子树类型敏感的 JavaScript 引擎灰盒测试技术

王聪冲[1], 甘水滔[2], 王晓锋[1]

[1]江南大学人工智能与计算机学院 无锡 中国 214122
[2]数字工程与先进计算国家重点实验室 无锡 中国 214083

摘要 JavaScript 引擎是浏览器的重要组成部分, 很多攻击都针对 JavaScript 引擎发起, 业界对面向 JavaScript 引擎的漏洞挖掘技术一直展现出强烈的需求。本文提出一种面向 JavaScript 引擎的子树类型敏感灰盒测试技术, 并且实现了系统 ILS, 在路径反馈的模糊测试框架上, 通过对 JavaScript 代码的语法分析, 构建子树类型敏感的变异策略, 能够大幅提升测试种子的有效率, 从而驱动更高的代码覆盖能力和漏洞发现能力。通过将 ILS 和多个主流 JavaScript 引擎漏洞挖掘工具 Superion、CodeAlchemist 进行性能对比, 在 Jerryscript、ChakraCore 和 JavaScriptCore 等典型 JavaScript 引擎对象上的测试实验表明: ILS 在 24 h 内, 其种子测试有效率上提升 36%, 代码行覆盖率上能提升 72%, 代码函数覆盖率上能提升 80%, 漏洞发现效率上提升 100%。最后, ILS 在这 3 个 JavaScript 引擎总共发现 26 个未知 Bug, 并得到厂商的确认和修复。

关键词 路径反馈; 模糊测试; JavaScript 引擎; 抽象语法树; 子树类型敏感
中图法分类号 TP311 TN92 DOI 号 10.19363/J.cnki.cn10-1380/tn.2021.07.08

Subtree Type Sensitive Greybox Testing Technique of JavaScript Engines

WANG Congchong[1], GAN Shuitao[2], WANG Xiaofeng[1]

[1] School of Artificial Intelligence and Computer Science, Jiangnan University, Wuxi 214122, China
[2] State Key of Laboratory of Mathematical Engineering and Advanced Computing, Wuxi 214083, China

Abstract JavaScript engine is critical part of any browser. Many attacks of browser are launched from JavaScript engine that bringing strong demand to industry for vulnerability analysis of JavaScript engine. In this paper, we propose a new subtree type-sensitive gray box testing technology for JavaScript engine, and implement a prototype System that called ILS. Through designing a subtree type sensitive mutation strategy based on the path feedback fuzzing framework with syntax analysis on JavaScript code, ILS could greatly improve the effectiveness of test cases generation, that driving higher code coverage and vulnerability discovery capabilities. By comparing ILS with other typical tools Superion and CodeAlchemist on three familiar JavaScript engines (i.e., Jerryscript ,ChaKraCore and JavaScriptCore), ILS could reach 36% more seed generation efficiency, 72% more line coverage, 80% more function coverage, and find 100% more bugs in 24 hours. Moreover, ILS found 26 new bugs in this three JavaScript engines.

Key words path feedback; fuzzing; JavaScript engine; abstract syntax tree; subtree type-sensitive

1 介绍

JavaScript 引擎是浏览器的必要组成部分, 其安全性一直备受关注[1], 由于其代码量巨大、逻辑极其复杂, 导致漏洞挖掘技术发展一直无法满足 JavaScript 引擎的安全测试需求, 针对 JavaScript 引擎漏洞攻击事件层出不穷。通过对 CVE 漏洞库统计, 暴露在 Jerryscript[2]、ChakraCore[3]、SpiderMonkey[4]、JavaScriptCore[5]、V8[6]其高危漏洞条目超过 450 条, 其中 ChakraCore 和 V8 的数量分别超过 150 条和 200 条。并且在 2020 年上半年暴露的高危漏洞数量已超过 2019 年全年暴露的总数。具体如图 1 所示。通过进一步对过去 JavaScript 引擎漏洞暴露来源调查, 模糊测试是发现 JavaScript 引擎漏洞的主流自动化检测方法[7]。

模糊测试[8]通过不断自动构造随机非预期的输入数据给目标软件, 实时监视目标软件运行异常信号, 能有效发现软件存在的脆弱性[9]。模糊测试主要

通讯作者: 甘水滔, 博士, Email: ganshuitao@gmail.com。
本课题得到国家自然科学基金项目(No. 61672264, No. 61972182), 国家重点研发计划项目(No. 2016YFB0800305)资助。
收稿日期: 2020-09-18; 修改日期: 2020-12-11; 定稿日期: 2021-06-24

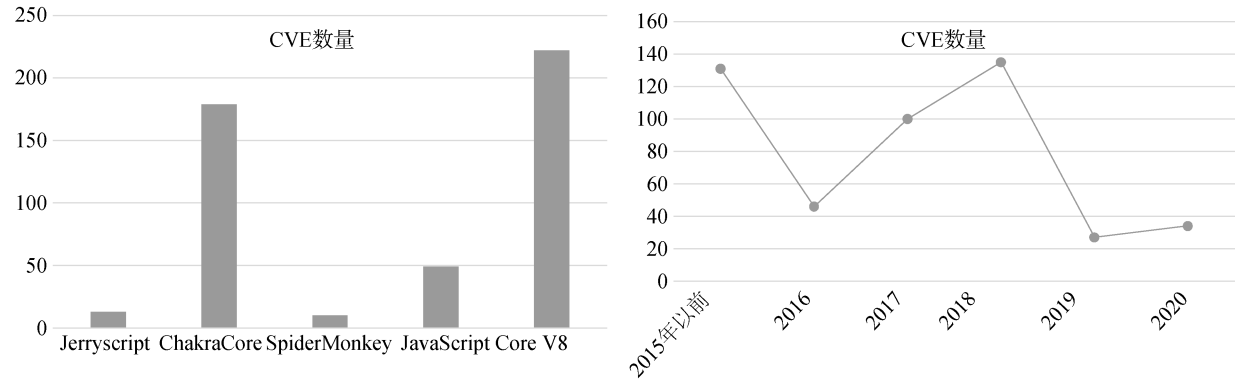


图 1　主流 JavaScript 引擎的 CVE 漏洞暴露统计情况

Figure 1　CVE vulnerability exposure statistics for mainstream JavaScript engines

分为黑盒和灰盒模型, 在 2014 年以前基本以黑盒模型为主, 黑盒模型不具备程序信息反馈能力, 具有运行速度快, 能适应复杂规模代码软件等优点, 但测试效率一直受限于随机性过大、代码覆盖能力弱等问题[10]。从 2015 年开始, 自经典灰盒测试系统 AFL[11]问世以来, 通过路径覆盖信息实时反馈, 在文档解析、图片解析、音频解析等文件型输入的软件目标测试过程中达到较高代码覆盖率, 发现大量未公开漏洞, 并实现了向协议型[12]和内核型[13]对象应用扩展。灰盒模型已逐步成为最有效的主流模糊测试手段, 并且和 libFuzzer[14]、honggfuzz[15]等灰盒工具集成在谷歌的可持续开源模糊测试服务 OSS-fuzz[16]中, 截止到 2020 年 6 月, OSS-fuzz 已经在 300 个开源项目中发现了上万个可触发漏洞[17]。

模糊测试在 JavaScript 引擎的应用方法同样划分为黑盒和灰盒模型两类, 其难点在于如何高效的生成并测试高质量的 JavaScript 代码测试用例。不同于文档解析等文件型测试例, JavaScript 代码需要遵从严格的语法规范。JavaScript 引擎在解析 JavaScript 代码过程中, 一旦遇见不符合 JavaScript 语法规则代码, 就会终止对整个测试例的解析, 从而无法触及 JavaScript 引擎的关键功能模块, 难以达到理想的测试效果。然而, 如果直接采用 AFL 等灰盒工具的字节或比特级随机变异策略, 极易破坏 JavaScript 代码的合法性, 导致 JavaScript 引擎测试过程中效率极其低下。因此, JavaScript 引擎模糊测试在过去以基于代码生成的黑盒模型为主, 代表性工具有 JSfunfuzz[18]、LangFuzz[19]、CodeAlchemist[20]。JSfunfuzz 通过 JavaScript 代码语法模板生成测试用例, 容易产生合法的 JavaScript 代码, 但需要花费大量的人力构建语法模板, 而且容易漏掉有价值的语法模式。LangFuzz 通过对已有合法 JavaScript 代码进行解析, 转换为抽象语法树形式, 将样本拆分成子树代码片段放入代码池中, 并不断使用代码池中的子树替代当前测试用例的非终止节点来构建新的测试用例, 这种方式极大提升了测试例自动生成能力, 但仍然容易产生非法的代码, CodeAlchemist 进一步引入控制流和数据流分析, 定位和缓解未定义变量问题, 进一步提升了代码生成的正确性。但总体来说, 黑盒模型难以摆脱执行速度慢和代码覆盖能力弱等缺点。

为了弥补 JavaScript 引擎黑盒模型的不足, 软工顶级会议工作 Superion[21]构建了基于 JavaScript 代码子树变异的灰盒模型, 在 AFL 模型中增加了子树变异阶段, 以子树替代的方式去对种子变异, 一定程度降低了 AFL 的 JavaScript 代码生成错误概率, 同等测试环境下, 其代码覆盖和漏洞发现能力明显优于黑盒模型。然而, Superion 在变异阶段对子树类型并不敏感, 这种盲目的子树替换策略导致生成的代码出现严重的类型错误和语法错误。

基于现有 JavaScript 引擎灰盒模型的不足, 本文提出了一种基于子树类型敏感的 JavaScript 引擎灰盒测试技术, 并且实现了灰盒测试系统 ILS。ILS 在路径反馈的模糊测试框架基础上[22], 通过对 JavaScript 代码进行精准语法分析, 对子树集合进行类型识别和分类, 并且在变异阶段构建了子树类型敏感的变异策略, 使用同类型子树进行覆盖变异, 不仅大幅提升了测试种子的有效率, 而且带来了更高的代码覆盖能力和漏洞发现能力。本文挑选了 Jerryscript、ChakraCore 和 JavaScriptCore 三个常用 JavaScript 引擎作为测试对象, 通过对 ILS 和 CodeAlchemist、Superion 等工具进行 24 h 性能测试对比, 实验表明: 在保持灰盒模型执行性能优势的情况下, 错误率比 Superion 降低了 36%, 比 CodeAlchemist 降低了 21%。在代码行覆盖率上, ILS 相比 Superion 提升了 72%, 相比 CodeAlchemist 提升 48%; 在函数覆盖率上, ILS 相比 Superion 提升 80%, 相比 CodeAlchemist 提升

50%。在漏洞发现数量上, ILS 相比 Superion 提升了 100%。最后, ILS 在这三个 JavaScript 引擎上总共发现了 26 个未知 Bug, 并均得到了厂商的确认和修复。

本文的贡献总结如下:

1) 面向 JavaScript 引擎对象, 提出了一种子树类型敏感的灰盒测试模型, 并实现了相应原型系统 ILS。

2) 与经典黑盒和灰盒测试工具相比, ILS 能显著降低 JavaScript 代码测试用例的错误率, 并且能大幅提高 JavaScript 引擎的代码覆盖能力。

2 背景与相关工作

本节先对 JavaScript 引擎的基本工作原理进行描述, 然后对 JavaScript 引擎漏洞挖掘方法相关工作进行总结, 最后进一步说明本文方法和传统方法的差异。

2.1 JavaScript 引擎工作原理

JavaScript 引擎是浏览器重要组成部分, 是根据 ECMAScript[23] 定义的语言标准来动态执行 JavaScript 字符串的重要程序。JavaScript 引擎具有代码量大, 工作原理和逻辑结构复杂等特点。JavaScript 引擎动态解析 JavaScript 代码过程可以分为语法检查阶段和运行阶段, 语法检查阶段包括词法分析和语法分析, 运行阶段包括预解析和实际执行。其具体工作流程如图 2 所示: JavaScript 引擎首先对 JavaScript 代码进行语法分析得到抽象语法树(Abstract Syntax Tree, AST), 然后进一步解析得到字节码(bytecode), 在生成字节码的同时, 优化编译器生成高度优化的机器代码, 生成的机器代码最后替换原来的字节码。JavaScript 引擎在优化代码的时候会使用未优化的字节码进行工作, 有效提高了 JavaScript 引擎性能。

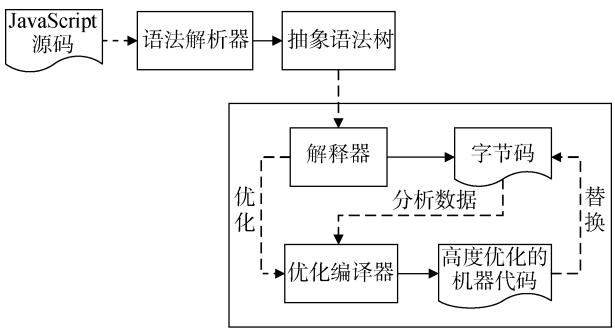


图 2 JavaScript 引擎工作流程

Figure 2 JavaScript engine workflow

在 JavaScript 解析过程中, 一旦遇到错误后面的代码就无法执行。在 ECMAScript 标准中定义了 5 种运行时错误: 语法错误(SyntaxError)、范围错误(RangeError)、引用错误(ReferenceError)、类型错误(TypeError)和 URI 错误(URIError)。如图 3 所示: 第 1 行中, var 后面没有申明变量, 直接进行赋值操作, 不符合语法规则, 就会抛出语法错误; 第 2 行中, 一旦变量数值超出有效范围, 便会抛出范围错误; 第 3 行中, 一旦引用一个不存在的变量时, 便会抛出引用错误; 第 4 行中, 当值的类型或参数不是预期类型时, 便会抛出类型错误; 第 5 行中, 使用全局 URI 处理函数遇见不正确参数时, 便会抛出 URI 错误。由于 URI 错误很少在 JavaScript 解析过程中出现, 因此本文只考虑前 4 种错误类型。

```
1  var = 1;  //SyntaxError
2  [].length = −1;  //RangeError
3  a;  //ReferenceError
4  a = new 2();  //TypeError
5  decodeURI(℃%N0);  //URIError
```

图 3 JavaScript 引擎解析代码过程中常见的运行时错误类型

Figure 3 Common runtime errors in the process of JavaScript engine parsing code

2.2 JavaScript 引擎漏洞挖掘相关工作

经典程序分析方法符号执行[24-27]和污点分析[28-31]需要对测试软件的指令进行逐条解析, 分析速度慢, 当处理复杂代码对象 JavaScript 引擎, 符号执行极其容易受限于路径爆炸和复杂约束不可解问题, 污点分析极易遭受内存爆炸、过度污染和欠污染问题[32], 难以在 JavaScript 引擎分析中产生作用, 在过去鲜有针对 JavaScript 引擎的符号执行和污点分析工作。另外, 经典静态分析工具 Coverity[33]、Klockwork[34]、Fortify[35] 只对程序语言敏感, 因此也能适用于 JavaScript 引擎分析, 但虚报率高, 需要大量人力去验证静态报告的真实性。针对 JavaScript 引擎适配和优化的静态分析工作[36-38]更多关注如何降低 JavaScript 引擎运行开销。

相比之下, 模糊测试具有速度快、能适应于复杂代码对象等优势, 具备直接发现可触发漏洞的能力。因此, 本文在相关工作部分主要关注模糊测试方法, 从测试例生成方法上看, 模糊测试大体可分为基于变异的模糊测试方法和基于生成的模糊测试方法, 以下分别针对 JavaScript 引擎对这两类相关工作进行归纳。

2.2.1 基于生成的模糊测试方法

基于生成的模糊测试方法借助手工构建的输入语法模型生成有效的输入数据, 在一个理想的输入语法模型支持下, 才能测试重要功能模块。在 JavaScript 引擎模糊测试应用上, 基于生成的模糊测试技术一直是发展的重点, 主要发展路线可概括为以下三点: 1)不断提升语法模型的自动化生成能力; 2)不断丰富生成的 JavaScript 代码模式; 3)不断消减 JavaScript 代码的错误率。

在代码自动生成能力提升方面, 早期的代表性工具有 Peach[39], 在应用到 JavaScript 引擎测试上, 需要为每个对象创建模板文件, 需要划分好随机字段和确定字段, 这将消耗大量的人力和时间成本。JSfunfuzz 基于 JavaScript 语法模板, 根据公开漏洞样本中学习已有特征生成测试用例, 只能适应 SpiderMonkey 解析引擎, 扩展性能力差。CodeAlchemist、DIE[40]等方法通过对初始代码样本构建语法特征抽象模型, 尽量保留有价值的控制流和数据流特征, 在该基础上构建变异策略生成新的代码样本; Skyfire[41]、Montage[42]等方法进一步提升了代码生成的自动化能力, 在已有训练代码样本基础上, 采用学习的方式生成新的代码样本; 以上几种方法对初始代码样本极其敏感, 人力成本主要体现在初始代码样本筛选上, 在实际应用过程中, 这些模型更多借助标准测试集和公开漏洞触发样本, 最后均达到了较为理想的代码生成效果。

在 JavaScript 代码语法模式生成能力方面, Peach、JSfunfuzz 主要依赖已有语法模板, 不容易推导新的语法模式。Skyfire 利用带概率的上下文敏感语法(PCSG)从大量样本中学习语法和语义规则, 在代码生成过程中优先选择低概率的语义产生规则以生成分布合理的测试用例。CodeAlchemist 会按照 JavaScript 语句的粒度分解种子文件, 针对每个块语句生成多种变体提升代码块的丰富性, 通过对多个代码块组合产生嵌套循环等结构。Montage 从 Test262[43]数据集中收集了超过 3 万个 JavaScript 代码样本, 通过对代码样本中变量和函数等标识符重新命名和子树分片, 形成 LSTM 模型的初始训练数据, 能有效学习出 JavaScript 代码中的控制流和数据流依赖关系, 从而推导出传统模型能力之外的语法模式, 该系统经过一个半月的测试, 共发现了 34 个未知 Bug。DIE 以公开漏洞库可触发漏洞样本为基础, 识别并保留其中隐式的循环、特定跳转结构信息以及抽象语法树类型信息, 这些特征的保留有助于模糊测试阶段集中在容易引发脆弱性的功能模块(JIT 优化模块)上进行, 最后发现了 48 个未知 Bug。

在 JavaScript 代码错误率消减方面, JSfunfuzz 在已有种子模板基础上, 结合实现预先准备好的函数、对象、操作符等 JavaScript 代码特征库, 随机填充可写字段, 生成可用测试例, 由于缺乏各语句和语句间的合法性检查, 这种方式会带来较严重的语法错误、引用错误和类型错误。LangFuzz 在生成代码的过程中, 用当前已有的变量去替代生成的代码片段中未知变量, 但无法避免变量在使用后才声明, 而且会增加该变量对象的重复使用概率, 从而导致大量的引用错误。Skyfile 在生成测试用例的过程中, 也没有考虑变量定义、使用等情况, 同样容易造成运行时错误。针对这一问题, CodeAlchemist 提出使用数据流分析技术确定未定义变量和已定义变量, 并结合运行时插桩技术探测已执行变量的类型。CodeAlchemist 在 JavaScript 代码生成过程中, 不断向代码尾部扩展代码片段, 每次扩展代码片段都会确保其中的变量在之前已经被定义并且具备正确的类型, CodeAlchemist 的优化方法能有效降低 LangFuzz 方法中存在的错误率, 相比 JSfunfuzz, 能生成超出 5 倍以上的有效测试例。Montage 采用 LSTM 模型推导不同代码片段之间的关联关系, 能有效学习出类似变量定义和使用的先后次序等信息, 有效降低非学习代码生成策略存在的引用错误率。DIE 通过保留的结构化语义和类型信息, 能有效提升变异的准确性, 大幅增加了种子的有效率, 相比 CodeAlchemist, 可以降低一半的代码错误率。

2.2.2 基于变异的模糊测试方法

基于变异的模糊测试方法[44-48]通过对已有的测试例进行随机变异产生新的测试例, 可以不依赖输入语法模型, 正由于其模型简单易用, 对人工经验依赖小, 相比基于生成的模糊测试方法而言, 在非结构化语法输入对象中应用尤为广泛[49-51]。但对于 JavaScript 代码等结构化语法输入, 变异策略需要考虑结构化语法特征, 不然会引发严重的错误率。在 JavaScript 引擎模糊测试应用上, 基于变异的模糊测试技术发展历程可概括为: 1)从黑盒模型逐步转向带路径覆盖信息反馈的灰盒模型; 2)从随机变异策略逐步发展为 JavaScript 代码语法语义敏感的变异策略。

IFuzzer[52]为针对 JavaScript 引擎的黑盒模型, 首先把 JavaScript 代码样本转换为树的形式, 再采用遗传算法对测试样本进行选择、交叉和变异, 选择阶段主要参考适应度函数值大小对代码样本进行优先选择, 并对代码中子树节点进行交叉变异生成新的测试例, 该方法和 LangFuzz 相比, 发现了 SpiderMonkey

上 LangFuzz 之外的 16 个未知 Bug。Fuzzilli[53]会把 JavaScript 代码样本转换为中间代码, 中间代码的每条指令由输入变量、操作码以及输出变量构成, 在中间代码上进行变异不仅有助于继承已有控制流和数据流特征, 还能降低代码错误率。不同于 AFL, Fuzzilli 在测试过程中忽略了边的循环次数, 这样容易丢失覆盖 JIT 优化功能模块的高价值测试例。近几年业界在 JavaScript 引擎模糊测试的变异策略上增加了子树分析, 以子树为变异单位进行变异操作, 如 Montage、CodeAlchemist、Nautilus、Superion。其中, Superion 充分发挥 AFL 的优势, 在其基础上添加子树变异阶段, 和 AFL、JSfunfuzz 对比, 大幅提升 JavaScript 引擎对象上的代码覆盖能力和漏洞发现能力。但以上工作普遍没有对子树进行类型分析, 对子树类型的不敏感会极大削弱变异策略的种子生成效率。

2.3 本文方法与已有方法区别

根据以上相关工作分析可知, 子树变异和路径覆盖反馈能力是当前多个 JavaScript 引擎模糊测试的重要方法特征, 但这些工作普遍缺乏对子树的类型分析, 容易导致严重的 JavaScript 代码错误率, 从而丢失一些子树类型敏感的脆弱路径。本文根据当前工作的不足, 构建了一个具备子树敏感变异能力和路径覆盖反馈能力的灰盒测试系统 ILS。表 1 更清晰展示 ILS 和现有工作的方法差异, 主要从是否具备路径反馈能力、是否考虑 JavaScript 语法结构特征、是否采用子树覆盖变异、是否考虑同类型子树覆盖操作、是否开放工具源码、是否自动化程度较高这几个方面进行比较。

表 1 ILS 与现有相关工作的方法差异

Table 1 The difference between ILS and existing JavaScript engine fuzzers

工具名称	路径反馈	JavaScript 语法结构特征	子树覆盖	同类型子树覆盖	开源	自动化程度高
JSfunfuzz	×	✓	×	×	✓	×
Langfuzz	×	✓	×	✓	×	✓
AFL	✓	×	×	×	✓	✓
Skyfire	×	✓	×	×	✓	✓
Skyfire+AFL	✓	✓	×	×	✓	✓
Fuzzilli	✓	✓	×	×	✓	✓
CodeAlchemist	×	✓	×	×	✓	✓
Superion	✓	✓	✓	×	✓	✓
Nautilus	✓	✓	✓	×	✓	✓
Montage	×	✓	×	×	✓	✓
ILS	✓	✓	✓	✓	-	✓

为了更直观地展示 ILS 在漏洞检测能力上的提升, 图 4 和图 5 给出了具体案例分析。如图 4, Jerryscript 引擎暴露的 CVE-2018-11418 和 CVE-2018-11419 两个缓存区溢出漏洞, 二者的触发漏洞样本仅体现在 RegExp 函数中的参数有所不同, 如果对 CVE-2018-11418 的漏洞触发样本中的参数部分进行变异就能快速的触发漏洞 CVE-2018-11419。在后续的实际测试中, ILS 通过同类型子树覆盖, 提取

```
(new RegExp("[\u0020")).exec("u");
```
(a) CVE-2018-11418

```
(new RegExp("[\u0")).exec("u");
```
(b) CVE-2018-11419

图 4 已知的 Jerryscript 漏洞

Figure 4 Existing Jerryscript vulnerabilities

```
new RegExp("\ud800", "u").exec("u")
```
(a) bug-3870

```
new RegExp("\\u', 'u').exec("u")
```
(b) bug-3871

图 5 ILS 新发现的漏洞

Figure 5 New bugs found by ILS

CVE-2018-11418 中的子树信息, 发现如图 5 的两个未知缓存区溢出漏洞 bug-3870 和 bug-3871。可以发现, 这两个新漏洞样本只替换了 CVE-2018-11418 样本中 RegExp 函数的参数, 该参数内容通过语法树解析得到的是同一种子树类型。

3 ILS 系统设计与实现

3.1 整体工作流程

本文所提出的 ILS 方法在 AFL 基础上进行扩展, 更改了 AFL 的变异策略。ILS 使用一种基于子树类型敏感的 JavaScript 引擎灰盒测试方法, 降低了 AFL 在生成 JavaScript 代码时的错误概率, 并具有更高的代码覆盖能力和漏洞发现能力。

AFL 是谷歌开发的标杆性灰盒测试系统, 它的工作流程如图 6 中的 Execution 所示, 可以分为两步: 对目标程序进行插桩和对已插桩的程序进行模糊测

试。在插桩阶段, AFL 对目标程序编译时, 在每个基本块插入一段代码, 用来记录当前基本块和上一个基本块的信息, 主要代码如下:

```
1 cur_location = <COMPILE_TIME_RANDOM>;
2 shared_mem[cur_location ^ prev_location]++;
3 prev_location = cur_location >> 1;
```

cur_location 的值是随机产生的, 用来标记当前基本块, shared_mem 数组记录上一个基本块和当前基本块这条边的命中次数, 如果命中次数的分类发生改变, 则认为产生了新的覆盖率, prev_location 表示上一个基本块的标记值, 右移一位是为了区分基本块的执行顺序。插桩完成后, AFL 对已插桩的目标程序进行模糊测试, 可以分为如下 5 个步骤:

1) 从输入队列(包括初始种子和产生新路径的测试用例)中挑选出一个种子;

2) 保证覆盖率不变的情况下减小种子的长度;

3) 对种子进行变异生成新的测试用例;

4) 执行新的测试用例, 并监控目标程序执行状态;

5) 根据目标程序执行结果更新输入队列, 然后转到步骤 1。

通过不断循环上述 5 个步骤, AFL 持续对目标程序进行测试, 如果在步骤 4 中目标程序发生崩溃, 则记录崩溃信息, 并保存产生唯一崩溃的测试用例。

ILS 的整体工作流程如图 6 所示, 可以分为初始种子预处理和运行两部分, 图中蓝色框部分是 ILS 在 AFL 基础上增加的工作。ILS 首先在初始种子预处理阶段通过相关语法分析工具对初始种子进行语法分析生成抽象语法树, 然后对得到的抽象语法树进行类型识别和分类构建相同类型子树池, 最后在变异阶段对测试输入增加语法分析, 从同类型子树池中随机挑选出子树进行覆盖变异。下面, 本文详细讨论 ILS 方法的实现细节。

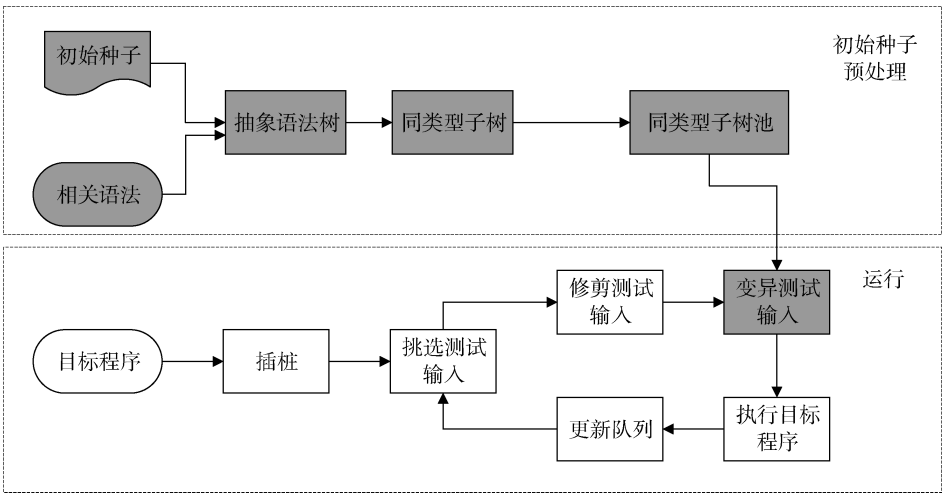


图 6 ILS 工作流程

Figure 6 The workflow of ILS

3.2 初始种子预处理

ILS 对初始种子进行语法分析得到抽象语法树, 在语法树层次对子树进行类型识别, 并按类型分类构建子树池, 详细内容如算法 1 所示。算法输入为初始种子 *seeds* 和相关语法 *G*。依次从 *seeds* 中选择一个 *seed* 进行解析, 首先根据语法 *G* 将测试输入 *in* 解析成抽象语法树 AST(第 4 行), 解析过程中如果出现错误, 则说明测试输入 *in* 是和语法无关的, 将其舍弃(第 5~7 行)。如果成功解析得到抽象语法树 *seed_ast*, 则依次访问语法树的每个节点, 获取节点的类型和内容, 并保存到 *typetree* 数据集合中(第 8 ~ 12 行), *typetree* 是一个 map 容器, 它的 key 为子树的类型, value 是一个 set 容器, 保存子树的内容。对所有的初始种子解析完成后, 可以得到所有子树类型和子树内容集合 *typetree*。根据 *typetree* 集合中每个类型的名字 *tree.id* 在数据库中创建对应的表, 并在表中插入该类型的所有子树 *tree*[*id*]构建同类型子树池(第 14~17 行)。

算法 1. 初始种子预处理

输入: *seeds*: 初始种子, *G*: 相关语法

1: *type* = ∅

2: *typetree*[*id*][*text*] = ∅ //id is the type name, text is the set of subtree's contexts

3: **foreach** *seed* in *seeds* **do**

4: parse *seed* into AST *seed_ast* according to *G*

5: **if** there are any parsing errors **then**
6: **return**
7: **endif**
8: *ctx* = VISIT(*seed_ast*) // *ctx* is the visiting node according to *G*
9: **while** *ctx* **do**
10: *typetree*[TYPEID(*ctx*)] = *typetree*[TYPEID(*ctx*)] ∪ {GETTEXT(*ctx*)}
11: VISITCHILDREN(*ctx*)
12: **endwhile**
13: **endfor**
14: **foreach** *tree* in *typetree* **do**
15: CREATETABLE(*tree.id*)
16: INSERTTABLE(*tree*[*id*])
17: **endfor**

3.3 同类型子树变异

根据 3.2 节中得到的同类型子树池, 更改 AFL 的变异策略, 使用同类型子树覆盖变异生成新的测试用例进行模糊测试。算法 2 介绍了 ILS 同类型子树变异过程, 算法输入为测试输入 *in*, 相关语法 *G* 和同类型子树池 *typetree*, 输出为新生成的测试用例集合 *T*。首先使用语法 *G* 对测试输入 *in* 进行解析, 解析过程和算法 1 解析相同(第 3~6 行)。解析完成后, 依次访问抽象语法树的每个节点, 获取节点的类型和内容, 并保存到 *subtree* 数据集合中, 此步骤和算法 1 中访问语法树相同(第 7 ~ 11 行)。获取语法树所有节点类型和内容信息后, 对语法树 *in_ast* 进行替换变异, 依次选择子树类型集合 *tree*[*id*], 并把 *in_ast* 中 *tree*[*id*] 类型的每个子树 *s* 随机替换为子树池 *typetree*[*id*]中同一类型的子树 *typetree*[*id*][*random*], 生成新的测试用例 *ret*, 并保存到集合 *T* 中(第 12~20 行)。最后返回测试用例集合 *T*(第 21 行)。

参考其他相关工作中对单个测试用例变异的次数, 本文选择单个测试用例最大变异次数为 10000。通过 10000 除以这个测试用例中的子树类型数目, 再除以该类型子树个数得到每个子树的平均变异次数(第 14 行), 这样保证测试用例中每个子树都能被替换, 并且保证每个子树类型的总替换次数相同。

算法 2. 同类型子树变异

输入: *in*: 测试输入, *G*: 相关语法, *typetree*[*id*][*text*]: 同类型子树池
输出: *T*: 新生成的的测试用例集合
1: *T*= ∅
2: *subtree*[*id*][*interval*] = ∅ //*id* is the type name, *interval* is the set of subtree's location
3: parse *in* into AST *in_ast* according to *G*
4: **if** there are any parsing errors **then**
5: **return**
6: **endif**
7: *ctx* = VISIT(*in_ast*) // *ctx* is the visiting node according to *G*
8: **while** *ctx* **do**
9: *subtree*[TYPEID(*ctx*)] = *subtree*[TYPEID(*ctx*)] ∪ { GETTEXT (*ctx*)}
10: VISITCHILDREN(*ctx*)
11: **endwhile**
12: **foreach** *tree* in *subtree* **do**
13: **foreach** *s* in *tree*[*id*] **do**
14: *count* = 10000/*subtree.size*()/*tree*[*id*].*size*()
15: **foreach** *i* in *count* **do**
16: *ret* = replace *s* in *in_ast*'s copy with type-tree[*id*][*random*()]
17: *T* = *T* ∪ {*ret*}
18: **endfor**
19: **endfor**
20: **endfor**
21: **return** *T*

以下本文用一个例子详细介绍 ILS 的工作流程。使用图 7a 作为算法 1 中的初始种子, 图 7b 作为算法 2 中的测试输入。图 7a 经过语法解析得到图 8 所示语法树, 其中每一个非叶子节点和它的所有孩子节点表示一颗子树, 该非叶子节点的类型表示了这颗子树的类型。从 program 根节点开始访问该语法树, 到<EOF>叶子节点结束, 访问结束可以得到所有子树类型和子树内容集合构建子树池, 如表 2 所示, Id 表示子树的类型, Text 表示子树的内容。

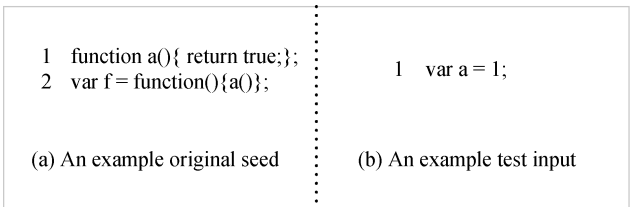


图 7 输入种子和测试用例
Figure 7 The original seed and test input

从表 2 中可以看到, 在不同的类型中, 可能它们的子树是相同的, 如图 4 红色框部分的子树 a(), 它的类型有 FunctionBody、ExpressionSequence、ArgumentsExpression 3 种。这增加了子树类型的丰富性, 在 ILS 子树变异过程中可以更有效的生成新的测试用例。

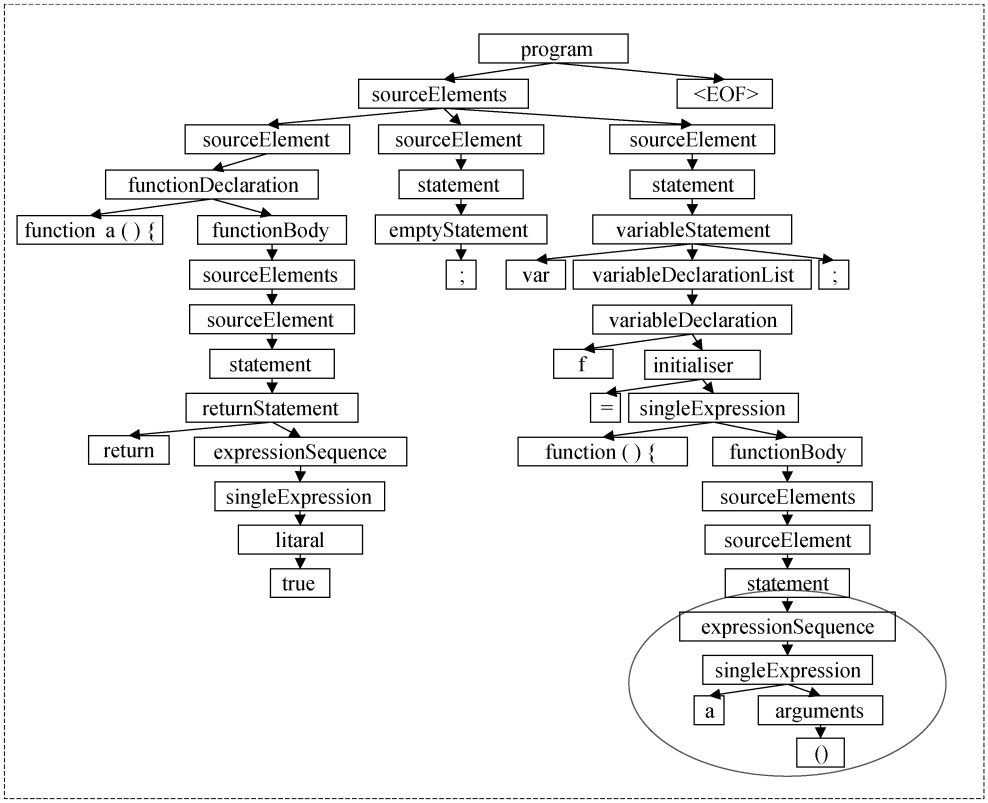


图 8　抽象语法树 AST

Figure 8　Abstract Syntax Tree

表 2　图 7a 通过语法分析得到的类型和内容

Table 2　The type and context obtained through syntax analysis from figure 7a

Id	Text
Literal	true
Arguments	()
Initialiser	= function(){a()}
FunctionBody	return true; a()
ReturnStatement	return true;
LiteralExpression	true
ExpressionSequence	true a()
FunctionExpression	function(){a()}
ArgumentsExpression	a()
IdentifierExpression	a

在同类型子树替换变异阶段, 对图 7b 通过语法分析得到表 3。依次选择表 3 中的 Id 的每个子树替换为表 2 中相同 Id 的子树, 可以得到 3 个新的测试用例, 如下所示:

1 var a = true; //Id 为 Literal

2 var a = function(){a()}; //Id 为 Initialiser

3 var a = true; //Id 为 LiteralExpression

在语法解析过程中还能得到 Program、SourceElements、Statement 等类型的子树, 第一个类型表示整颗语法树, 进行整个语法树替换变异是没有意义的, 第二、三个类型表示的是这个子树的源元素和状态, 在整个语法树中会多次出现, 但是并没有实际意义, 所以删除这些无用的类型, 只保留具有一定意义的子树类型, 如表 2 中一共得到 10 种不同有意义的类型。

表 3　图 7b 通过语法分析得到的类型和内容

Table 3　The type and context obtained through syntax analysis from figure 7b

Id	Text
Literal	1
Initialiser	=1
NumericLiteral	1
LiteralExpression	1

和 ILS 相比, Superion 的子树替换策略是无类型的, 这样在替换的时候会造成很多错误, 如图 7b 中的数字 1 被图 7a 中的 a()替换, 生成 var a = a(), 会造成 TypeError。但是 ILS 同类型子树替换策略就会减少这类问题, Literal 类型的数字 1 只能被表 2 中同类型的 true 子树替换, 生成新的测试用例 var a = true, 可以顺利被解析引擎运行。

3.4 实现细节

本文在 AFL 基础上, 使用 C/C++语言实现子树类型敏感的灰盒模糊测试系统 ILS。该系统使用 ANTLR 4[54]作为语法分析工具对初始种子进行词法分析和语法分析, 获取初始种子的类型和子树构建同类型子树池, 在子树变异阶段, 使用 ANTLR 4 对测试输入进行词法分析和语法分析, 随机从子树池中挑选同类型的子树替换测试输入中的每个子树生成新的测试用例。

该系统充分利用了 AFL 的优势, 能够对目标进行快速测试, 并通过覆盖率反馈, 提高对目标的代码覆盖率, 增加发现漏洞的概率。而且该系统容易扩展, 扩展相关的语法可以对其他结构化语法输入对象(如 XML 解析器)进行测试。

4 实验与测试分析

4.1 实验设置

为了验证 ILS 的有效性, 本文将此模型与 Superion 和 CodeAlchemist 进行实验对比, 在表 1 所列出的开源工具中, Superion 是基于子树变异的模糊测试方法中的典型工具, 在相关 JavaScript 引擎中发现了 23 个未知的 Bug, CodeAlchemist 是基于生成的模糊测试方法中的典型工具, 在相关 JavaScript 引擎中发现了 11 个未知的 Bug。本文与这两个工具进行实验对比, 可以体现出 ILS 方法的先进性。

本文分别从漏洞发现能力、代码覆盖能力、测试例有效率、执行速度 4 个方面进行性能对比: 漏洞发现能力最能体现出工具的有效性; 代码覆盖能力是衡量测试结果好坏的基本标准, 能够整体上体现出模糊测试工具的效果; 生成测试用例有效性对于 JavaScript 引擎的模糊测试来说尤其重要, 一般随机变异产生的测试用例都不能被 JavaScript 引擎解析运行; 执行速度是模糊测试工具中重要的一环, 表示时间效率。

本文选择三个开源的 JavaScript 引擎 JerryScript、ChakraCore 和 JavaScriptCore 作为测试对象, JerryScript 版本为 2.2.0, ChakraCore 版本为 1.12.0.0-beta, JavaScriptCore 版本为 2.27.4。JerryScript 是三星集团开发的轻量级物联网 JavaScript 引擎, 适用于嵌入式设备, 只需要很低的 CPU 的性能和内存空间。ChakraCore 是 Microsoft Edge 浏览器中 Chakra JavaScript 引擎的核心部分。JavaScriptCore 是 WebKit 浏览器的 JavaScript 解析引擎部分, WebKit 是一个跨平台的浏览器引擎, 它支持 Safari、iBook 和 App Store, 以及各种 Mac 系统, iOS 和 Linux 平台。

为了平衡实验的系统资源开销, 本文对这 3 个模糊测试工具都只使用 4 个并行进程进行实验, 保证同时运行 4 个 JavaScript 引擎, 在一定程度上体现系统资源的公平性。

在测试用例方面, 对 JerryScript 引擎收集了 1698 个测试用例, 对 ChakraCore 引擎收集了 1766 个测试用例, 对 JavaScriptCore 引擎收集了 2713 个测试用例。

实验中所用机器为带有 64G 内存的 Dell PowerEdge R740, 操作系统为 64 位的 Ubuntu 18.04。本文参考其他相关工作的测试时间, 一般测试时间为 24 h, 所以本文也对 JavaScript 引擎测试 24 h。由于模糊测试具有随机性, 所以重复进行 10 次实验, 对代码覆盖率、测试例有效率、执行速度取平均值进行对比, 并对比 10 次实验的漏洞发现总数。

4.2 漏洞发现能力

重复进行 10 次 24 h 实验后, Bug 发现情况如表 4 所示。在 Bug 发现率方面, ILS 相比于 Superion 提升了 100%。CodeAlchemist 并不支持对 JerryScript 引擎进行测试, 而且在 ChakraCore 和 JavaScriptCore 引擎中都没有发现 Bug。表 5 详细列出了截止本文写作时间, ILS 发现的所有 Bug 具体信息, 共发现 26 个不同的 Bug。在 JerryScript 引擎中, 共发现 22 个 Bug, 包括 1 个 Use-After-Free、3 个 Buffer Overflow、2 个 Stack Overflow、1 个 Memory Corruption 和 15 个 Assertion Failure。在 ChakraCore 引擎中, 一共发现 3 个 Assertion Failure。在 JavaScriptCore 引擎中, 一共发现 1 个 Memory Corruption。由于 CodeAlchemist 并没有发现 Bug, 所以没有放入表 5 中进行对比。

表 4 24 h Bug 发现情况对比

Table 4 The comparison of Bug discovery in 24 hours

JS Engine	ILS	Superion	CodeAlchemist
Jerryscript	18	9	N/A
ChakraCore	2	1	0
JavaScriptCore	1	0	0

表 5 ILS 发现的 Bug

Table 5 The Bugs found by ILS

#	JS Engine	Bug	Type	Status	Superion
1	Jerryscript 2.2.0①	Bug-3748	Use-After-Free	Fixed	√
2		Bug-3749	Buffer Overflow	Fixed	×
3		Bug-3753	Stack Overflow	Fixed	√
4		Bug-3819	Assertion Failure	Fixed	√
5		Bug-3820	Assertion Failure	Fixed	×
6		Bug-3821	Assertion Failure	Fixed	√
7		Bug-3822	Assertion Failure	Fixed	√
8		Bug-3823	Assertion Failure	Fixed	√
9		Bug-3824	Assertion Failure	Fixed	×
10		Bug-3825	Assertion Failure	Fixed	√
11		Bug-3830	Memory Corruption	Fixed	×
12		Bug-3834	Assertion Failure	Fixed	×
13		Bug-3835	Assertion Failure	Fixed	√
14		Bug-3845	Assertion Failure	Fixed	×
15		Bug-3869	Assertion Failure	Fixed	√
16		Bug-3870	Buffer Overflow	Fixed	×
17		Bug-3871	Buffer Overflow	Fixed	×
18	ChakraCore 1.12.0.0-beta②	Bug-3880	Assertion Failure	Fixed	√
19		Bug-3881	Assertion Failure	Fixed	×
20		Bug-3934	Stack Overflow	Fixed	×
21		Bug-3935	Assertion Failure	Fixed	×
22		Bug-3936	Assertion Failure	Fixed	√
23		Bug-6453	Assertion Failure	Confirmed	√
24		Bug-6454	Assertion Failure	Confirmed	×
25		Bug-6455	Assertion Failure	Confirmed	×
26	JavaScriptCore 2.27.4③	Bug-212460	Memory Corruption	Fixed	×

4.3 代码覆盖提升

在模糊测试中, 代码覆盖率是衡量测试结果好坏的基本标准, 能够整体上体现出模糊测试工具的效果, 代码覆盖率越高, 越有可能发现新的 Bug。

本文分别对 JerryScript 和 JavaScriptCore 进行了覆盖率对比实验, 并使用 afl-cov[55]收集覆盖率信息, 结果表明 ILS、Superion 和 CodeAlchemist 对代码行覆盖率和函数覆盖率都有提升, ILS 提升效果更好。因为 CodeAlchemist 不支持测试 JerryScript 解析引擎, 所以没有进行 JerryScript 解析引擎的覆盖率对比。

对于 JerryScript, 实验结果如图 9 所示。在代码行覆盖率方面, ILS 相比于 Superion 提升 48%, 在函数覆盖率方面, ILS 相比于 Superion 提升 41%。

对于 JavaScriptCore, 实验结果如图 10 所示。在代码行覆盖率上, ILS 相比 Superion 提升了 72%, 相比 CodeAlchemist 提升 48%; 在函数覆盖率上, ILS 相比 Superion 提升 80%, 相比 CodeAlchemist 提升 58%。

4.4 测试用例有效率提升

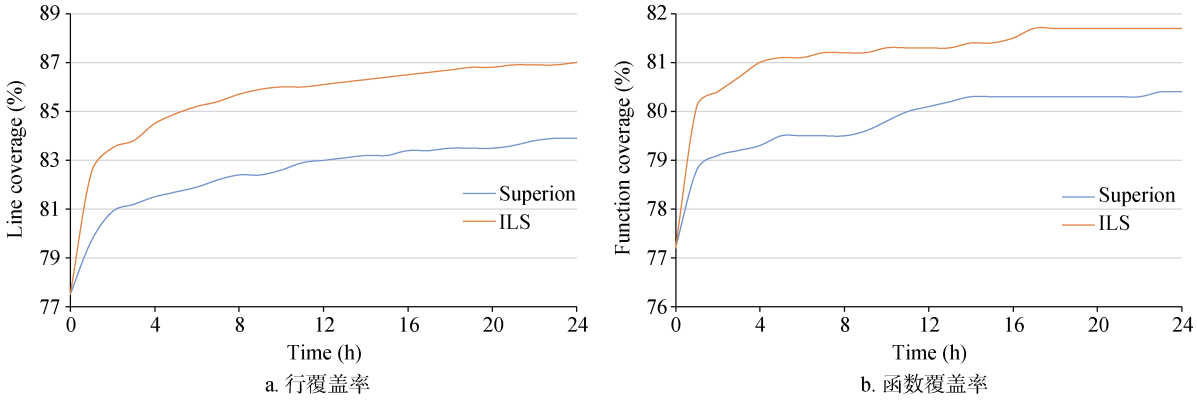
生成测试用例有效性对于 JavaScript 引擎的模糊测试来说非常重要, 一般的模糊测试工具如 AFL, 随机变异产生的测试用例基本上都是无效的, 因为如果不符合对应的语法规则, 解析引擎将拒绝继续执行, 所以很难发现系统运行时的 Bug。

图 11 显示了生成测试用例错误率的实验结果, ILS、Superion 和 CodeAlchemist 的平均错误率分别是 46.16%, 58.74%和 72.49%。因为 ILS 在替换子树时没有对子树的引用进行检查, 所以 ILS 的 ReferenceError 错误是最多的, 但是 ILS 能有效的减

① https://github.com/jerryscript-project/jerryscript/issues/created_by/owl337

② https://github.com/microsoft/ChakraCore/issues/created_by/owl337

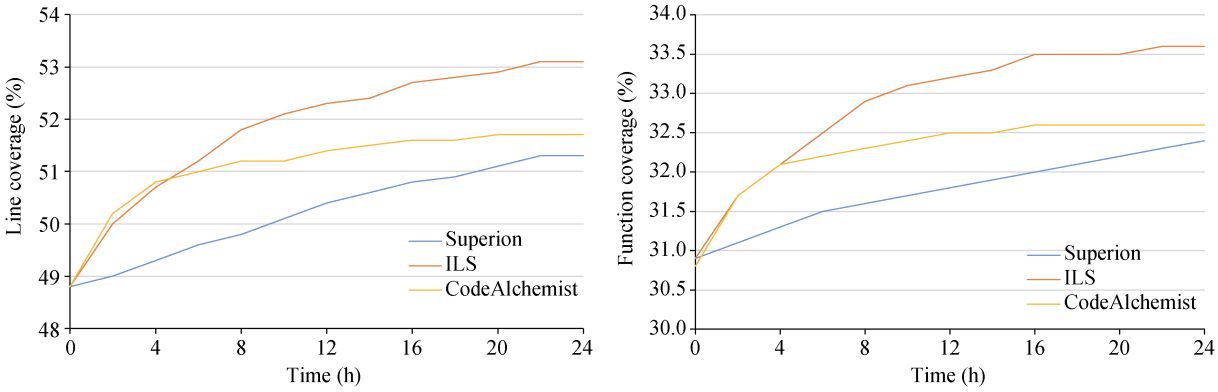
③ https://bugs.webkit.org/show_bug.cgi?id=212460



a. 行覆盖率 b. 函数覆盖率

图 9 JerryScript 引擎覆盖率对比

Figure 9 The comparison coverage rate of JerryScript



a. 行覆盖率 b. 函数覆盖率

图 10 JavaScriptCore 引擎覆盖率对比

Figure 10 The comparison coverage rate of JavaScriptCore

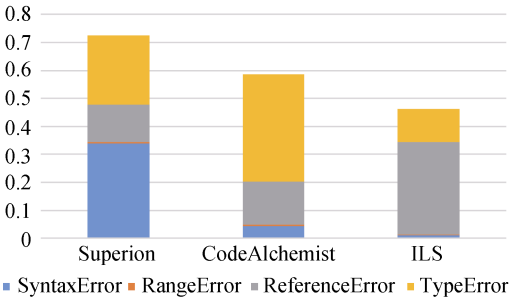


图 11 生成种子错误率对比

Figure 11 The error Comparison of new seeds

少其他的错误率，种子测试有效率比 Superion 提升了 36%，比 CodeAlchemist 提升了 21%。

4.5 执行性能分析

执行速率表示单位时间内调用目标软件运行的次数，执行速度越快，越有可能在短时间内发现新的 Bug。

本文统计 10 次实验每个工具生成的测试用例个数，并平均计算得到每个 JavaScript 引擎每秒运行次数，结果如表 6 所示。由于 ILS 和 Superion 都是基于 AFL 开发的，具有 AFL 的 forkserver 模式加速，所以执行速度快，CodeAlchemist 是自己控制 JavaScript 引擎执行，并没有对执行速度进行优化，所以性能较低。在 JerryScript 引擎中，平均执行速率 ILS 比 Superion 每秒降低了 3.03 次；在 Chakracore 引擎中，平均执行速率 ILS 比 Superion 每秒降低了 1.01 次；在 JavaScriptCore 引擎中，整体平均执行速率 ILS 比 Superion 每秒降低了 2.6 次。

ILS 执行速率比 Superion 有所降低。但总体来说相差不大，在覆盖率提升显著的情况下，这个执行速率的差异是可以接受的。

5 总结与讨论

本文详细分析了针对 JavaScript 引擎的模糊测试技术，并在路径反馈的模糊测试框架基础上提出一种基于子树类型敏感的 JavaScript 引擎灰盒测试方法 ILS。该方法通过对 JavaScript 代码进行语法分析，把子树进行识别分类，并在变异过程中使用同类型子

表 6　每秒执行速率对比

Table 6　The execution rate pre second comparison

JS Engine	ILS	Superion	CodeAlchemist
JerryScript	105.11	108.14	N/A
ChakraCore	27.93	28.94	0.53
JavaScriptCore	23.88	26.48	0.51

树进行替换变异。实验结果表明, 该方法能显著提升生成测试用例的有效率, 并具有更高的代码覆盖能力, 而且发现了 26 个新的 Bug。

ILS 在下一步工作中, 将结合基于生成的模糊测试技术, 对 JavaScript 代码进行更细粒度的控制流分析和数据流分析, 在子树替换变异过程中消除引用未知变量, 进一步消除引用错误以提升代码生成的有效率。

参考文献

[1] Reis C, Dunagan J, Wang H J, et al. BrowserShield[J]. *ACM Transactions on the Web*, 2007, 1(3): 11.

[2] Samsung. Jerryscript, JavaScript engine for the Internet of Things. https://github.com/jerryscript-project/jerryscript. 2020.

[3] Microsoft. ChakraCore,The core part of the Chakra JavaScript engine that powers Microsoft Edge. https://github.com/microsoft/ChakraCore. 2020

[4] Mozilla. SpiderMonkey, The JavaScript engine for Firefox. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. 2020

[5] Apple. JavaScriptCore, The built-in JavaScript engine for WebKit. https://trac.webkit.org/wiki/JavaScriptCore. 2020

[6] Google. V8, V8 is Google's open source high-performance JavaScript and WebAssembly engine. https://v8.dev/. 2020

[7] CVE. Search JavaScript CVE list. https://cve.mitre.org/. 2020

[8] Richard McNally, Ken Yiu, and Duncan Grove. Fuzzing: The State of the Art[C]. *Australia: DSTO Defence Science Organisation*, Jan. 2012.

[9] Meng G Z, Liu Y, Zhang J, et al. Collaborative Security[J]. *ACM Computing Surveys*, 2015, 48(1): 1-42.

[10] 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.

[11] Zalewski M. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. 2015.

[12] Pham V T, Böhme M, Roychoudhury A. AFLNET: A Greybox Fuzzer for Network Protocols[C]. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*, 2020: 460-465.

[13] J. Hertz and T. Newsham. ProjectTriforce: AFL/QEMU fuzzing with full-system emulation. Available: https://github.com/nccgroup/TriforceAFL, 2017.

[14] Serebryany K. Continuous Fuzzing with libFuzzer and AddressSanitizer[C]. *2016 IEEE Cybersecurity Development*, 2016: 157.

[15] R. Swiecki. Honggfuzz. http://code. google. com/p/honggfuzz, 2016.

[16] Google. Continuous fuzzing of open source software. https://opensource.google/projects/oss-fuzz, 2020

[17] Google. Bugs found by OSS-fuzz. https://bugs.chromium.org/p/oss-fuzz/issues/list, 2020.

[18] M. Security, funfuzz. https://github.com/MozillaSecurity/funfuzz, 2007.

[19] Holler C, Herzig K, Zeller A, et al. Fuzzing with code fragments[C]. *Usenix Security Symposium*, 2012: 38.

[20] 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.

[21] Wang J J, Chen B H, Wei L, et al. Superion: Grammar-Aware Greybox Fuzzing[C]. *2019 IEEE/ACM 41st International Conference on Software Engineering*, 2019: 724-735.

[22] Böhme M, Pham V T, Roychoudhury A. Coverage-Based Greybox Fuzzing as Markov Chain[C]. *The 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016: 1032-1043.

[23] Ecma International. ECMAScript 2015 language specification. https: //www.ecma-international.org/ecma-262/6.0/, 2015.

[24] King J C. Symbolic Execution and Program Testing[J]. *Communications of the ACM*, 1976, 19(7): 385-394.

[25] Sun W Q, Li Z P. A Static Analyzer of the C++ Programs Based on Symbolic Execution[J]. *Electronic Technology*, 2018, 47(8): 97-104.
(孙文全, 李兆鹏. 基于符号执行技术的 C++程序静态分析[J]. *电子技术*, 2018, 47(8): 97-104.)

[26] Cadar C, Dunbar D, Engler D, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs[C]. Operating Systems Design and Implementation, 2008: 209-224.

[27] Yi Q P, Yang Z J, Guo S J, et al. Eliminating Path Redundancy via Postconditioned Symbolic Execution[J]. *IEEE Transactions on Software Engineering*, 2018, 44(1): 25-43.

[28] Denning D E. A Lattice Model of Secure Information Flow[J]. *Communications of the ACM*, 1976, 19(5): 236-243.

[29] Denning D E, Denning P J. Certification of Programs for Secure Information Flow[J]. *Communications of the ACM*, 1977, 20(7): 504-513.

[30] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-Aware Evolutionary Fuzzing[C]. *The 2017 Annual Network and Distributed System Security Symposium*, 2017.

[31] Gan S T, Zhang C, Chen P, et al. GREYONE: Data Flow Sensitive Fuzzing[C]. *The 29th Security Symposium Security*, 2020: 2577-2594.

[32] Kang M G, McCamant S, Poosankam P, et al. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation[C]. *The 2011 Annual Network and Distributed System Security Symposium*, 2011.

[33] Synopsys. Coverity scan static analysis. https://scan.coverity.com/.

[34] Perforce. Klockwork. Static Code Analysis for C, C++, C#, and Java. https://www.perforce.com/products/klocwork.

[35] HP. Fortify Static Code Analyzer. https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview.

[36] Gary Soeller. uchex. https://cseweb.ucsd.edu/~dstefan/cse291-fall16/.

[37] Brown F, Narayan S, Wahby R S, et al. Finding and Preventing Bugs in JavaScript Bindings[C]. 2017 IEEE Symposium on Security and Privacy, 2017: 559-578.

[38] Omar C, Aldrich J. Programmable semantic fragments: the design and implementation of typy[J]. *ACM SIGPLAN Notices*, 2016, 52(3): 81-92.

[39] Peach fuzzer platform. Peachfuzz. http://www.peachfuzzer.com/products/peach-platform/.

[40] Park S, Xu W, Yun I, et al. Fuzzing JavaScript Engines with Aspect-Preserving Mutation[C]. *2020 IEEE Symposium on Security and Privacy* , 2020: 1629-1642.

[41] Wang J J, Chen B H, Wei L, et al. Skyfire: Data-Driven Seed Generation for Fuzzing[C]. *2017 IEEE Symposium on Security and Privacy*, 2017: 579-594.

[42] Lee S, Han H, Cha S K, et al. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer[EB/OL]. 2020

[43] Technical Committee 39 ECMA International. Test262. https://github.com/tc39/test262.

[44] Miller B P, Cooksey G, Moore F. An Empirical Study of the Robustness of MacOS Applications Using Random Testing[J]. *Operating Systems Review*, 2007, 41(1): 78-86.

[45] Spike Fuzzing Platform. SPIKE. http://www.immunitysec.com/resources free software.shtml.

[46] Chen T Y, Leung H, Mak I K, et al. Adaptive random testing[J]. *Lecture Notes in Computer Science*, 2004: 320-329.

[47] FileFuzz platform. FileFuzz. http://www.fuzzing.org/.

[48] Zzuf. Transparent application input fuzzer. http://caca.zoy.org/wiki/zzuf, 2007.

[49] Renata H, Akos, Tibor Gyimothy. Grammarinator: a grammar-based open source fuzzer[C]. *The 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018: 45-48.

[50] Miller B P, Koski D, Lee C P, et al. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services[R]. *University of Wisconsin-Madison Department of Computer Sciences*, 1995.

[51] Forrester J E, Miller B P. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing[C]. *WSS'00: The 4th conference on USENIX Windows Systems Symposium*, 2000(4): 6.

[52] Veggalam S, Rawat S, Haller I, et al. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming[C]. *Computer Security - ESORICS 2016*, 2016: 581–601.

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

[54] Antlr's grammar list for different languages. https://github.com/antlr/grammars-v4.

[55] M. Rash. afl-cov-afl fuzzing code coverage. https://github.com/mrash/afl-cov.

王聪冲 于 2018 年在江南大学大学计算机科学与技术专业获得学士学位。现在江南大学学校计算机科学与技术专业攻读硕士学位。研究领域为漏洞挖掘、软件安全。研究兴趣包括: 软件漏洞挖掘, 网络安全。Email: congchongwang@163.com

甘水滔 博士, 长期从事系统安全和程序分析方法研究。Email: ganshuitao@gmail.com

王晓锋 于 2007 年在哈尔滨工业大学计算机系统与结构专业获得博士学位。现任江南大学副教授。研究领域为网络仿真、网络安全。Email: wangxf@jiangnan.edu.cn