.
[16] Apple. JavaScriptCore[EB/OL]. (2022) [2022-12-21]. https：//github. com/phoboslab/JavaScript-Core-iOS.
[17] TANG Y X，JIANG H，ZHOU Z D，et al. Detecting compiler warning defects via diversity-guided program mutation. [C]// IEEE Transactions on Software Engineering，2021.
[18] CHEN J J，HU W X，HAO D，et al. An empirical comparison of compiler testing techniques. [C]// Proceedings of the 38th International Conference on Software Engineering. 2016：180-190.
[19] Grammar-based interpreter fuzz testing[D]. Christian Holler：Saarland University，2011.
[20] 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，2019，47(11)：2312-2331.
[21] BARTON P M，LARS F，BRYAN S. An empirical study of the reliability of unix utilities[J]. Communications of the ACM，1990，33(12)：32-44.
[22] HAN H S，SANG K C. IMF：Inferred Model-based Fuzzer[C]// Acm Sigsac Conference. ACM，2017.
[23] HOLLER C，HERZIG K，ZELLER A. Fuzzing with Code Fragments[C]// Proceedings of the 21st Usenix Security Symposium. 2012.
[24] YANG X，YANG C，EIDE E，et al. Finding and understanding bugs in C compilers[C]// ACM Sigplan Conference on Programming Language Design & Implementation. ACM，2011.
[25] MICHAZ. AFL[EB/OL]. (2022)[2022-12-21]. http：//lcamtuf. coredump. cx/afl/.
[26] SUN C，LE V，SU Z. Finding and Analyzing Compiler Warning Defects[C]// IEEE/ACM International Conference on Software Engineering. IEEE，2017.
[27] SLASHDATA. State of the developer nation [EB/OL]. (2021) [2022-12-21]. https：//slashdataweb-sitecms. s3. amazonaws. com/sample_reports/VZtJWxZw5Q9NDSAQ. pdf.
[28] ARIYA HIDAYAT. esprima [EB/OL]. (2021) [2022-12-21]. https：//github. com/jquery/esprima.
[29] YUSUKE SUZUKI. escodegen[EB/OL]. (2020) [2022-12-21]. https：//github. com/estools/escodegen.
[30] LYU C，JI S，ZHANG C，et al. MOPT：optimized mutation scheduling for fuzzers[C]// USENIX Security Symposium. 2019.
[31] CHEN Y，SU T，SU Z. Deep differential testing of JVM implementations[C]// 2019 IEEE/ACM 41st International Conference on Software Engineering(ICSE). IEEE，2019.
[32] CHEN Y，SU T，SUN C，et al. Coverage-directed differential testing of JVM implementations[C]// ACM Sigplan Conference on Programming Language Design & Implementation. ACM，PUB27，New York，NY，USA，2016.
[33] METROPOLIS N，ROSENBLUTH A W，ROSENBLUTH M N，et al. Equation of state calculations by fast computing machines[J]. The Journal of Chemical Physics，1953，21(6)：1087-1092.
[34] CHEN J，BAI Y，DAN H，et al. Learning to Prioritize Test Programs for Compiler Testing[C]// IEEE/ACM International Conference on Software Engineering. IEEE Computer Society，2017.
[35] LE V，AFSHARI M，SU Z. Compiler validation via equivalence modulo inputs[J]. ACM Sigplan Notices，2014，49(6)：216-226.
[36] OFENBECK G，ROMPF T，PÜSCHEL M. RandIR：differential testing for embedded compilers[C]// ACM Sigplan Symposium on Scala. ACM，2016.



LU Ling，born in 1998，postgraduate. His main research interests include software test and so on.

JIANG He，born in 1980，Ph.D，professor，Ph.D supervisor，is a member of China Computer Federation. His main research interests include intelligent software engineering and industrial software testing.

（责任编辑：喻藜）