高技术通讯 2022 年 第 32 卷 第 12 期:1226-1235

doi:10.3772/j.issn.1002-0470.2022.12.002

脚本语言执行引擎的模糊测试技术综述①

孙力立② 武成岗③ 许佳丽 张培华 唐博文 谢梦瑶

(计算机体系结构国家重点实验室(中国科学院计算技术研究所) 北京 100190)

(中国科学院大学计算机科学与技术学院 北京 100190)

摘 要 脚本语言作为解释性语言,需要由脚本语言执行引擎动态解释执行。由于脚本语言的广泛应用,其执行引擎也在各种平台上得到广泛部署。因此,脚本语言执行引擎中的安全漏洞往往具有很高的安全影响。模糊测试作为一种有效的自动化漏洞挖掘方法,在挖掘脚本语言执行引擎的软件缺陷和漏洞方面也有重要作用。本文对近年来国内外学者在该领域的研究进行了系统的总结,介绍了模糊测试和脚本语言执行引擎的基本概念,整理了现有的脚本语言执行引擎的模糊测试工作的评价指标,分类梳理了脚本语言执行引擎的模糊测试工作,阐述了该领域所关注的研究问题和解决方法。最后,根据现有工作的不足和研究趋势,提出具有潜力的下一步研究方向。

关键词 脚本语言执行引擎; 模糊测试; 漏洞挖掘; 软件缺陷检测

0 引 言

脚本语言[1]是一种解释性语言。不同于 C、C++等编译型语言,脚本语言依赖于相应的脚本语言执行引擎解释执行。由于脚本语言的广泛使用,其执行引擎也被广泛部署在各种软硬件设备上。例如,浏览器、PDF 阅读器中内嵌有 JavaScript 的执行引擎,很多物联网(Internet of Things, IoT)设备中也部署了 Python、JavaScript 等脚本语言的执行引擎[2]。

脚本语言执行引擎具有代码量庞大且逻辑复杂的特点,这也使得其中难免包含软件缺陷和漏洞。以部署在 iOS 系统 Safari 浏览器中的 JavaScript 执行引擎 JavaScriptCore[3]为例,其包含 42 万行 C/C++代码,定义了近 2 万多个类/结构体。引擎本身的复杂度使得开发者在软件开发过程中难免引入软件缺陷和漏洞。而且,一些语言自身也处于不断演化的状态,因此,引擎中的软件缺陷和漏洞是一个长期存在的问题。

引擎安全漏洞种类多样,主要包括栈溢出、堆溢出、释放后使用和越界访问等常见的内存安全破坏型漏洞,此外也包括类型混淆、竞争条件等逻辑型漏洞等。引擎漏洞具有良好的可利用性,攻击者可以通过编写脚本语言代码,利用引擎漏洞并放大其安全危害,从而对目标机器造成严重侵害。因此,脚本语言执行引擎中的安全漏洞具有很高的安全影响。

鉴于脚本语言执行引擎漏洞的危害性,安全人员一直在尝试通过人工及自动化方法挖掘漏洞,而模糊测试[4]技术正是一种有效的方法。模糊测试方法借助机器算力,自动化地生成大量测试用例,交由被测应用执行。通过观察被测应用在执行中有无异常,如发生崩溃等,测试工具可以发现被测应用中潜藏的软件缺陷。模糊测试是一种有效的漏洞挖掘方法,但将模糊测试应用于脚本语言执行引擎中的漏洞挖掘仍存在一些挑战。首先,脚本语言执行引

① 国家自然科学基金(U1736208, 61902374)资助项目。

② 女,1992 年生,博士生;研究方向:系统与软件安全;E-mail: sunlili@ict.ac.cn。

③ 通信作者,E-mail: wucg@ict.ac.cn。

(收稿日期:2021-09-09)

擎的输入文件需要符合文法规则和语义约束,传统的测试用例生成方法难以满足上述条件。其次,引擎的工作流覆盖多个处理模块,模糊测试需要设计合适的测试用例生成方法和变异策略才能实现对各模块的测试,或是对某一模块的深度测试。此外,面对引擎无穷大的样本输入空间,提升模糊测试的测试效率也是一大挑战。针对上述挑战,目前已有较多工作提出了相应的解决方法,并发展出一系列针对脚本语言执行引擎的模糊测试工具的评价指标。

本文对近年来出现的各种脚本语言执行引擎的模糊测试技术进行了分析及归类,主要贡献如下。

(1)梳理了自 2012 年以来的国内外针对脚本语言执行引擎的模糊测试文献,详细讨论了该领域的研究现状。

(2)总结了脚本语言执行引擎模糊测试工具的常用的评估指标和评估方法。

(3)分别从测试用例生成方式、聚焦的测试模块和采用的研究手段 3 个不同的角度对现有工作进行划分,阐述了该领域所存在的研究问题和现有的解决方案。

(4)总结了脚本语言执行引擎的模糊测试技术当前存在的研究挑战,分析了该领域未来具有发展潜力的研究方向。

1 背景知识

1.1 模糊测试

模糊测试技术利用算力,随机生成大量且多样的测试用例,交由被测应用执行。模糊测试工具观测被测应用的执行过程有无异常,收集可触发被测应用崩溃的测试用例,交由安全分析人员进行分析。依据对被测应用分析程度的不同,模糊测试可分为黑盒、灰盒和白盒测试[5]。黑盒测试无需对被测应用的任何了解,其直接生成随机输入作为测试用例。白盒测试需要全面了解程序的内部结构,并且对所有逻辑路径进行测试。灰盒测试介于黑盒和白盒测试之间,其对被测应用的内部逻辑有部分了解。当前最具代表性的模糊测试工具 AFL[6](American Fuzzy Lop)就使用灰盒测试技术。由于脚本语言执行引擎具有代码量庞大、逻辑复杂的特点,其模糊测试工具大多采用黑盒和灰盒测试。

1.2 脚本语言执行引擎

脚本语言执行引擎负责解释执行脚本语言代码。其通常由解析器、中间表示生成器和解释器模块构成,具体工作流图如图 1 所示。脚本语言执行引擎最前端的是解析器模块,包括词法分析器(Lexer)和语法解析器(Parser)[7]。解析器模块通过词法、语法分析,将脚本语言代码转为抽象语法树(abstract syntax tree, AST)。随后,中间表示生成器将抽象语法树转为中间表示,如字节码。最后,解释器模块负责对字节码逐一解释执行。为了提升执行效率,一些脚本语言执行引擎还引入了即时(just-in-time, JIT)编译器[8]模块。对于解释器模块频繁解释执行的字节码,例如循环体或被频繁调用的函数体,JIT 编译器为其编译生成实现相同语义功能的二进制代码,并存储在内存中。当上述字节码再次执行时,执行引擎会将控制流跳转到相应的二进制代码执行,而无需由解释器解释执行。

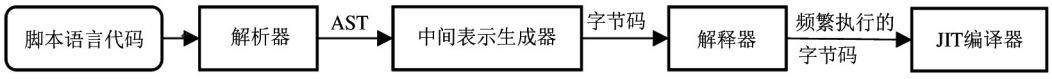


图 1 脚本语言执行引擎的通用工作流程

2 评价指标

随着脚本语言执行引擎的模糊测试技术的发展,越来越多的模糊测试工具相继出现。因此,需要有效的评估方法和评估指标用以衡量测试工具的有效性和先进性。本文总结了近年来脚本语言执行引擎模糊测试工具所使用的评价指标,如表 1 所示。在所有评估指标中,测试工具所挖掘到的漏洞和缺陷数目是最为关键的评价指标。此外,代码覆盖率、语法语义通过率和测试速度也是较为常见的测试指标。

表1给出了一些代表性的脚本语言执行引擎模

表 1 部分脚本语言执行引擎模糊测试工具的评价情况

Fuzzer 工具	时间	缺陷/漏洞发现情况	覆盖率	通过率	测试速度
jsfunfuzz[9]	2007	js:280 +	-	-	-
LangFuzz[10]	2012	js:164,php:20	-	-	-
GramFuzz[11]	2013	js:36	-	-	Y
IFuzzer[12]	2016	js:17(1)	-	-	-
SkyFire[13]	2017	xsl/xml:19(11)	*L*,*F*	-	Y
Fuzzilli[14]	2018	js:47(35)	*E*	-	-
Nautilus[15]	2019	ruby:7(6),php:3,jsc:2,lua:1	*B*	-	-
Superion[16]	2019	xml:11(10),js:23(9)	*L*,*F*	-	Y
Grimoire[17]	2019	php:4	*BB*	-	Y
CodeAlchemist[18]	2019	js:19(5)	-	SYN,SEM	-
DIE[19]	2020	js:48(12)	*E*	SYN,SEM	-
Montage[20]	2020	js:37(3)	-	SYN,SEM	-
Comfort[21]	2021	js:129	-	SYN	-
POLYGLOT[22]	2021	js:26(5),R:7,php:35,sql:27(4),lua:14(9),pascal:8	*E*	SYN,SEM	-
SoFi[23]	2021	js:51(10)	*E*	SYN,SEM	-

糊测试工具的评测情况。其中,“-”表示无相应实验数据,“Y”表示有相应实验数据。“缺陷/漏洞发现情况”列中给出了测试工具所发现的脚本语言执行引擎的缺陷数目,括号中的是 CVE 数目。“覆盖率”列中给出了实验所使用的代码覆盖率评估粒度。其中,*L* 代表行覆盖率,*F* 代表函数覆盖率,*E* 代表边覆盖率,*B* 代表分支覆盖率,*BB* 代表基本块覆盖率。“通过率”列中,SYN 表示实验评测了语法通过率,SEM 表示实验评测了语义通过率。

2.1 漏洞和缺陷数目

测试工具所发现的漏洞和缺陷的数目是评估该工具有效性的最关键的指标。漏洞 CVE 编号[24]和引擎厂商已确认的错误报告编号都可以作为模糊测试工具发现漏洞和缺陷的凭证。在测试工具收集到可触发执行引擎崩溃的测试用例后,研究人员可以向引擎开发团队提交错误报告或漏洞报告。报告需附上可重现该错误的脚本语言代码,即 PoC(proof-of-concept)。之后,引擎开发者会判断该错误是否为软件缺陷或漏洞,并对其进行修复。对于存在安全危害的软件缺陷,引擎厂商或研究人员可为其申请 CVE 编号,并发布漏洞公告。

鉴于脚本语言执行引擎具有代码量庞大、逻辑复杂的特点,为了测试充分,模糊测试工具的测试时间通常以月为单位。测试时间较长的如文献[13],可达 450 d。测试时间最短的如文献[19]仅耗时 3 d。大多数相关工作的测试用时在 30 ~ 90 d,如文献[10,12,16,22]。其中,文献[19]的测试用时之所以仅为 3 d,是因为该系统部署在大规模的机器集群上,集群的机器总核心数目是其他工作所用核心数目的百倍以上。

为了体现测试工具的先进性,研究人员需要与其他测试工具进行测试实验对比。但是,复现上述测试所需的时间代价很大。当前的对比实验通常限制在一个更短的时间内完成,例如文献[10,22]中评估了 24 h 内不同模糊测试工具所发现的引擎缺陷数目。

2.2 代码覆盖率

代码覆盖率[25]是评估模糊测试工具有效性的另一重要指标。高代码覆盖率说明测试工具能够执行到其他工具无法覆盖到的代码。而测试不充分的代码中往往包含软件缺陷,因此,高代码覆盖率也意味着有更多发现软件缺陷的可能性。

代码覆盖率包含多种不同粒度的评估方式:行覆盖率、基本块覆盖率、函数覆盖率、分支覆盖率、边覆盖率和路径覆盖率。行覆盖率是指被测应用在测试过程中所执行到的代码行数占源码总行数的比

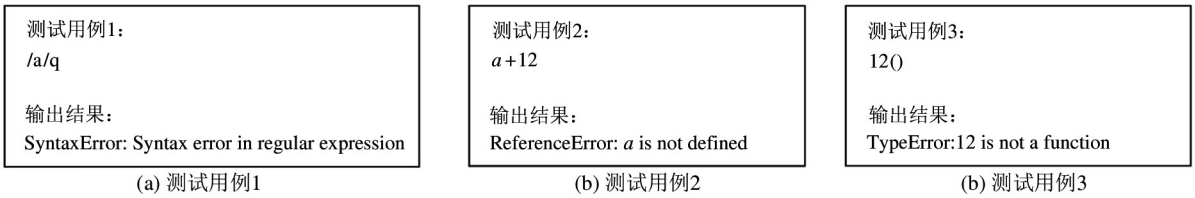
例。其他覆盖率的定义以此类推。在上述覆盖率中,边覆盖率的使用较为常见,除了表 1 中的文献,AFL 使用的也是边覆盖率。由于路径爆炸问题[26],路径覆盖率在实际工作中极少被应用,且通常使用所覆盖到的路径数目替代。虽然代码覆盖率的评估粒度多样,但并没有一个通用的最优选择[27],所以,不同工作按需选取合适的评估方式。虽然用代码覆盖率能否评估测试工具有效性一直存在争议[28],但其仍然是一个不可或缺的评估指标。

收集覆盖率信息需要对编译器进行修改或配置。主流编译器如 Clang[29] 和 GCC[30] 提供了一些命令行选项[31-32],支持在编译源码的同时插桩收集覆盖率信息的代码。此外,研究人员也可以使用定制的编译器来实现覆盖率信息的收集。

2.3 语法语义通过率

语法语义通过率是指能够通过语法和语义检查的测试用例数目占测试用例总数的比例,是评估引擎的模糊测试工具的一个重要指标。较低的语法语义通过率意味着执行引擎的执行流仅仅局限在引擎前端,无法深入引擎后端的代码逻辑。

当测试用例中存在语法错误时,引擎执行到解析器模块便会报出该语法错误并终止执行。图 2(a)展示了一个 JavaScript 的语法错误[33]实例,由于正则表达式中没有为“q”的标志,因此执行引擎检测出语法错误。如果测试用例通过了语法检查,后续的解释器模块还会进行语义检查。对于存在语义错误的情况,引擎也会及时报错并终止执行。图 2(b)和图 2(c)分别展示了 2 种不同的 JavaScript 语义错误实例。其中,图 2(b)中使用了未定义的变量 a,因此执行引擎检测出引用错误[34];图 2(c)中 12 不是函数却用作函数调用,因此执行引擎检测出类型错误[35]。



(a) 测试用例1　(b) 测试用例2　(b) 测试用例3

图 2　JavaScript 语言的语法和语义错误实例

通过解析执行引擎在执行测试用例过程中的输出信息,可以判别出该测试用例是否触发了语法语义错误,以及具体的错误类型。

2.4 测试速度

测试速度是指模糊测试工具在单位时间内生成并执行的测试用例的数量。测试速度并不是一个关键评价指标,因为测试速度的快慢和模糊测试工具的优劣并非正相关。越是简单的变异策略,如随机字节变异,越能带来很高的测试速度。但这种变异方式会导致很低的语法语义通过率,并不能带来测试深度的提升。不过,一些工作(如文献[11]等,见表 1)还是在实验中对该指标进行了评测。

3 方法分类

本文从 3 个角度对现有的脚本语言执行引擎的模糊测试工作进行划分。如图 3 所示,从测试用例生成方式的角度可以分为基于生成和基于变异;从所聚焦的测试模块的角度可以分为针对解析器、针对解释器和针对JIT编译器;从所采用的技术手段

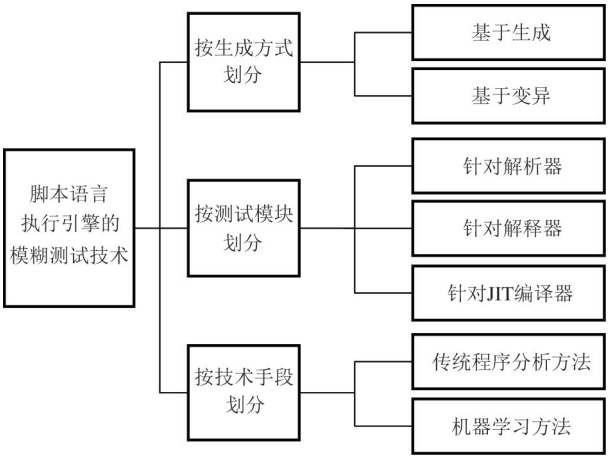


图 3　脚本语言执行引擎模糊测试技术分类

的角度可以分为传统程序分析方法和机器学习方法。

3.1 按生成方式划分

3.1.1 基于生成

基于生成的方法无需初始种子文件,从头开始生成符合语言文法的测试用例。最早的工作是2007 年的 jsfunfuzz[9],其将 JavaScript 代码的文法规则硬编码在程序中,在程序执行中不断推导文法规则,最后生成符合文法的测试用例。该方法能够生成大量多样的 JavaScript 代码,并发现了数百个引擎缺陷。但是,该方法也具有缺乏通用性的缺点,且生成的测试用例难以满足语义约束。GramTest[36] 解决了通用性的问题。对于任何可用巴科斯范式(Backus-Naur form, BNF)表述文法规则的语言,该方法都可以生成符合文法约束的测试用例。具体实现如图 4 所示。图中左侧黑框内是一个使用 BNF 表述的语言文法,其中,N 表示非终结符,T 表示终结符,R 表示该文法所包含的推导规则,S 表示文法推导的起始符号。图中右侧给出了从 PROG 开始的一次随机文法推导过程。推导过程依次选取推导规则(1)→(4)→(5)→(6)→(8),最终得到“a = 1”这个满足文法要求的字符串。这种基于文法推导规则的测试用例生成方法不仅被基于生成的方法[15] 所使用,很多基于变异的工作[10,12,19,22] 也采用该技术来生成变异所需的代码片段。为了进一步提升通用性,文献[17]提出了一种通过测试自动归纳文法规则的方法,从而使测试系统无需语言文法作为输入。

上述方法所得到的测试用例虽然满足文法约束,但难以满足语义约束。测试用例在执行中会触发各种语义错误[37],例如未定义的变量使用错误、非预期的数据类型错误等。因此,一些工作使用比文法规则更高级的概念来指导生成,从而在测试用例生成时考虑到语义信息的维护。FuzzIL[14] 引入了自定义的中间表示语言,其在中间表示的层面进行生成和变异。CodeAlchemist[18] 引入了“代码砖石”这一概念。“代码砖石”是由大量 JavaScript 种子文件做语句级拆解而得到的代码片段。其中不仅包含 JavaScript 语句,还包含语句中所定值和使用的变量及这些变量的类型信息。

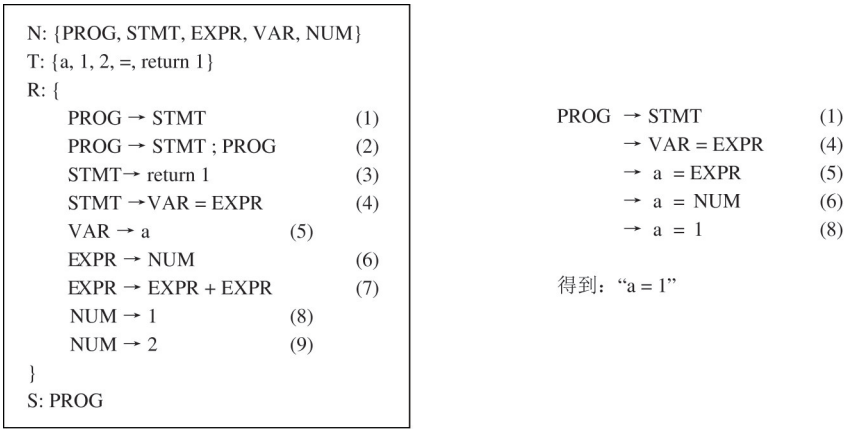


图 4 基于文法推导规则的测试用例生成

3.1.2 基于变异

基于变异的方法需要一批语法语义正确的初始种子作为输入,通过对其进行变异,生成测试用例。为了保证测试用例的语法正确性,基于变异的方法(如文献[10-12,16,19,38])会依据文法,先将种子文件转为抽象语法树(AST),再在语法树节点上做符合语法规则的变异。图 5 展示了在抽象语法树上的语法子树替换变异。图 5 表示的是“a = 6 * (7 + 3)”对应的抽象语法树。通过将虚线框中的 Literal 子树节点替换为另一个相同语法类型的子树节点,可以实现符合语法规则的子树变异。

为了生成符合语义约束的测试用例,POLYGLOT[22] 将语法树进一步转为包含变量作用域、变量类型等语义信息的中间表示,之后对中间表示进行变异。其变异策略不仅考虑到语法正确性,也维

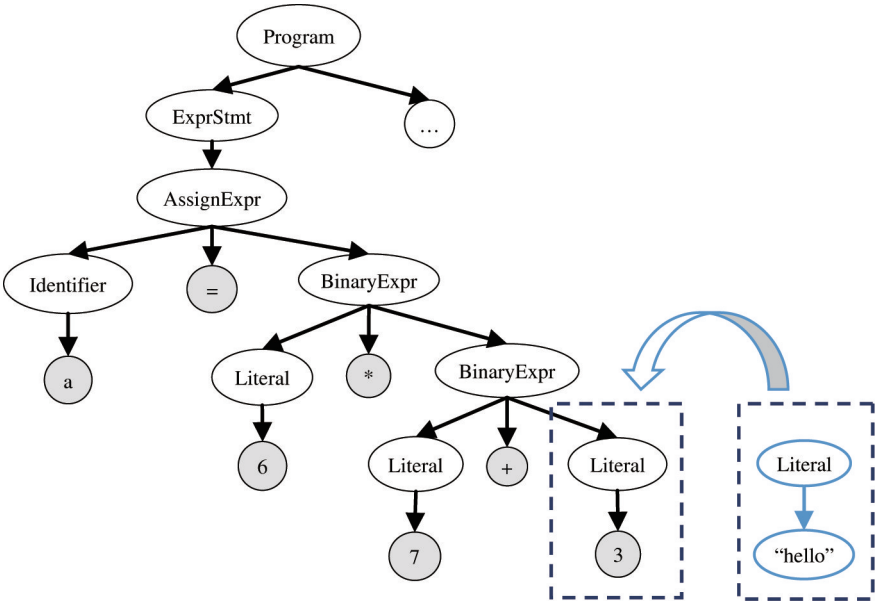


图 5　语法子树替换变异

护了语义正确性。

基于生成的方法和基于变异的方法各有优劣。基于变异的方法所得到的测试用例往往有更高的语法语义通过率,因为其是在原本就语法语义正确的测试用例的基础上做小范围的变异。不过,基于变异的方法难以像基于生成的方法一样,生成一些不符合人工编码习惯,但又满足语法规则的测试用例。此外,基于变异的方法还存在一大限制,即需要大量的初始种子以覆盖各种语法结构、内建对象和内建函数。当前基于变异的方法通常使用执行引擎和语言规范中自带的测试集,以及从网上爬取脚本语言代码,作为初始种子。

3.2　按测试模块划分

3.2.1　针对解析器

解析器模块作为执行引擎最前端的一个模块,也是测试最为充分的一个模块,但这并不意味着该模块中就没有软件缺陷。一些脚本语言自身处于不断演化的状态,需修改解析器以支持新增的语言特性。这些修改可能会引入软件缺陷。另一方面,由于解析器模块所接收的输入通常是符合文法的,反而导致一些错误处理相关的逻辑并未被充分测试。因此,针对解析器的模糊测试既应覆盖解析器中的正常处理逻辑,也应覆盖到各种错误处理逻辑。

文献[15,16]保留了 AFL 中的变异策略,并在此基础上引入了基于文法的化简机制和变异策略。基于文法的变异策略可以生成满足语法约束的测试用例,从而测试解析器中的正常处理逻辑。AFL 原生的比特翻转、字节翻转等变异策略,能够生成大量不合文法、且难以人工构造的测试用例,从而测试解析器的错误处理逻辑。

3.2.2　针对解释器

解释器模块位于解析器和字节码生成器之后。为实现针对解释器模块的充分测试,需要模糊测试技术生成语法语义正确且多样的测试用例。

对于正确性这一目标,文献[9,10,12,15,16]使用文法规则指导测试用例的生成和变异(如 3.1 节所示)。但是,基于文法的生成/变异方法并不能保证语义的正确性。一些工作针对语义正确性提出了相应的解决方法。例如,LangFuzz[10] 使用变异点作用域下可用的变量名,修正变异所引入的未定义标识符。POLYGLOT[22] 将种子文件转为中间表示,该中间表示不仅包含语法节点信息,也包含语义信息(数据的类型和作用域)。因此在变异中可以同时考虑到对两者正确性的维护。文献[39]针对函数调用进行变异,其提出了一种函数参数类型的推断方法,实现了对测试用例的精确变异。文献[23]不仅采用了路径敏感的变量识别方法,还使用了动静态方法实现对变量类型的准确推导。此外,文

献[23]还提出了启发式修复方法,利用测试用例执行时获得的语法或语义出错信息,指导对测试用例进行语法和语义修复。

对于多样性这一目标,大多数工作依靠初始种子来实现生成测试用例的语法语义多样性。文献[23]利用运行时反射信息,获取数据对象所包含的属性和函数,获得语义多样的变异素材。目前,尚未有相关工作提出语法语义多样性的评估方法。不过,模糊测试的关键指标是发现缺陷的数目。所以,也可以通过这一指标来间接评估生成测试用例的多样性。例如文献[23]分别评测了 SoFi _ V(保留了和正确性相关的变异机制)与 SoFi _ D(保留了和多样性相关的变异机制)的缺陷发现数目,以评估变异机制对测试效果的影响。除了运行时反射信息,语言规范也可以指导生成语法语义多样且正确的测试用例。文献[21]利用 JavaScript 语言规范生成针对内建函数的测试用例。其从语言规范中抽取出内建函数潜在的参数类型,并生成相应的实参以触发各种边界条件。

3.2.3 针对 JIT 编译器

与解释器不同,语言规范中并不定义 JIT 编译器的实现标准。不同的引擎厂商有着独有的 JIT 编译优化机制。JIT 编译优化机制复杂多样,一些激进的优化策略在特殊的边界情况下,可能带来安全隐患。针对 JIT 编译器的模糊测试,需要生成的测试用例既可触发 JIT 编译器工作,又可覆盖各种优化策略。

一种简单的解决方法是使用模板。文献[40]的模板中定义了循环体和被循环调用的函数。该代码结构可以触发 JIT 编译器对循环中的函数进行编译优化。同时,通过对函数体内的代码进行变异,可以使 JIT 编译器的输入具有多样性。从实验结果上看,该方法所生成的测试用例虽具有很高的 JIT 成功率和 JIT 覆盖率,却难以触发 JIT 编译器中的软件缺陷。一些工作虽并非针对 JIT 编译器进行测试,但在 JIT 编译器的漏洞挖掘上却取得了很好的效果。例如,FuzzIL[14]构造出的测试用例具有复杂且不合常规的数据流和控制流,在挖掘 JIT 编译器的软件缺陷和漏洞中也有很好的效果[41]。DIE[19]使用漏洞 PoC 和回归测试作为初始种子,并通过“方面-保存”的变异策略,使变异不会改变原有种子文件的控制流结构和数据类型信息。DIE 的变异策略可以使原本可触发 JIT 编译优化的循环体或函数在迭代变异中得到保留,其中可以触发某些特殊优化策略的代码也有一定概率不被破坏,其实验结果也证实了 DIE 能够发现 JIT 编译器中的软件缺陷和漏洞。

3.3 按技术手段划分

3.3.1 传统程序分析方法

传统的程序分析方法在脚本语言测试用例的自动生成方面得到了普遍使用。其中,最为常用的技术是使用抽象语法树表示测试用例,例如文献[10-13,15,16,19]等。该方法借助语法解析器,将测试用例由字符串转成抽象语法树进行存储。这既可以实现对测试用例的语法正确性检测,又为后续的语法子树变异提供了素材。抽象语法树虽然提供了语法信息,但缺少语义信息。文献[19]在抽象语法树的基础上,对测试用例进行变量的作用域分析和数据类型分析。作用域分析明确了各个变量的生命周期,数据类型分析明确了变量的类型。它们都为后续的变异或生成策略提供了合适的数据素材。此外,构造中间表示也是一种常用的程序分析方法,常见的中间表示如三地址码。使用中间表示来存储测试用例,可以隐去测试用例中复杂的语法结构,更适合进行语义分析。文献[14,22]便是中间表示的典型应用。使用程序分析方法对测试用例进行分析,具有准确性高、可操作性强的优点。但该方法也具有依赖专家经验和难以获得通用性的缺点。

3.3.2 机器学习方法

机器学习作为当下非常热门的研究方向,其解决方法在很多领域都得到了成功应用。文献[13,42]利用统计学习方法指导测试用例生成。文献[42]从大量样本中学习出文法推导规则的概率模型,用来指导生成和样本相似的语言样例,从而确保语法语义的正确性。文献[13]进一步学习出上下文敏感的文法推导规则的概率模型,在该模型的指导下,可以生成语法语义正确且在样本中较为罕见的语言样例。文献[20]和文献[21]分别提出了使用神经

网络语言模型和 GPT-2 语言生成模型自动生成的 JavaScript 代码的方法。机器学习方法的优点在于具有良好的通用性、也无需应用领域相关的先验知识,缺点在于需要大规模的训练数据,且生成代码的语法语义正确性不够高。

4 研究不足和未来趋势

依据对现有研究成果的分析,脚本语言执行引擎的模糊测试技术存在以下研究挑战。

(1)反馈机制的设计

反馈机制对模糊测试工具的漏洞挖掘能力和执行效率都有着至关重要的影响。当前大多数模糊测试技术[6,43-44]所使用的反馈机制都是代码覆盖率反馈。而程序执行状态的另一组成部分——数据,却被长期忽视。不过,在最新的模糊测试研究中,一些工作[45-46]开始将数据状态引入反馈机制,并取得了不错的漏洞挖掘效果。脚本语言执行引擎内部有着非常丰富的数据状态。设计合适的数据相关的反馈机制指导脚本语言执行引擎的模糊测试,是一个很有前景的研究方向。

(2)语言标准指导的模糊测试

为了确保不同引擎对脚本语言具有相同的解释执行结果,引擎厂商原则上应严格遵循语言标准,实现解释器。语言标准作为脚本语言执行引擎实现的依据,也可以用于指导针对脚本语言执行引擎的模糊测试。但当前仅有少数工作利用语言标准指导测试,且利用程度并不充分。随着自然语言处理、知识图谱、知识推理等技术的发展,这些技术可以从语言标准中抽取出丰富的“基本事实”(ground truth),用以指导模糊测试。

(3)语义多样性

现有工作在生成脚本语言测试用例时,主要关注语法语义正确性,而缺少对语义多样性的关注。当前的技术主要依靠大量的初始种子文件来保障语义多样性。如果种子文件中缺少某种内建函数的调用,就无法生成包含该内建函数的测试用例。针对生成测试用例的语义多样性,设计合适的评估方法和变异机制,也是一个很有意义的研究方向。

5 结 论

脚本语言执行引擎的安全漏洞具有很高的安全危害和利用价值。通过模糊测试技术挖掘脚本语言执行引擎的软件缺陷和漏洞,是近年来的一个研究热点。本文围绕脚本语言执行引擎的模糊测试技术的研究现状进行了梳理和分类,总结了当前的评估指标,并从不同角度分析了现有工作关注的研究难点和采取的解决方案。此外,本文讨论并总结了脚本语言执行引擎模糊测试技术面临的挑战和发展趋势,为后续研究提供了重要的参考价值。

参考文献

[1] OUSTERHOUT J K. Scripting: higher level programming for the 21st century[J]. *Computer*, 1998, 31(3): 23-30

[2] GitHub. Next generation AWS IoT Client SDK for Node.js using the AWS Common Runtime[EB/OL]. https://github.com/aws/aws-iot-device-sdk-js-v2: GitHub, [2021-09-09]

[3] Trac. JavaScriptCore[EB/OL]. https://trac.webkit.org/wiki/JavaScriptCore: Trac, [2021-09-09]

[4] BOEHME M, CADAR C, ROYCHOUDHURY A. Fuzzing: challenges and reflections[J]. *IEEE Software*, 2021, 38(3): 79-86

[5] ACHARYA S, PANDYA V. Bridge between black box and white box-gray box testing technique[J]. *International Journal of Electronics and Computer Science Engineering*, 2012, 2(1): 175-185

[6] Lcamtuf. American Fuzzy Lop (2.52b)[EB/OL]. https://lcamtuf.coredump.cx/afl/:Lcamtuf, [2021-09-09]

[7] NESTERUK D. Interpreter[M]//Berkeley: Design Patterns in. NET Core 3. Apress, 2020: 227-239

[8] AYCOCK J. A brief history of just-in-time[J]. *ACM Computing Surveys (CSUR)*, 2003, 35(2): 97-113

[9] RUDERMAN J. Introducing jsfunfuzz[EB/OL]. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz: Word Press, (2007-08-02),[2021-09-09]

[10] HOLLER C, HERZIG K, ZELLER A. Fuzzing with code fragments[C]//The 21st USENIX Security Symposium (USENIX Security 12), Bellevue, USA, 2012: 445-458

[11] GUO T, ZHANG P, WANG X, et al. Gramfuzz: fuzzing

testing of web browsers based on grammar analysis and structural mutation[C]//2013 2nd International Conference on Informatics and Applications (ICIA), Lodz, Poland, 2013: 212-215

[12] VEGGALAM S, RAWAT S, HALLER I, et al. Ifuzzer: an evolutionary interpreter fuzzer using genetic programming[C]//European Symposium on Research in Computer Security, Heraklion, Greece, 2016: 581-601

[13] WANG J, CHEN B, WEI L, et al. Skyfire: data-driven seed generation for fuzzing[C]//2017 IEEE Symposium on Security and Privacy, San Jose, USA, 2017: 579-594

[14] GROB S. FuzzIL: coverage guided fuzzing for javascript engines[D]. Braunschweig: TU Braunschweig, 2018

[15] ASCHERMANN C, FRASSETTO T, HOLZ T, et al. NAUTILUS: fishing for deep bugs with grammars[C]// Network and Distributed Systems Security, San Diego, USA, 2019:1-15

[16] WANG J, CHEN B, WEI L, et al. Superion: grammar-aware greybox fuzzing[C]//2019 IEEE/ACM 41st International Conference on Software Engineering, Montreal, Lanada, 2019: 724-735

[17] BLAZYTKO T, BISHOP M, ASCHERMANN C, et al. GRIMOIRE: synthesizing structure while fuzzing[C]// The 28th USENIX Security Symposium, Santa Clara, USA, 2019:1985-2002

[18] HAN H S, OH D H, CHA S K. CodeAlchemist: semantics-aware code generation to find vulnerabilities in JavaScript engines[C]//Network and Distributed Systems Security, San Diego, USA, 2019:1-15

[19] PARK S, XU W, YUN I, et al. Fuzzing javascript engines with aspect-preserving mutation[C]//2020 IEEE Symposium on Security and Privacy, San Francisco, USA, 2020: 1629-1642

[20] LEE S, HAN H S, CHA S K, et al. Montage: a neural network language model-guided javascript engine fuzzer [C]//The 29th USENIX Security Symposium, Boston, USA, 2020: 2613-2630

[21] YE G, TANG Z, TAN S H, et al. Automated conformance testing for JavaScript engines via deep compiler fuzzing[C]//Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual, 2021: 435-450

[22] CHEN Y, ZHONG R, HU H, et al. One engine to fuzz'em all: generic language processor testing with semantic validation[C]//Proceedings of the 42nd IEEE Symposium on Security and Privacy, San Francisco, USA, 2021: 642-658

[23] HE X Y, XIE X F, LI Y K, et al. SoFi: Reflection-augmented fuzzing for JavaScript engines[C]//ACM Conference on Computer and Communications Security, Virtual, 2021:2229-2242

[24] CVE. CVE[EB/OL]. https://cve.mitre.org/sitemap.html:MITRE, [2021-09-09]

[25] GOPINATH R, JENSEN C, GROCE A. Code coverage for suite evaluation by developers[C]//Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 2014: 72-82

[26] CADAR C, GANESH V, PAWLOWSKI P M, et al. EXE: automatically generating inputs of death[J]. *ACM Transactions on Information and System Security (TISSEC)*, 2008, 12(2): 1-38

[27] WANG J, DUAN Y, SONG W, et al. Be sensitive and collaborative: analyzing impact of coverage metrics in greybox fuzzing[C]//The 22nd International Symposium on Research in Attacks, Intrusions and Defenses, Beijing, China, 2019: 1-15

[28] INOZEMTSEVA L, HOLMES R. Coverage is not strongly correlated with test suite effectiveness[C]//Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 2014: 435-445

[29] Clang. Clang: a C language family frontend for LLVM [EB/OL]. https://clang.llvm.org/:Clang team,[2021-09-09]

[30] GCC. GCC, the GNU compiler collection[EB/OL]. https://gcc.gnu.org/:GCC team,[2021-09-09]

[31] Clang. Clang 13 documentation: source-based code coverage[EB/OL]. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html:Clang team,[2021-09-09]

[32] GCC. Program instrumentation options[EB/OL]. https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/Instrumentation-Options.html:GCC team,[2021-09-09]

[33] Mozilla. Syntax error[EB/OL]. https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError:Mozilla,[2021-09-09]

[34] Mozilla. Reference error[EB/OL]. https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/

Global _ Objects/ReferenceError:Mozilla,[2021-09-09]
[35] Mozilla. Type error[EB/OL]. https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global _ Objects/TypeError:Mozilla,[2021-09-09]
[36] GitHub. GramTest[EB/OL]. https://github.com/code-lion/gramtest:GitHub,[2021-09-09]
[37] Mozilla. Error[EB/OL]. https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global _ Objects/Error:Mozilla,[2021-09-09]
[38] 王聪冲,甘水滔,王晓锋. 子树类型敏感的 JavaScript 引擎灰盒测试技术[J]. 信息安全学报, 2021,6(4):119-131
[39] 曹帅. 基于类型推断的 JavaScript 引擎模糊测试方法研究[D]. 西安:西北大学信息科学与技术学院, 2020
[40] 王越,孙亮,王轶骏,薛质. 一种针对 JavaScript 引擎 JIT 编译器的模糊测试方法[J]. 通信技术,2021,54(1):175-180
[41] GitHub. Bug Showcase[EB/OL]. https://github.com/googleprojectzero/fuzzilli#bug-showcase:GitHub,[2021-09-09]
[42] PATRA J, PRADEL M. Learning to fuzz: application-independent fuzz testing with probabilistic, generative models of input data, TUD-CS-2016-14664[R]. *TU Darmstadt, Department of Computer Science*, Technical Report, 2016
[43] PHAM V T, BÖHME M, SANTOSA A E, et al. Smart greybox fuzzing[J]. *IEEE Transactions on Software Engineering*, 2021, 47(9):1980-1997
[44] BÖHME M, PHAM V T, ROYCHOUDHURY A. Coverage-based greybox fuzzing as markov chain[J]. *IEEE Transactions on Software Engineering*, 2017, 45(5):489-506
[45] GAN S, ZHANG C, CHEN P, et al. GREYONE: data flow sensitive fuzzing[C]// The 29th USENIX Security Symposium, Boston, USA, 2020: 2577-2594
[46] ASCHERMANN C, SCHUMILO S, ABBASI A, et al. Ijon: exploring deep state spaces via fuzzing[C]// 2020 IEEE Symposium on Security and Privacy, San Francisco, USA, 2020: 1597-1612

Survey of fuzzing for scripting language execution engines

SUN Lili, WU Chenggang, XU Jiali, ZHANG Peihua, TANG Bowen, XIE Mengyao
(State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)
(School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100190)

Abstract

The scripting language, being an interpreted language, needs to be dynamically interpreted by the scripting language execution engine. Because scripting languages are widely used in many fields, their execution engines have also been widely deployed on various platforms. As a result, security vulnerabilities in scripting language execution engines often have a high-security impact. Fuzzing, as an effective automatic vulnerability finding method, also plays an important role in mining the defects and vulnerabilities in scripting language execution engines. This paper systematically summarizes recent domestic and foreign research in this field. First, it introduces the fundamental concepts of fuzzing and scripting language execution engines; then it lists the evaluation indicators of the existing fuzzing work for scripting language execution engines; next, it categorizes the fuzzing work for scripting language execution engines and expounds on the research problems and solutions in this field; finally, it puts forward the future potential research directions according to the inadequacies and research trends of current work.

Key words: scripting language engine, fuzzing, vulnerability finding, software flaw detection