HTTPFuzzer: 强化学习引导的 Web 服务器程序灰盒模糊测试方法

陈 乾, 洪 征, 江 川, 张国敏, 秦素娟, 古津榜, 崔 帅

中国人民解放军陆军工程大学 指挥控制工程学院 南京 中国 210000

摘要 Web 服务器作为互联网架构的核心组成部分, 承载着海量的数据交互与业务逻辑处理, 其安全性直接关系到整个信息系统的稳固。Web 服务器程序作为关键组件, 受到了攻击者的广泛关注。因此, 对 Web 服务器程序的漏洞进行挖掘和修复至关重要。基于变异的灰盒模糊测试方法广泛应用于 Web 服务器程序的漏洞挖掘中, 此类方法通常以真实的 HTTP 报文为“种子”, 对“种子”进行变异以产生测试用例。测试用例的质量取决于“种子”变异位置的选择和变异算子的调度。现有方法在变异位置的选择和变异算子的调度方面主要遵循预设规则, 盲目地遵循预设规则使变异缺乏针对性, 效率较低, 导致大量无效用例的产生, 降低了模糊测试效率。针对上述问题, 提出了 HTTPFuzzer, 一种强化学习引导的 Web 服务器程序灰盒模糊测试方法。HTTPFuzzer 对种子的变异分为两个阶段。第一个阶段是变异位置探索阶段, 首先对种子进行分段, 将对各个段的探索转化成多臂老虎机问题。在对段的探索过程中, 依据代码覆盖率和测试目标的反馈报文, 筛选出使测试目标正常响应又能提高代码覆盖率的段作为变异域。第二个阶段是基于 Q 学习的变异算子调度阶段, 这个阶段以提升代码覆盖率为目标, 动态调整变异算子调度策略, 使模糊器根据不同的变异域选择较优的变异算子实施变异。实验表明, HTTPFuzzer 具备产生高质量测试用例的能力, 对比基准方法能够在较短时间内覆盖更多的程序路径, 并且能够较为快速地触发崩溃, 发现漏洞。

关键词 网络安全; 强化学习; Web 服务器; 协议模糊测试; 变异算子

中图法分类号 TP393.08 DOI 号 10.19363/J.cnki.cn10-1380/tn.2026.03.09

HTTPFuzzer: Reinforcement Learning Guided Greybox Fuzzing for Web Server Programs

CHEN Qian, HONG Zheng, JIANG Chuan, ZHANG Guomin, QIN Sujuan, GU Jinbang, CUI Shuai

Command and Control Engineering College, Army Engineering University of PLA, Nanjing 210000, China

Abstract In the realm of Internet architecture, Web servers function as pivotal elements, facilitating extensive data interchange and intricate business logic execution. The security posture of these servers is intimately tied to the overall stability and resilience of the information systems. Web server programs, as critical components, have received extensive attention from attackers. Consequently, the significance of diligently exploring and promptly remediating vulnerabilities within Web server programs cannot be overstated. The mutation-based greybox fuzzing method is widely used in vulnerability mining of Web server programs. This kind of method usually takes real HTTP messages as "seeds" and mutates the "seeds" to generate test cases. The quality of the test cases depends on the selection of the mutation locations and the scheduling of the mutation operators. Existing methods mainly follow the preset rules in the selection of mutation location and the scheduling of mutation operators. Blindly following the preset rules renders the mutation less targeted and less efficient, resulting in many invalid test cases, which affects the efficiency of fuzzing. To address the aforementioned problems, HTTPFuzzer, a reinforcement learning guided greybox fuzzing method for Web server programs is proposed. HTTPFuzzer's mutation process is partitioned into two stages. The first stage is the mutation location exploration, where the seeds are segmented first, and the exploration of each segment is transformed into a multi-armed bandit machine problem. During the segment exploration process, according to the code coverage and the responses of the test target, the segments that can make the test target respond correctly and improve the code coverage is selected as the mutation-regions. The second stage is the mutation operator scheduling stage based on Q-learning. This stage aims to improve the code coverage, dynamically adjusts the mutation operator scheduling strategy, and the fuzzer selects better mutation operators according to different mutation-regions. Experimental results demonstrate that HTTPFuzzer can produce high-quality test cases, surpassing benchmark methods by covering more execution paths within a shorter time frame. Furthermore, it is capable of triggering crashes and uncovering potential vulnerabilities in a shorter time.

通讯作者: 洪征, 博士, 副教授, Email: hz5215@163.com。

本课题得到国家重点研发计划(No. 2019YFB2101704)资助。

收稿日期: 2024-09-03; 修改日期: 2024-12-09; 定稿日期: 2026-01-26

Key words network security; reinforcement learning; Web servers; protocol fuzzing test; fuzzing operators

1 引言

Web 服务器程序作为 Web 应用的载体, 承载着互联网上的众多服务与数据交互, 随着 Web 服务架构的日益复杂, 其面临的安全威胁也急剧增加, 成为网络攻击的重要目标[1]。对 Web 服务器程序进行有效的漏洞挖掘变得尤为重要。

以 AFL[2]为代表灰盒模糊测试方法在漏洞挖掘领域得到了广泛应用, 其变体 AFLnwe[3], AFLNet[4]以及 StateAFL[5]经配置均可有效地应用于 Web 服务器程序的漏洞挖掘中。此类方法属于基于变异的模糊测试方法,不需要测试者预先对协议格式进行定义, 在变异算子的作用下直接对“种子”进行变异以产生测试用例实施测试。应用于 Web 服务器程序的模糊测试时, “种子”往往为真实的 HTTP 报文, 模糊器通过对报文进行位翻转、特殊字符插入等变异操作, 产生畸形的 HTTP 数据作为用例展开测试。

基于变异的方法通常利用变异算子对“种子”的特定位置进行变异, 变异算子决定了测试用例的形态, 即在原有“种子”的基础上如何操作以产生畸形数据作为测试用例[6]。然而主流方法对变异位置的选择和变异算子的调度都具有盲目性, 降低了测试效率。例如, AFL 及其扩展工具, 如 AFLnwe、AFLNet, 往往基于预设规则选择变异位置和变异算子, 其中包含了大量的顺序操作和随机操作, 导致无效用例的产生。理想状况下, 对 Web 服务器程序的模糊测试, 测试用例应遵循 HTTP 报文的格式规范, 但以 AFL 为代表的模糊测试工具并没有充分考虑变异位置的选择和种子调度策略的合理使用, 往往难以产生理想的测试用例。以对 GET 请求的请求头变异为例, 请求头为“GET /url/index.html HTTP/1.1\r\n”。顺序选择变异位置的情况下, 首先会对“GET”进行变异, 然而“GET”代表了请求方法, 是不可变的, 无论对“GET”字段如何变异, 只能使程序进入异常处理流程, 难以深入功能性代码进行探索。当变异“GET”后的空格位置时, 由于 HTTP 协议中空格是字段之间的分割符, 属于固定的格式符号, 对空格进行变异所产生的测试用例都属于无效用例。同样的, 在后续变异中对标识请求头结束的“\r\n”进行操作, 也会破坏 HTTP 报文格式, 产生无效的测试用例。因此, 为了提高测试效率, 需要对变异位置进行探索和分析, 针对性地选定位置进行变异。

变异算子在变异过程中也发挥着重要作用, 不同的变异算子在触发新的程序路径方面往往有不同的效果, 变异算子应基于变异算子的历史表现进行调度[7]。在基于变异的方法中, 通常将能够引起路径覆盖率提升的测试用例视为“有趣”的测试用例。变异算子有效性可以通过在变异算子作用下产生的“有趣”测试用例的数量来反映。盲目地对变异算子进行调度可能会使计算资源耗费在低效的变异操作上, 降低模糊测试效率。此外, 本文通过测试实验发现, 在对 Web 服务器程序进行模糊测试时, 不同的变异算子在作用于不同字段时, 有效性也有差异。因此需要进行合理设计, 使模糊器针对不同的变异位置调度高效的变异算子进行变异操作, 以提升变异效果, 进而提升模糊测试整体效率。

近年来, 强化学习在决策优化问题中应用广泛。

在强化学习的框架下, 智能体(Agent)通过与环境(Environment)进行交互感知环境的变化, 并在此基础上自动优化智能体的行为策略[8]。强化学习适用于智能体进行行动选择的许多场景。在对 Web 服务器程序的模糊测试中, 模糊器可以被视作智能体, 目标 Web 服务器程序可以视为环境。模糊器通过对 HTTP 报文进行变异得到测试用例, 通过发送测试用例与测试目标进行交互。所收集到的测试路径覆盖信息、返回报文、崩溃信息等都可以视作环境的变化。模糊器对 HTTP 报文的哪些位置进行变异以及调度何种变异算子进行变异均是模糊器可以选择的行为。结合强化学习, 可以使模糊器在与测试目标的交互过程中优化变异位置的选择和变异算子的调度。

基于上述分析, 本文提出了一种强化学习引导的 Web 服务器程序模糊测试方法 HTTPFuzzer。HTTPFuzzer 将模糊测试的变异划分为两个阶段。第一个阶段主要进行变异位置的探索。该阶段将变异位置探索建模为多臂老虎机(Multi-Armed Bandit, MAB)问题[9], 首先将 HTTP 报文按照特殊符号分割为多个段。接着对每个段进行变异, 该阶段变异算子主要起到破坏原有文本形态的作用, 根据覆盖率提升情况和响应报文等指标对每个段进行打分, 相应计算每个段的 Beta 分布值以衡量段的变异价值。Beta 分布值越高代表当前段变异的价值越高。在预定次数的探索后, 将 Beta 值低于规定值的段标记为不可变域, 其他段标记为可变域。第二个阶段将基于 Q 学习(Q-Learning)[10]针对可变域进行变异算子的调度。此阶段需要构建价值表, 在价值表中, 可变域作为行, 变异算子作为列, 表中的条目代表对某个域

调用某个变异算子的价值。模糊测试过程中根据路径覆盖情况动态更新价值表。测试过程中将根据价值表对变异域进行选择同时对变异算子进行调度。测试将聚焦于具有较高价值的变异域, 同时调度更有效的变异算子实施变异。本文贡献可概括如下:

1) 提出了一种强化学习引导的 Web 服务器程序模糊测试方法, HTTPFuzzer。方法融合了多臂老虎机和无模型强化学习两种策略, 使模糊测试自动化地探索变异位置和筛选出能够实施变异的变异域, 进而选择变异域并对变异算子进行合理调度。方法提高了测试用例的质量, 提升了测试效率。

2) 提出一种基于多臂老虎机问题框架的变异位置探索方法。将报文划分成多个段后, 对每个段进行变异并发送至测试目标, 根据测试目标的反馈对每个段进行打分并计算 Beta 分布值, 排除不适合实施变异的不可变域, 促使模糊器自动探索变异位置, 提高测试用例质量。

3) 提出一种基于 Q-Learning 的变异算子调度方法。方法基于变异域和变异算子建立 Q 值表, 随着模糊测试的推进对 Q 值表进行更新。模糊器以 Q 值表为基础选择变异域并相应地对变异算子进行调度, 提高了变异效率。

4) HTTPFuzzer 的原型系统采用了 C 语言与 Python 语言混合开发。C 语言专注于实现底层功能, 如对 Web 服务器程序进行状态监控及共享内存的管理, 保证系统的稳定性和运行效率。Python 语言具有丰富的库支持及快速的原型开发能力, 被用于构建模糊器, 便于功能扩展和系统维护。

2 背景和相关工作

基于变异的模糊测试方法通过对种子进行变异产生测试用例。近年来, 很多研究人员将灰盒模糊测试和基于变异的方法相结合, 在模糊测试开始前对目标程序进行插桩, 测试过程中通过插入的桩代码推断模糊测试的覆盖率, 覆盖率越高代表着发现漏洞的概率越高。AFL[2]、HonggFUZZ[11]以及 Libfuzzer[12]是此类方法的代表, AFL 的变体 AFLnwe[3], AFLNW[13]通过修改测试用例的投喂方式以适配网络服务程序的交互, 在 Web 服务器程序的模糊测试领域应用广泛。AFLNet[4]、StateAFL[5]、AFLNeTrans[14]针对有状态协议设计, 虽然可以通过手动配置, 删除协议状态学习等操作, 对无状态协议进行适配用于 Web 服务器程序的模糊测试, 但受到预设规则的限制, 往往盲目地进行变异操作, 影响模糊测试效率。

算法 1.AFL 的变异算子调度.

Algorithm 1. Operator Scheduling for AFL

Input: Input seeds: *Seeds*; Target Program: *Target*.

Output: Error queue: *Q_Error*; Seeds queue: *Q_seeds*.

```
1.  Initialize Q_seeds
2.  Initialize Q_error
3.  WHILE TRUE DO
4.    Select a seed S from Q_seeds
5.    FOR opt IN fuzz_operators DO
6.      FOR i IN range(len(S)) DO
7.        S' = Mutate(S, opt, i)
8.        Send S' to Target
9.        IF ERROR occured DO
10.         Update(Q_Error)
11.       IF S' is interesting DO
12.         Update(Q_Seed)
```

AFL 模糊测试引擎的变异算子调度如算法 1 所示。进入测试循环前需要初始化种子队列 *Q_seeds* 和错误队列 *Q_Error*。种子队列用于存放种子, 错误队列用于存储引起程序异常的测试用例。测试中, 系统首先从种子队列中选取一个种子 *S*, 接着根据预定义的顺序从中选择变异算子, 对种子的每个位按顺序进行变异。将变异后产生的测试用例 *S'*发送至测试目标, 如果发生崩溃或超时等错误就对错误队列 *Q_Error* 进行更新, 若当前测试用例成功触发了新的程序路径, 则对种子队列 *Q_seeds* 进行更新, 使测试用例可以在后续的测试中得到复用, 提高触发新路径的可能。如算法 1 的第 5 行和第 6 行所示, AFL 对于变异算子进行顺序调度, 对种子的变异也是从第一位到最后一位按序变异。一方面, 按序的变异并没有考虑到程序仅接收特定格式的数据, 对每个位置依次变异会产生大量无效用例。另一方面, 在变异算子的调度方面依照特定顺序, 存在盲目性。对于每个种子而言, 方法盲目地对不同变异算子进行尝试, 影响测试效率。

MOPT[7]、AMSFuzz[15]、文献[16]和文献[17]对变异算子的调度问题进行了研究。模糊测试过程中变异算子发挥着不同的效能, 应该基于变异算子在模糊测试过程中的历史表现进行调度[7]。MOPT 将粒子群优化算法应用于变异算子的选择, 以代码覆盖率作为优化指标, 提升模糊测试的效率。AMSFuzz 将变异算子的调度问题建模为多臂老虎机问题, 若当前变异算子引起了覆盖率的提升或导致异常则视为一次成功的尝试。在多轮迭代后模糊器会倾向于调度那些具有高得分的变异算子, 这种方法也有效地提高了模糊测试效率。文献[16]聚焦于并行模糊测

试, 分析变异算子对目标程序的作用效果, 并对优势变异算子进行调度, 提高并行模糊测试的整体运行效果。文献[17]尝试将深度强化学习运用于变异算子的调度过程。但是, 在模糊测试过程中加入深度神经网络会给系统带来资源消耗方面的压力, 降低模糊测试效率。

变异位置的选择同样会对模糊测试造成影响。首先, 对于 Web 服务器程序的模糊测试而言, 测试用例以 HTTP 报文为基础变异得到, 但如算法 1 的第 6 行所示, 传统做法对每个位置顺序进行变异, 但并不是每个位置都是可变的。一些研究人员将程序分析技术引入模糊测试中辅助变异位置的判断。代表性的工作有 VUzzer[18]、REDQUEEN[19]和 GreyOne[20]。VUzzer 通过污点分析技术提取数据流特征, 并利用它们推断输入中哪些位置的数据将作为分支约束条件, 进而在变异过程中针对性地选择变异位置, 避免对不太可能引发新的程序行为的输入位置进行变异。REDQUEEN 在程序执行时追踪并 hook 所有的比较指令以提取参数, 判断对种子特定位置的修改是否会触发新的执行路径。如果对某个位置的修改会引起执行路径信息的变化, 则对相应位置进行变异。GreyOne 通过模糊测试驱动污点分析, 基于数据流特性进行变异位置的分析。GreyOne 关注程序分支中变量和种子变异位置的关系, 以判断种子中哪些位置是可变的。如果种子的某个位置发生变化, 导致程序中某个条件判断所使用的变量随之变化, 那么认为这个条件判断分支受到了污点影响, 种子的相应位置是可变的。上述方法涉及程序分析, 会给系统带来较大的负担, 影响工具的实际使用。而且这些方法仅关注变异位置的选择, 并没有考虑变异算子的调度对模糊测试的影响。

本文通过实验分析发现, 对于不同的变异位置, 不同变异算子的有效性有很大差异。具体而言, 不同的变异算子作用于 HTTP 报文的不同段时, 产生“有趣”测试用例的比例也不相同。本文对 Lighttpd 1.4.66、1.4.67、1.4.51、1.4.52 四个版本的程序以及 Thttpd 2.27、2.28 和 2.29 三个版本的程序开展实验, 对每个程序实例分别进行 4 组长达 1 小时的模糊测试。实验中对图 1 报文中的标记字段进行变异, 对测试结果取均值进行有趣测试用例的占比计算。从表 1 的实验结果中可以看出, 不同字段在不同变异算子的作用下, 所产生的有趣测试用例的比例有明显差异。

```
GET /index.html HTTP/1.1
Host: 127.0.0.1:8080
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
95.0.4638.54 Safari/537.36
Accept: text/html,application/xhtml+xml;q=0.9,image/avif,image/webp,*/*;q=0.8,application/signed-
exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: zh-GN,zh;q=0.9
Connection: close
```

图 1 HTTP 报文示例

Figure 1 An example of HTTP message

在表 1 中, 对 URL 值和 Host 值进行变异, 20%左右的有趣测试用例通过调度 Arithmetic 变异算子产生。对 User-agent 值和 Accept 值的变异, 约有 50%的有趣测试用例通过调度 Arithmetic 算子产生。而对于 Upgrade-Insecure-Requests 值而言, 仅有 3%的有趣测试用例在 Arithmetic 算子作用下产生。再看 Havoc 算子, 对 URL 字段和 Host 字段进行变异, 有趣测试用例中 50%左右在 Havoc 算子作用下变异而来。而对于 Upgrade-Insecure-Requests 值而言该比例达到了 89%, 在 User-agent 值和 Accept 值的变异中该指标仅有 10%。对于变异算子 Bitflip, 按照变异粒度进一步划分 Bitflip1/1、Bitflip2/1……Bitflip32/8 等变异算子, 在不同变异算子作用于各个字段时, 效果也存在差距, Bitflip1/1、Bitflip2/1 变异算子相对而言, 表现较优。

基于上述分析, 本文提出一种强化学习引导的 Web 服务器程序灰盒模糊测试方法。在对 Web 服务器程序进行模糊测试的过程中融入强化学习策略, 首先在测试过程中基于程序覆盖率和返回报文高效进行变异位置的分析。其次, 使模糊器能够在调度变异算子时考虑不同的变异位置, 并根据测试中积累的经验, 依据变异位置选择效果较好的变异算子, 提升模糊测试效率。

3 HTTPFuzzer 的设计

本部分介绍 HTTPFuzzer 的设计原理与架构。首先分析 HTTPFuzzer 的总体设计, 论述其主要工作原理。随后, 聚焦 HTTPFuzzer 的两个核心模块——基于 MAB 的变异位置探索模块以及基于 Q-Learning 的变异算子调度模块, 对两个模块的设计进行分析。

表 1 不同变异算子的效果

Table 1 Efficiency of different mutation operators

Mutation field 变异域	Mutation Operator 变异算子	Percentage of Interesting Test Cases 有趣测试用例的比例	
URL address URL 地址	Bitflip 1/1	17%	33%
	Bitflip 2/1	6%	
	Bitflip 4/1	4%	
	Bitflip 16/8	2%	
	Bitflip 32/8	4%	
	Arithmetic 8/8	21%	22%
	Arithmetic 16/8	1%	
	Havoc	45%	45%
Host address Host 地址	Bitflip 1/1	17.7%	25.3%
	Bitflip 2/1	4%	
	Bitflip 4/1	3%	
	Bitflip 8/8	0.3%	
	Bitflip 16/8	0.3%	
	Arithmetic 8/8	18.7%	19.7%
	Arithmetic 16/8	1%	
	Havoc	55%	55%
Upgrade-Insecure-Requests value Upgrade-Insecure-Requests 值	Bitflip 1/1	4%	8%
	Bitflip 2/1	2%	
	Bitflip 4/1	2%	
	Arithmetic 8/8	3%	3%
	Havoc	89%	89%
User-Agent Value User-Agent 值	Bitflip 1/1	17%	34.8%
	Bitflip 2/1	9%	
	Bitflip 4/1	7%	
	Bitflip 8/8	0.6%	
	Bitflip 16/8	0.6%	
	Bitflip 32/8	0.6%	
	Arithmetic 8/8	52%	52.2%
	Arithmetic 16/8	0.2%	
	Havoc	13%	13%
Accept value Accept 值	Bitflip 1/1	18%	37.8
	Bitflip 2/1	12%	
	Bitflip 4/1	5%	
	Bitflip 8/8	0.5%	
	Bitflip 16/8	0.5%	
	Bitflip 32/8	0.8%	
	Arithmetic 8/8	51%	51.2%
	Arithmetic 16/8	0.2%	
	Havoc	11%	11%

3.1 总体设计

HTTPFuzzer 属于基于变异的灰盒模糊测试工具, 以 AFL 架构为基础开发。HTTPFuzzer 以路径覆盖为导向, 根据测试过程中的路径覆盖信息引导模糊测试的实施。HTTPFuzzer 的总体架构如图 2 所示。

在测试开始前, 采用 AFL-GCC 工具对 Web 服务器程序的源代码进行插桩, 运用编译时插桩技术将源代码编译成包含桩代码的可执行文件作为待测试服务器程序(Server Under Test, SUT)。在测试过程中, 将通过这些桩代码获取程序运行时的内存信息。测试开始时, 由 HTTPFuzzer_fuzz 主进程初始化并分配一块共享内存用于存储模糊过程中程序的路径覆盖信息。测试过程中, 主进程通过读取共享内存信息获取路径覆盖信息以引导模糊测试。SUT 进程对共享内存具有读写权限, 执行测试用例时, 如果遇到新的基本块, 即尚未执行过的代码片段, SUT 进程会更新共享内存中的路径覆盖信息, 标记该基本块的代码已执行。

图 3 展示了 HTTPFuzzer 的基本工作流程。HTTPFuzzer 从数据包捕获文件(Packet Capture, PCAP)中提取应用层载荷作为种子, 并将种子存放在种子池(Seeds pool)中, 每个种子对应着一条完整的 HTTP 报文的应用层载荷。模糊测试开始时, 从种子池中选择一个种子, 进入变异位置探索阶段(具体将在 3.2 节进行详细介绍)。本文将变异位置的探索建模为多臂老虎机问题, 将 HTTP 报文根据空格、换行符等特殊字符切分为多个段。每次选择一个段进行变异, 变异后将各段按序拼接, 封装成数据包并通过网络套接字(Socket)发送至目标程序 SUT, 根据返回报文和程序覆盖率的变化对当前段的变异价值进行评判。经过 n 轮的探索后将变异价值较低的段视为不可变域, 其余的段则视为可变域。可变域将在基于 Q_Learning 的变异阶段使用(具体将在 3.3 节进行阐述)。基于 Q_Learning 的变异阶段的主要工作是使模糊器自适应地根据不同的变异域对历史表现较好的变异算子进行调度。在该阶段, HTTPFuzzer 将依据推断出的可变域位置信息, 构建一个 Q 值表, 量化对不同变异域执行不同变异算子的价值。在测试过程中, Q 值表将依据测试覆盖率与异常触发信息进行迭代优化, 从而动态调整。模糊测试以 Q 值表作为决策依据, 引导变异算子的调度。

模糊过程中, 能够引起覆盖率提升的测试用例将被认为是有趣的测试用例, 将存储进种子池中进行复用。选择提升覆盖率的测试用例作为种子, 产生新的测试用例, 旨在提升触发新路径的可能, 进而提高发现漏洞的概率。模糊测试过程中, 将通过异常监控模块监控 SUT 的运行状态, 记录异常挂起和崩溃等错误信息, 并将引起程序错误的测试用例存储到异常用例库, 方便还原现场以确定漏洞的成因。

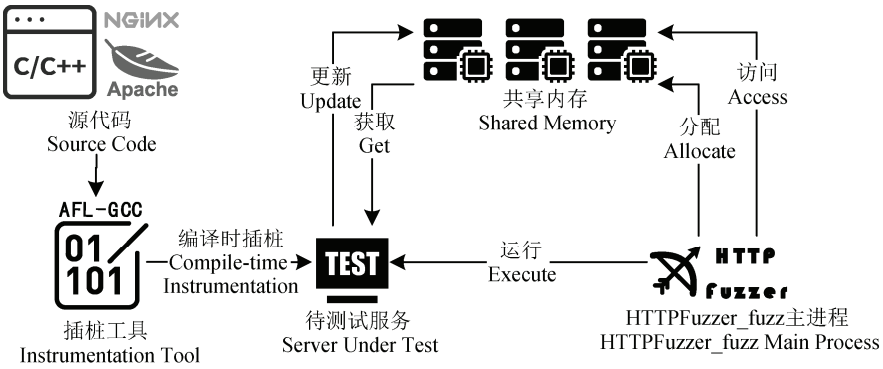


图 2　HTTPFuzzer 的总体架构

Figure 2　Overall architecture of HTTPFuzzer

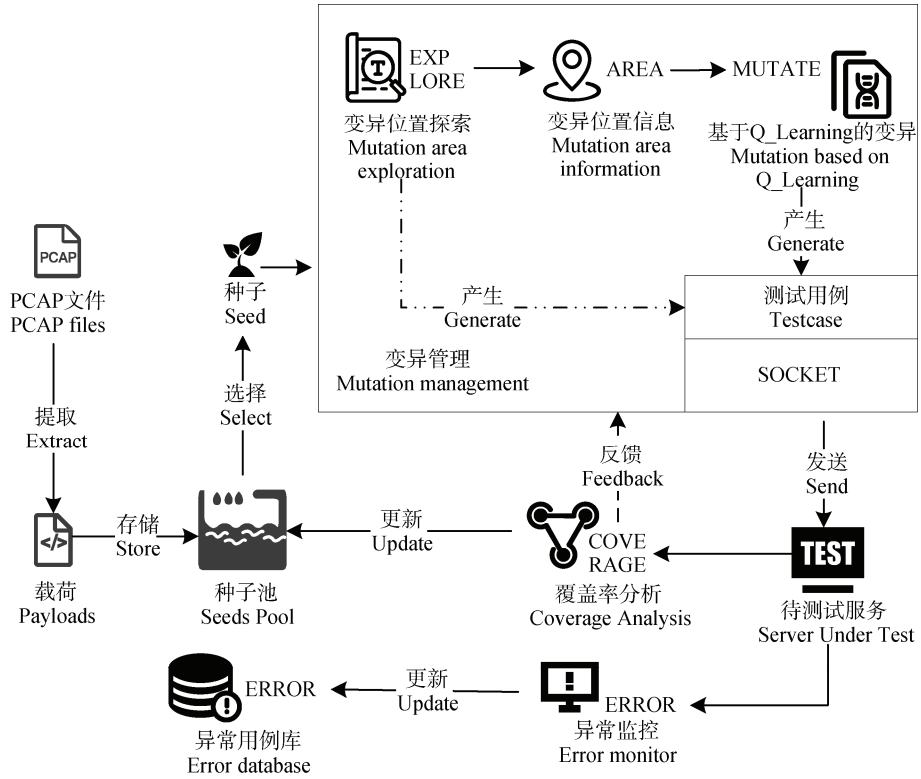


图 3　HTTPFuzzer 的工作流程

Figure 3　Workflow of HTTPFuzzer

HTTPFuzzer 将强化学习融入 Web 服务器程序的模糊测试过程, 通过强化学习引导模糊器自适应地进行变异位置的探索, 并针对不同的变异域进行变异算子的调度。下文将对基于 MAB 的变异位置探索模块和基于 Q_Learning 的变异算子调度模块进行介绍。

3.2　基于 MAB 的变异位置探索模块

在选定一个种子后, HTTPFuzzer 对种子进行两阶段的处理。首先是基于 MAB 的变异位置探索, 在此阶段模糊器主要判定种子中哪些位置具有较高变异价值。其次是基于 Q_Learning 的变异阶段, 此阶段将根据可变域动态地调整变异算子的调度, 旨在根据不同的变异域选择效果较好的变异算子进行变异操作。

3.2.1　变异位置探索模块的设计

变异位置探索阶段主要考虑两方面问题: 首先, 需要对整个报文进行充分探索, 确保报文的每个字段都能够被考虑到。其次, 在较低的资源消耗下快速准确地定位可变域。这就要求变异位置的探索策略需要具备较高的灵活性和快速迭代能力。因此, HTTPFuzzer 的变异位置探索模块将变异位置的探索问题建模为 MAB 问题, 总体流程如图 4 所示。

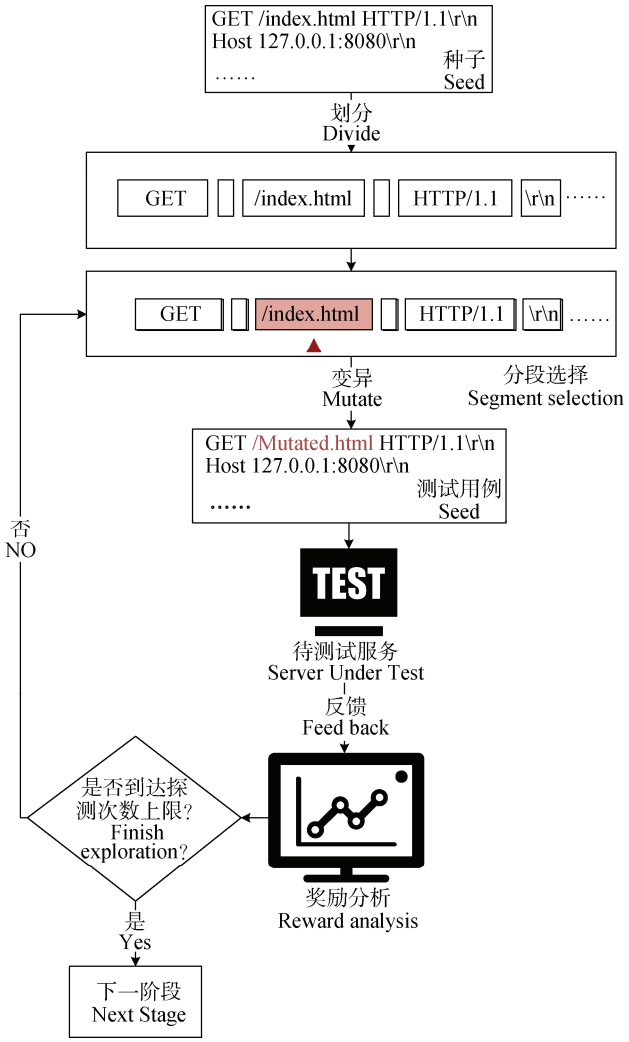


图 4 变异位置探索的工作流程

Figure 4 Workflow of mutation location exploration

首先将种子根据特殊字符进行划分，采用空格、换行符等特殊字符将种子划分为多个段。由于本阶段关注的是修改特定段的内容所产生的测试用例是否合法，因此主要采用 Bitflip 变异算子和特殊符号插入等代价较小的变异方法。

选择一个段采用 Bitflip 变异算子变异，破坏该段的文本内容。随后将各个段拼接成测试用例发送至 SUT，根据返回报文和路径覆盖信息对段的变异价值进行打分。如果返回的报文是“Not Implemented”类型或者“Bad Request”类型，需要对变异所涉及的段进行惩罚。“Not Implemented”类型主要标识了对 HTTP 标准方法字段变异导致服务器无法识别。“Bad Request”类型主要标识报文格式错误导致了异常。

如果测试用例的变异没有破坏报文格式，且服务器返回的报文不是“Not Implemented”或类似的由于操作不被支持而产生的错误，则给予测试用例所变异的段基本奖励。表示该变异是有效的，没有引起意外的错误。如果测试用例不仅能够使系统返回正常报文，而且能够提升 SUT 的代码覆盖率，则对变异涉及的段进行额外奖励。表示该测试用例在发现潜在问题和增强测试效果方面具有较高的价值。

在探索过程中需要设置探索次数的阈值，系统根据在此范围内获得的奖惩信息，将具有较高变异价值的段作为可变域，将具有较低变异价值的段作为不可变域。

3.2.2 变异位置探索模块的实现

在实施 Web 服务器程序的模糊测试时，所选择的种子是 HTTP 协议报文，如图 5 所示。HTTP 协议属于文本协议，报文中存在空格、回车换行符等特殊符号将报文划分为字段。本文采用这些符号作为分割符，将种子分割成多个段。

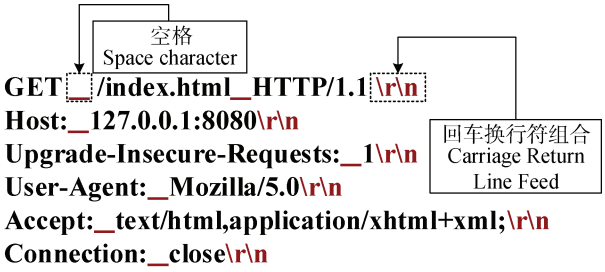


图 5 HTTP 报文示例

Figure 5 Example of HTTP message

变异位置的探索采用了强化学习策略。强化学习的核心是奖励规则，算法依据“动作”获得的奖励，动态调整后续动作。这里对所设计的奖励规则进行介绍。

以特殊符号进行分割后，种子被划分为 n 个段，每个段的变异价值是不同的。以 HTTP 请求报文的请求头为例，本文方法会将其划分为如下列表结构：[“GET”,“ ”,“/index.html”,“ ”,“HTTP/1.1”,“\r\n”]。利用变异算子对“GET”内容进行破坏。比如，使该段内容变为“G%2FET”的形式，在将测试用例发送至 SUT 后，会收到“Not Implemented”类型的反馈报文。再如，对空格以及回车换行组合符“\r\n”进行变异，则会破坏报文格式，产生 Web 服务器程序拒绝处理的无效测试用例。本文认为“GET”“\r\n”等段不具备变异价值。与此同时，通过实验测试，可以观察到对“/index.html”的变异，不但没有返回错误信息，还促使代码覆盖率得到提升。本文认为这种变异后没有产生无效报文，而且能引起代码覆盖率提升的段具有更高的变异价值。

基于上述分析，本文将段分为四类：

1) HTTP 请求中的标准方法字段(如‘GET’、

‘PUT’)。变异并发送至服务器后, 导致服务器因无法识别或处理该变异后的请求而返回‘Not Implemented’(501)状态码或类似表明方法不受支持的反馈报文的段。

2) 空格和“\r\n”等变异后破坏报文格式产生无效用例的段。

3) 变异后引起正常反馈的段。

4) 变异后引起正常反馈, 而且代码覆盖率得到提升的段。

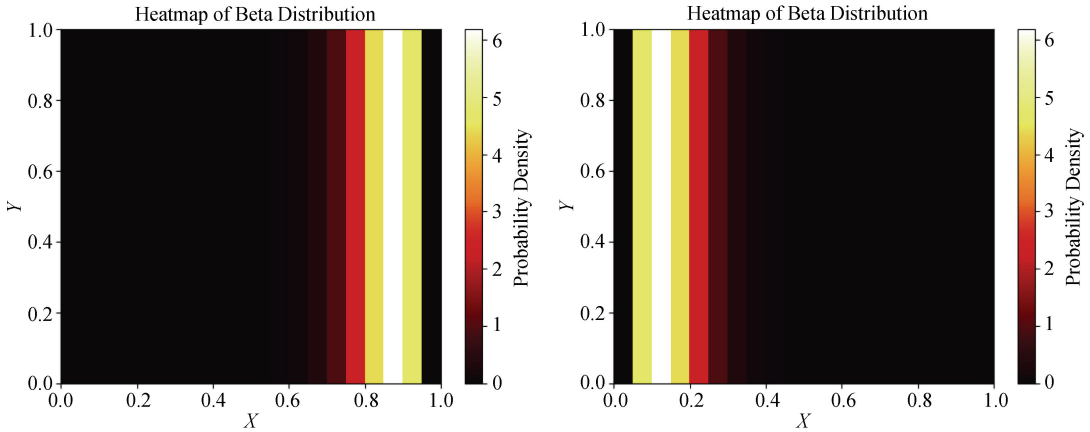
针对这四类段, 本文设计了变异位置探索阶段的奖励规则。设定 mutatable 变量, 当段为类型 1) 和类型 2) 时, 设置 mutatable=0, 否则 mutatable=1。同时设定 interesting 参数, 如果对当前段的变异使代码覆盖率提升, interesting=1, 否则 interesting=0。奖励规则如式(1)所示。

$$\text{reward}=\begin{cases}0\ ,\ \text{mutatable}=0\\ 1\ ,\ \text{mutatable}=1\\ 2\ ,\ \text{mutatable}=1\ \text{and interesting}=1\end{cases}\tag{1}$$

奖励规则是为每个段进行打分的基础, 而打分是后续段选择的依据。本文方法的位置探索阶段关注灵活性和效率。Beta 分布的计算仅由正向权重和负向权重决定, 可以较好地适应模糊测试过程中根据式(1)所积累的观测数据, 具备较强的灵活性。此外 Beta 分布为二项分布的共轭先验分布, 这意味着在给定二项分布似然函数的情况下, 后验分布仍然是 Beta 分布。使用 Beta 分布值作为段变异价值的衡量指标可以简化计算过程, 提高探索效率。因此, 在奖励规则的基础上, 本文对每个段进行打分。并计算每个段的 Beta 分布值, 作为段选择的依据。本文将 MAB 问题与变异位置探测相融合。Beta 分布的计算依赖于式(1), reward=0 时, 对当前段的变异促使 mutatable 参数为 0, 代表当前段不具备变异价值。reward=1 时, 代表对当前段的变异没有产生使 mutatable 参数为 0 的报文, 当前段具备一定变异价值。reward=2 代表对当前段进行变异既不会令 mutatable 参数为 0, 又能使代码覆盖率提高, 当前段具备较高变异价值。段对应 Beta 分布的计算如式(2)所示。

$$\begin{aligned}f(x;\alpha,\beta)&=\frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}=\frac{x^{\alpha-1}(1-x)^{\beta-1}\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\\&=\frac{x^{\alpha-1}(1-x)^{\beta-1}(\alpha+\beta-1)!}{(\alpha-1)!(\beta-1)!}\end{aligned}\tag{2}$$

Beta 分布的概率密度在区间(0, 1)内。式(2)中 $\forall x\in(0,1)$, α 为正向权重, α 为 reward 的累加值, α 越大代表该段越有变异价值。β 为负向权重, β 值越大代表该段越不具有变异价值。当获得 reward 值为 0 时, β 值加 1, $B(\alpha,\beta)$ 为 Beta 函数。图 6 以热力图的形式直观展示了 α 值和 β 值对 Beta 分布的影响。具备较高变异价值段对应 Beta 分布的热力图如图 6(a)所示, 当 α 值较大时, Beta 分布中心接近 1(对应高亮区域接近 1), 代表当前段具有较高变异价值。反之如图 6(b)所示, 分段不具备变异价值时 β 值较大, Beta 分布中心接近 0(对应高亮区域接近 0), 代表当前段变异价值较低。



(a) 高α值情况下的Beta分布
(a) Beta distribution with high α value

(b) 高β值情况下的Beta分布
(b) Beta distribution with high β value

图 6 Beta 分布热力图

Figure 6 Heatmap of Beta distribution

段选择主要依据每个段价值的 Beta 分布。在计算 Beta 分布前, 需要为每个段建立列表, 记录每个段的奖励情况, 列表根据式(1)进行动态更新。如算法 2 中的第 5 行和第 6 行, 统计每个段获取的奖励作为原始正向价值, α_origin。而原始负向价值系数 β_origin 则根据 reward 为 0 的情况得到。reward 为 0

时, 系统判定该次测试中对应段不具备变异价值, β_origin 加 1。为了突出并优先考虑具有高价值的段, 本文引入了“扩张因子”boost, 在算法 2 的第 7 行用于放大正向系数。如算法 2 中第 7 行和第 8 行所示, 为了避免测试初期因正向系数与负向系数皆为零而导致部分段被忽视, 本文设定 α 和 β 参数至少为 1, 确保所有段均有被分析的可能。在计算出 α 和 β 后, 获得 Beta 分布, 并返回 Beta 分布值最高的段在段列表中的位置信息。

算法 2.段选择算法

Algorithm 2. Segment Selection Algorithm
Input: Segment scale list: *Seg_scale,*
Expansion factor: *boost,*
Initial α value: *alpha_base,*
Initial β value: *beta_base;*
Output: Segment index: *index*.
1. **Initialize** *alpha_base* = 1
2. **Initialize** *beta_base* = 1
3. **FOR** *item* **IN** *Seg_scale* DO
4. α_*origin* = Sum(*item*)
5. β_*origin* = Count 0 in *item*
6. α=α_*origin* *boost+*alpha_base*
7. β = β_*origin* +*beta_base*
8. *distribution* = *B*(α, β)
9. *index* = location of max distribution in *items*
10. **RETURN** *index*

段选择是变异位置探索的核心工作。变异位置探索模块的工作如算法 3 所示。首先初始化列表 Seg_scale。Seg_scale 列表中的每个元素都是一个列表(List), 每个列表对应一个段, 用于记录当前段获得奖励的情况。例如[1,1,2,0]代表第一次探索获取奖励为 1, 第二次探索获取奖励为 1, 第三次探索获取奖励为 2, 第四次探索获取奖励为 0, 每次对该段探索所获得的奖励值被加入到列表尾部。接着初始化 Mutation_area 列表用于存储可变域的位置信息, 该列表将作为算法 3 的返回结果。根据预设的探索上限 num_up 进行多次探索, 当探索次数大于 num_up 时结束探索。如算法 3 的第 4 行和第 5 行所示, 为了防止变异位置探索过程完全依赖于算法 3 陷入局部最优, 本文方法设置了 30%的概率进行随机探索, 以确保探索的全面。如算法 3 的第 7 行所示, 以 70%的概率按照算法 2 进行段选择。在选择段后, 对相应段进行变异产生测试用例, 并对 Seg_scale 进行更新, 指导下一轮变异位置的探索。当达到预设的探索上限后, 按照式(2)计算每个段对应的 Beta 分布。与算法 2 不同的是, 这里的 Beta 分布仅用于衡量段的变异价值。在得出每个段的 Beta 分布值后, 对所有段的 Beta 分布求均值, 低于均值的段视为不可变域, 剩余段视为可变域。

算法 3.变异位置探索

Algorithm 3. Mutation Area Exploration
Input: Segment list: *Segments,*
Expansion factor: *boost,*
Output: Mutation area list:*Mutation_area*:
1. **Initialize** Segment scale list: *Seg_scale*
2. **Initialize** list *Mutation_area*
3. **FOR** *i* **IN** range(*num_up*) **DO**
4. **IF** *random.rand* < 0.3 **DO**
5. *index* = randint(0, len(*Segments*)-1)
6. **ELSE**
7. *index*= Seg_selection(*Seg_scale,boost*)
8. *Test_case* = mutate(*Segments,index*)
9. Socket.send(*Test_case*)
10. get_reward(*index,Seg_scale*)
11. Calculate Beta distribution of each segment
12. **IF** Beta distribution > mean value **DO**
13. *Mutation_area*.append(*index*)
14. **RETURN** *Mutation_area*

通过奖励规则的设计, 并利用扩张因子对高价值段进行正面强化, 本文方法筛选出具备变异价值的变异域。表 2 展示了实验中探索完成后部分段的 Beta 分布值。表中“GET”、“ ”和“\r\n”属于 3.2.2 节中所述的前两类段, 变异价值较低, 相应的上述段的 Beta 分布值较低。而对代表 URL 和协议版本的“/index.html”和“HTTP/1.1”进行变异可以有效引起代码覆盖率的提升, 具备较高变异价值, 因此上述段具有较高的 Beta 分布值。

表 2 段和相应的 Beta 分布值

Table 2 Segments and Beta value

编号 NO.	内容 Content	Beta 分布值 Value of Beta
1	b'GET'	0.00235
2	b' '	0.08385
3	b'/index.html'	0.99423
4	b' '	0.19562
5	B'HTTP/1.1'	0.97763
6	b'\r\n'	0.02139

3.3 基于 Q-Learning 的变异算子调度模块

经过基于 MAB 的变异位置探索, 可以筛选出有价值的段作为变异域。变异算子调度模块旨在根据

历史模糊测试中积累的经验, 为不同的变异域匹配效果较好的变异算子, 以提高变异效果, 提升模糊测试效率。为不同的变异域选择最合适的变异算子, 可以抽象为一个基于表格的二维决策问题。其中, 表格的两个维度分别代表有限的变异算子空间和变异域空间。采用 Q-Learning 算法进行决策, 一方面, Q-Learning 能够很好地适应变异算子调度的实际需求, 通过不断学习和更新动作价值, 逐渐逼近最优的变异策略。另一方面, Q-Learning 算法具备高效的迭代速度, 能够迅速响应模糊测试对效率的高要求, 从而在有限的时间内提升测试效率。

3.3.1 变异算子调度模块的总体设计

在正式介绍变异算子调度模块之前, 对如下概念进行介绍。

1) 变异域空间(Mutation-region Space): 在 3.2 节中所述模块处理后, 得到了可变域信息。变异位置限定在这些可变域范围, 本文将这些可变域集合称作**变异域空间**。

2) 变异算子空间(Mutation-Operator Space): 本文方法属于基于变异的模糊测试方法, 变异操作涉及不同的变异算子, 包括 Bitflip、Arithmetic、havoc 等各类变异算子, 本文将这些变异算子的集合称为**变异算子空间**。

3) 奖励(Reward): 奖励综合考虑了可变域空间和变异算子空间。具体而言, 根据模糊器在选择某个**变异域**的情况下, 采用了某个**变异算子**, 是否引起代码覆盖率的提升。如果提升, 则进行正向的奖励, 否则不奖励。

4) Q 值(Q_value): 模糊测试是一个长期迭代的过程, 变异域的选择与变异算子的调度构成了一个动态决策问题。为了构建智能且高效的模糊变异算子调度策略, 不仅需要关注即时奖励, 还需要考虑决策可能带来的长期收益。因此, Q 值的计算不仅局限于当前的变异位置和变异算子, 还考虑到未来可能选择的变异位置和变异算子。策略更新基于 Q 值进行, 依据 Q 值的高低来决定变异算子的调度以及变异域的选择。

变异算子调度模块的工作流程如图 7 所示。在 3.2 节所述模块的处理下, 可以获取到变异域空间的相关信息。本文方法在 AFL 的基础上进行拓展, 变异算子空间与 AFL 系列工具保持一致, 包括 Bitflip 系列算子、Byteflip 系列算子及 havoc 算子等多种变异算子。变异算子调度模块通过结合变异域空间和变异算子空间, 创建Q值表, 用于存储表示变异操作价值的 Q 值。在 Q 值表中, 行代表可变域, 列代表变异算子, 表中的条目代表对某个域调用某个变异算子的价值。

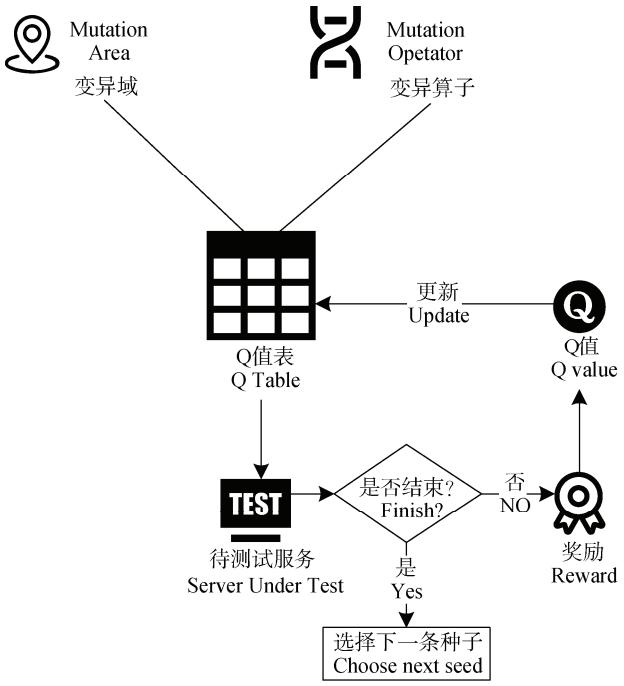


图 7 变异算子调度模块的工作流程

Figure 7 The workflow of the Mutation Operator Scheduling Module

Q 值表是变异算子调度的依据, 表中元素代表选择某一变异域时, 调用某一变异算子的价值。根据 Q 值表, 基于变异域选择历史表现最优的变异算子进行变异, 生成测试用例提交给 SUT。模糊器根据测试覆盖率的变化, 获得相应的奖励, 奖励值参与Q值的更新计算。下一轮的变异域选择与变异算子调度将基于最新的 Q 值表进行。上述过程循环反复, 直至达到预设的测试轮数。此后, 模块将从种子池中选取下一个种子开展测试。

3.3.2 变异算子调度模块的实现

变异算子调度模块首先根据变异域空间和变异算子空间建立一个元素全为 0 的 Q 值表, 如算法 4 所示。Q 值表的行代表各个变异域, Q 值表的列代表各个变异算子, 表中的元素代表对相应变异域调用对应变异算子的价值。

算法 4.Q 值表初始化

Algorithm 4. Q table initialization

Input: Mutation area space: *m_area,*

Mutation operator space: *m_operator;*

Output: Mutation area list:*Mutation_area*:

1. *Row_len* = len(*m_area*)

2. *Column_len* = len(*m_operator*)

3. **CREATE** a 2D array of size [*Row_len, Column_len*] filled with 0s **as** *Q_Table*
4. **RETURN** *Q_Table*

本阶段的“操作”是对某个变异域调用某个变异算子。本文用 Q 值来衡量“操作”的价值。Q 值的计算既基于当前调度操作所获得的即时奖励, 也考虑到未来的调度操作, 可以令基于 Q 值表的决策充分全面。这里先介绍本阶段的奖励规则。

$$\text{Q\_reward} = \begin{cases} 1, & \text{interesting\_opt} = 1 \\ 0, & \text{interesting\_opt} = 0 \end{cases} \tag{3}$$

本文方法以代码覆盖率为导向, 如果当前“操作”能够引起代码覆盖率的提升, 则奖励值为 1, 否则奖励值为 0。实际应用中可对奖励值的设置进行调整。为规范表述, 将能够引起代码覆盖率提升的“操作”称为“有趣操作”, 并设置指示标识 interesting_opt。如果该指示标识为 1, 表示进行了一次“有趣操作”, 获得的奖励为 1, 否则获得的奖励为 0, 如式(3)所示。

$$Q(A,O) \leftarrow Q(A,O) + \theta[\text{Q\_reward} + \gamma \max_{O'} Q(A',O') - Q(A,O)] \tag{4}$$

式(4)为 Q 值表的更新规则, 其中 A 代表变异域, O 代表变异算子, (A,O)组合表示在选定变异域 A 的情况下调用变异算子 O。式(4)中的 θ 表示学习率, 用于控制奖励对 Q 值的影响, 如果学习率较高(接近 1), 那么奖励对 Q 值的影响较大; 如果学习率较低(接近 0), 则奖励对 Q 值的影响较小。Q_reward 为即时奖励, 代表当前操作立即能获得的奖励。$\max_{O'} Q(A',O')$ 代表对未来操作的预测, 基于当前 Q 值表进行计算。为保证测试的全面性, 本文默认 A' 为当前变异域的相邻变异域, O' 为 A' 对应 Q 值最高的变异算子, $\max_{O'} Q(A',O')$ 则为 Q 值表中 (A',O') 对应 Q 值。值得一提的是, 由于此时操作并没有真实发生, 只是根据 Q 值表中的数值进行推断, 所以需要减小这一预测操作对 Q 值表更新的影响, 引入了 γ 作为衰减因子对 $\max_{O'} Q(A',O')$ 进行调节。

模糊测试是一个长期迭代的过程, 决策应具备全面性, 不能局限于当前的即时奖励, 还应考虑到长期收益(迭代过程中获得的奖励总和)。本文引入未来预期奖励, 基于 Q 值表对未来操作进行推断, 使得模糊器做出更合理的决策。上述过程如算法 5 所示。

算法 5.Q 值表更新
Algorithm 5. Q table update
Input: Mutation area space: *A,*
Mutation operator space: *O,*
Current mutation area*: A_current*
Current mutation operator: *O_current*
Initial Q table*: Q_table*
Current reward: *reward*
Output: Updated Q table: *Q_table*
1. **Initialize** θ,γ
2. *Next_A* = (*A_current*+1)%len(*A*)
3. *O_current* = *Q_table*.loc[*A_current*].idxmax()
4. *predict_next* =max(*Q_table*.loc[*next_A*])
5. *Q_value* = *Q_table*.loc[*A_current*,*O_current*]
6. *Q_value_new* = (1-θ)**Q_value* + θ**(reward* +γ **predict_next)*
7. *Q_table*.loc[*A_current,O_current*]=*Q_value_new*
8. **RETURN** *Q_Table*

基于 Q-Learning 的模糊测试过程如算法 6 所示。首先, 采用算法 4 初始化 Q 值表。随后基于当前的 Q 值表, 选取对应 Q 值最大的变异域和变异算子(初始阶段采用随机选择)。为避免算法过早陷入局部最优解, 所设计的策略有一定的随机性。有 30%的概率会随机挑选变异域和变异算子(如算法 6 第 4 行所示), 以确保充分探索变异域和变异算子。此外, 则依据 Q 值表的指导, 选择对当前变异域效果最佳(具备最大 Q 值)的变异算子。在根据 Q 值表选定了变异算子和变异域后, 对种子进行变异, 产生测试用例发送至 SUT。根据测试覆盖率, 依照式(3)规则获得奖励。接着遵循算法 5 对 Q 值表进行更新。上述过程将持续, 直至预设的测试次数上限。此时, 再测试下一个种子。

算法 6.基于 Q-Learning 的模糊测试
Algorithm 6. Fuzzing based on Q-Learning
1. **Initialize** *Q_table*
2. **Initialize** *Round*
3. **FOR** *i* **IN** range(*Round*) **DO**
4. **IF** *random.rand* < 0.3 **DO**
5. Choose randomly
6. **ELSE**
7. Choose based on *Q table*
8. Mutate and send to SUT
9. Get *reward* and update *Q_table*
10. Choose next seed

在前述优化策略的驱动下, 模糊测试器依据测试过程中积累的经验, 采用 Q 值表作为决策依据, 以提高代码覆盖率为导向, 对变异算子进行调度。所提出的方法增强了变异操作的有效性, 提升了模糊测

试的整体效率。

4 实验验证

为了验证 HTTPFuzzer 的有效性, 在一系列软件实体程序上开展了验证实验。HTTPFuzzer 原型工具部署在 Ubuntu 20.04 LTS 系统上, 运行内存为 16GB。原型工具基于 C 语言和 Python 语言混合开发, 具备较强的稳定性和扩展性。

在实验内容的设计上, 首先分析测试用例的质量, 对变异位置探索阶段和模糊测试整体过程中的测试用例进行分析, 验证 HTTPFuzzer 方法的优势。接着分析模糊测试的效果, 从漏洞触发能力、路径覆盖能力两方面对本文方法进行评估, 并和 AFLnwe、AFLNet 等主流工具进行横向对比。

4.1 测试用例质量

实验中为了生成测试用例, 采用 3.2 节所述方法对变异位置进行探索, 在处理完成后, 针对变异域实施变异。

测试开始前需要进行 800 轮的变异位置探索, 变异位置探索基于 MAB, 对种子的每个段进行分析。

实际使用中也可根据需求适当调整。如图 8 所示, 该阶段共产生了 82402 条用例, 其中 2109 条用例由于选择了“GET”、“PUT”一类的段进行变异, 产生了无法被 SUT 解析的用例。由于方法的奖励规则和策略更新机制, 在位置探索的过程中, 对于此类段探索的次数也会随着测试的深入而减少, 避免了无效用例的产生, 节省了测试资源。

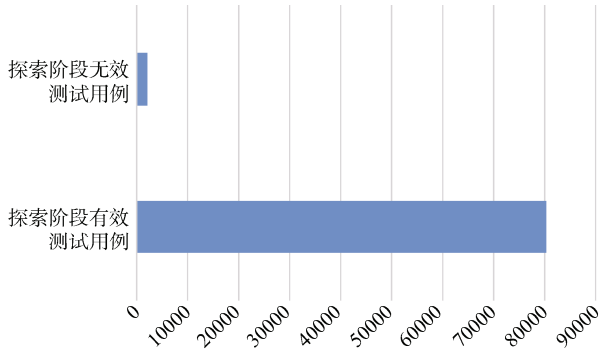


图 8 变异位置探索阶段测试用例分布

Figure 8 Distribution of test cases in the mutation location exploration phase under default configuration.

在位置探索结束后, 模糊器筛选出了变异域, 剔除了“GET”、“PUT”等不可变字段。如表 3 所示, 611966 条测试用例在探索阶段之后产生, 其中有效用例的占比可达 98.86%, 由于一些特殊符号(如换行)的插入, 在一些情况下也会产生非法用例(例如, 在 URL 前插入换行符), 无法完全避免无效用例的产生。但从总体上看, 无效用例被控制在了较小的范围。

表 3 不同阶段测试用例分布情况

Table 3 Distribution of test cases in different stages

	探索阶段 Exploration stage	后续阶段 Subsequent stage
有效用例数 number of valid test cases	80239	605002
无效用例数 Number of invalid test cases	2109	7018

4.2 模糊测试效果

实验将 HTTPFuzzer 运用于真实 Web 服务器程序的漏洞挖掘, 并和 AFLnwe、AFLNet 进行对比。本文选取了 Lighttpd[21]、Thttpd[22]和 Nginx[23]三款主流 Web 服务器程序作为实验目标, 包括了 Lighttpd 程序的 1.4.66、1.4.67、1.4.51、1.4.76、1.4.75 等版本, Thttpd 程序的 2.27、2.28 等版本以及 Nginx 1.4.2、1.4.4、1.24.0 等版本。

1) 测试对象

实验测试主要选择 Lighttpd、Thttpd 和 Nginx 三款具有代表性的 Web 服务器程序。

Lighttpd 和 Thttpd 被大量应用于嵌入式设备中。相比其他主流的 Web 服务器程序, 这两款 Web 服务器程序资源消耗更低, 设计也更加轻量化, 随之带来的安全性问题也不容忽视。漏洞 CVE-2019-11072[24]以及中危漏洞 CVE-2024-5294[25]和高危漏洞 CVE-2024-5293[26]均与 Lighttpd 直接相关, 高危漏洞 CVE-2017-17663[27]、CVE-2019-8387[28]均和 Thttpd 相关。Nginx 则是主流 Web 服务器程序的代表, 由于其部署便捷、支持功能广泛等特点受到企业和个人开发者的认可, 在实际场景中被广泛部署, 因此本文也将 Nginx 选为需要关注的测试目标之一。

2) 漏洞触发能力

HTTPFuzzer 具备良好的漏洞触发能力, 可以在较短的时间内触发漏洞。在 3.2 节所述变异位置探索模块的基础上, HTTPFuzzer 将具备高价值的段作为变异域, 这些变异域包括 URL 请求路径、HTTP 版本、HOST 信息以及请求参数等。HTTPFuzzer 在测试中聚焦于这些变异域, 并且随着测试的深入, 针对不同变异域调度不同的变异算子, 提高了变异的有效性。

如图 9 所示, HTTPFuzzer 成功引起了 Lighttpd 1.4.51、Lighttpd 1.4.52、Lighttpd 1.4.53 以及 Lighttpd 1.4.65 的崩溃。

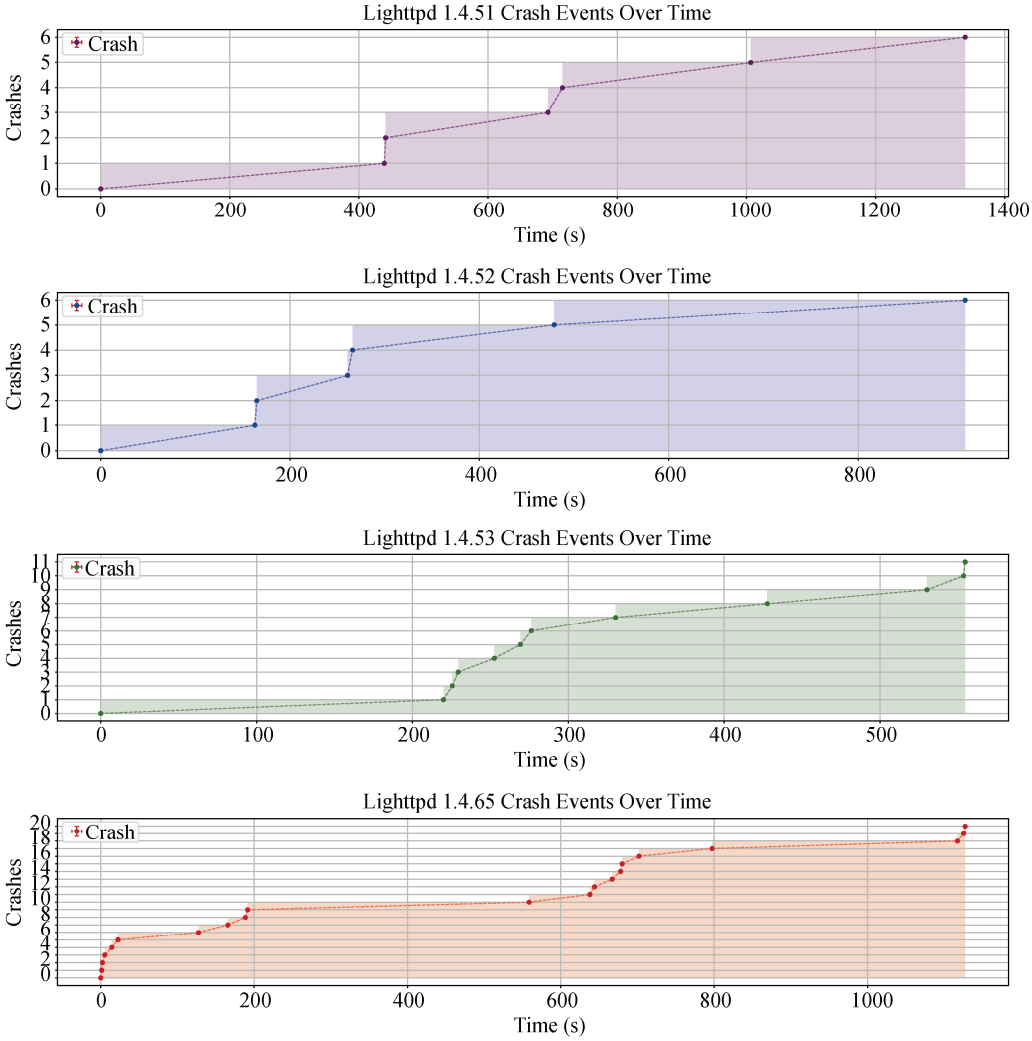


图 9 Lighttpd 程序崩溃情况

Figure 9 Crashes of Lighttpd

在对 Lighttpd 1.4.51 的测试中, 共引起 6 次崩溃, 第一次崩溃在测试开始后的第 439.18 秒。在对 Lighttpd 1.4.52 的测试中, 共观察到 6 次崩溃现象, 第一次崩溃在测试开始后的第 162.79 秒。在对 Lighttpd 1.4.53 的测试中, 共引起 11 次崩溃, 第一次崩溃在测试开始后的第 219.93 秒。经过对 POC 进行重放, 和排查以上三个版本均受到 CVE-2019-11072 影响。而 Lighttpd 1.4.65 受到 CVE-2022-37797 影响, 测试过程中共引发 20 次崩溃, 在测试开始后的第 2.29 秒就成功引起了程序的第一次崩溃。

进一步分析崩溃时间的差异, 经过漏洞排查和对造成崩溃的 POC 样本分析, 可以发现针对版本 1.4.51～1.4.53 的 POC 均由 GET 请求样本对 URL 路径的变异而得到。Lighttpd 1.4.65 的第一次崩溃在变异位置探索阶段, 耗时 2.29 秒触发, 崩溃现场如图 10 所示。经过对 POC 重放复现崩溃现场, 我们发现当 HTTP 报文的请求头中“Sec-WebSocket-Version: 13”代表值的“13”被变异为非数值型时, 服务程序在处理此类报文时会将状态码设定为 400, 但相关函数的指针并没有得到正确初始化, 当程序调用该指针时, 由于指针为空, 导致程序崩溃。本文方法在变异位置探索阶段, 采用的变异算子仅破坏种子中的文本, 加上更新策略的设定, 变异操作快速覆盖到了“Sec-WebSocket-Version: 13”中“13”的位置, 造成了程序的崩溃。此外, 如图 11, 本文方法在对 Thttpd 2.26(CVE-2017-17663) 的测试过程中, 于 277.3 秒成功触发了崩溃。根据模糊测试日志, 崩溃在基于 Q-Learning 的变异算子调度阶段触发。

总体上看, HTTPFuzzer 能够有效探索变异域, 在测试过程中根据不同的变异域调度变异算子, 从而在较短时间内发现漏洞。

为了测试崩溃的触发速度, 本文采用 AFLnwe 和 AFLNet 以及 HTTPFuzzer 对 CVE-2022-37797、CVE-2019-11072 和 CVE-2017-17663 进行验证测试,

```
==13725==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000
(pc 0x000000000000 bp 0x615000000300 sp 0x7ffeddb51d88 T0)
==13725==Hint: pc points to the zero page.
==13725==The signal is caused by a READ memory access.
==13725==Hint: address points to the zero page.

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV (<unknown module>)
==13725==ABORTING
```

图 10　Lighttpd 1.4.65 程序崩溃现场

Figure 10　The crash site of Lighttpd version 1.4.65

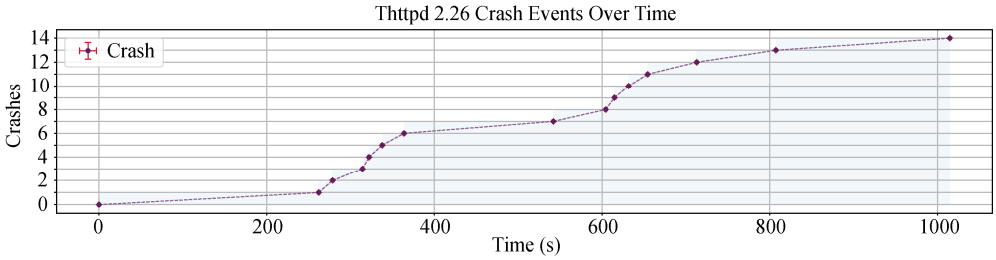


图 11　Thttpd 2.26 程序崩溃情况

Figure 11　Crashes of Thttpd 2.26

包含了 Lighttpd 1.4.51、Lighttpd 1.4.52、Lighttpd 1.4.53 等多个测试目标。本文对每个测试目标进行了 10 轮测试, 取第一次崩溃发生的时间区间进行对比。

结果如表 4 所示, 对于 Lighttpd 1.4.65, HTTPFuzzer 触发第一次崩溃时间普遍集中在 2~9 秒, 而 AFLnwe 和 AFLNet 普遍超过 200 秒。

表 4　第一次造成崩溃的时间

Table 4　First time to crash

CVE 编号	测试目标	HTTPFuzzer	AFLnwe	AFLNet
CVE-2022-37797	Lighttpd 1.4.65	2～9 s	> 200 s	> 200 s
CVE-2019-11072	Lighttpd 1.4.51 1.4.52 1.4.53	150～450 s	> 500 s	> 500 s
CVE-2017-17663	Thttpd 2.26	200～400 s	> 500 s	> 500 s

对于 Lighttpd 版本 1.4.51～1.4.53, HTTPHuntter 触发第一次崩溃的时间集中在 150～450 秒。对 Thttpd 2.26 的测试中, HTTPFuzzer 触发崩溃的时间集中在 200～400 秒, AFLnwe 和 AFLNet 触发崩溃的时间普遍超过 500 秒。

AFLnwe 和 AFLNet 的变异位置选择和变异算子调度均基于原生 AFL, 因此存在盲目性, 难以快速定位到相应变异位置并调度较优的变异算子。对于 Web 服务器的模糊测试以 HTTP 报文为种子展开, HTTP 报文较长, 采用传统方法难以使变异操作在较短时间内覆盖到种子的敏感位置。而 HTTPFuzzer 在强化学习算法的引导下, 筛选出了具有变异价值的段作为变异域, 接着根据变异域选择效果较好的变异算子进行调度, 导向性较强。因此在漏洞触发效率上优于传统方法。

3)　路径覆盖能力

本文针对 Lighttpd 和 Thttpd 两款较为常用的轻量化 Web 服务器程序的代表性版本实体展开测试。每个实体分别进行 5 轮测试, 测试时间均为 60 分钟。5 轮测试的平均测试路径覆盖信息如表 5 所示。从表中结果可以看到, 在覆盖路径总数上, 本文方法整体上高于对比方法。

表 5　路径覆盖信息

Table 5　Path coverage information

	HTTPFuzzer	AFLnwe	AFLNet
Lighttpd 1.4.65	**684.0**	680.4	654.0
Lighttpd 1.4.66	**651.2**	651.2	644.2
Lighttpd 1.4.67	**677.2**	655.6	658.2
Lighttpd 1.4.75	**713.0**	681.2	667.0
Lighttpd 1.4.76	**722.0**	689.4	661.0
Thttpd 2.29	**395.6**	399.2	395.2
Thttpd 2.28	**411.2**	401.8	410.2
Thttpd 2.27	**400.2**	397.2	399.8

本文提出的方法同样适用于规模相对较大的 Web 服务器程序。为验证方法的有效性, 本文选取了应用广泛且具有代表性的 Nginx 服务器作为测试对象, 并与 AFLnwe 进行横向对比。实验选取了 Nginx

的五个版本: 1.24.0、1.14.2、1.4.2、1.4.4 和 1.4.6。针对每个版本, 分别进行了 5 组实验。鉴于 Nginx 项目规模较大, 为确保测试的充分性, 每组实验的持续时间均设定为 180 分钟。

表 6 记录了实验结果的平均值, 包括实验覆盖的路径数以及本文方法相对于 AFLnwe 的覆盖率增长情况。通过观察实验数据, 可以发现, 在对 Nginx 进行实验时, 经过长时间的充分测试, 得益于有针对性的变异策略以及适应性的变异算子调度机制, 本文方法展现出了显著优势。在三个版本的实验中, HTTPFuzzer 的路径覆盖率相对 AFLnwe 均提高了 40%以上。

表 6 对 Nginx 的测试结果

Table 6 Test Results for Nginx

	HTTPFuzzer	AFLnwe	Increase
Nginx 1.24.0	**819.2**	529	54.9%
Nginx 1.14.2	**832.2**	531	56.7%
Nginx 1.4.2	**844.2**	564.4	49.5%
Nginx 1.4.4	**853**	587.2	45.7%
Nginx 1.4.6	**810**	577	40.3%

对比图 12 与图 13 所示的路径覆盖曲线, 不难发现本文所提出的方法在路径触发与覆盖方面展现出了显著的优势。具体而言, 通过这两幅图的直观展示, 本文方法的路径覆盖曲线以一种更为陡峭且持续的态势上升, 直接体现了其在路径探索速度上的高效性。本文方法仅需大约 500 秒的时间, 便能够成功覆盖并探索到超过 400 条不同的程序路径。对比 AFLnwe, 在同一时间段内仅能够覆盖到约 200 条路径, 明显落后于本文方法。

分析其中原因, 以 AFLnwe 为代表的 AFL 系列工具变异算子的调度和变异位置的选择往往采用随机调度或顺序调度的策略, 这导致此类工具在执行过程中容易构建出大量无效用例, 进而导致在无效路径或重复路径上耗费大量时间, 无法高效地覆盖程序路径。而本文方法则在变异位置探索阶段, 筛选出有价值的段作为变异域, 对这些段进行变异可以有效地提高代码覆盖率, 避免了后续变异耗费在其他不具变异价值的段上。通过基于 Q-Learning 的变异算子调度策略, 能够根据不同的变异域, 优先执行那些可以提升覆盖率的变异操作, 从而在短时间内覆盖了更多程序路径。

5 总结和展望

为了使模糊器在对 Web 服务器程序模糊测试时, 能够根据不同的变异位置优先调度有效的变异算子以提高测试用例质量进而提升模糊测试效率, 本文提出了 HTTPFuzzer, 一种强化学习引导的 Web 服务

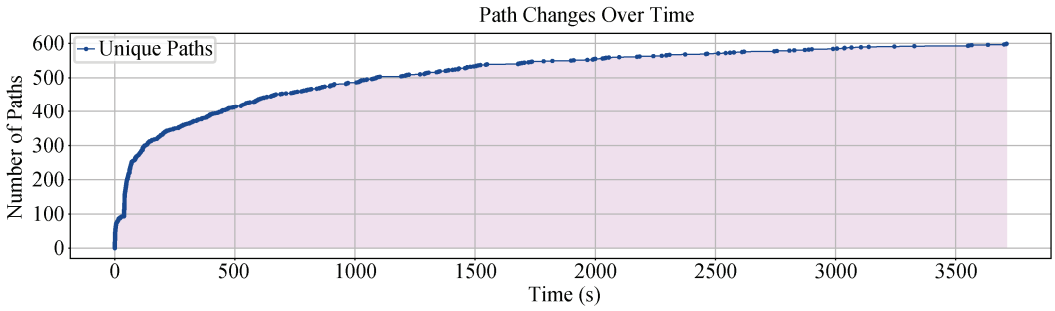


图 12 HTTPFuzzer 的路径覆盖曲线

Figure 12 Path Coverage curve of HTTPFuzzer

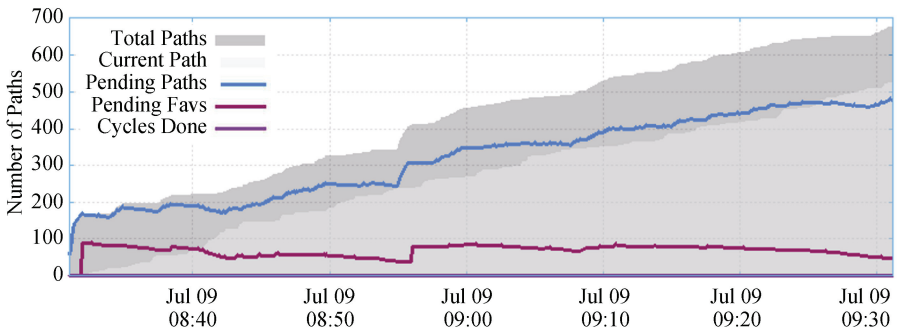


图 13 AFL 系列工具的路径覆盖曲线

Figure 13 Path Coverage curve of AFLs

模糊测试方法。HTTPFuzzer 在 AFL 的基础上扩展, 将变异分为两个阶段。首先基于 MAB 对变异位置进行探索, 此阶段结合 SUT 的返回报文和代码覆盖率变化进行联合引导, 会排除变异后产生无效用例的变异位置, 并给予代码覆盖率提高的变异位置较高的优先级, 快速有效地筛选出变异域。其次基于 Q-Learning 进行变异算子的调度策略优化, 使模糊器在不断深入的测试中根据不同的变异域优先调度效果较好的变异算子, 以避免低效变异带来的资源消耗。实验表明, HTTPFuzzer 能够产生大量符合 HTTP 报文规范的高质量有效用例, 同时具备快速触发程序崩溃的能力, 能够在较短时间内覆盖更多程序路径。

HTTPFuzzer 方法也存在着一些问题。首先, 测试覆盖率的提升仅依靠对现有种子的变异, 基于现有种子变异所产生测试用例的能力有限, 当达到种子的变异上限, 难以产生更多新形态的测试用例时模糊测试将陷入停滞, 难以深入。此外, 对 Web 服务器程序的模糊测试, 极大程度上依赖于配置文件的编写, 有些漏洞在特定配置下才会产生, 而配置文件的编写依赖于人工, 自动化程度还需提升。

基于上述总结, 未来将结合符号执行一类的动态程序分析方法, 当模糊测试发生停滞时利用符号执行计算程序约束值作为测试用例, 辅助模糊测试的深入; 其次, 可以利用自然语言处理方法, 对官方配置文档进行关键信息抽取, 以自动化地构造配置文件, 提升 Web 服务器程序模糊测试的自动化程度。

参考文献

[1] Raunak M S, Kuhn R, Kogut R, et al. Vulnerability Trends in Web Servers and Browsers[C]. *The 7th Symposium on Hot Topics in the Science of Security*, 2020: 1-2.

[2] American Fuzzy Lop. Google. https://github.com/google/AFL. Aug. 2024.

[3] Natella R, Pham V T. ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing[C]. *The 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021: 662-665.

[4] 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.

[5] Natella R. StateAFL: Greybox Fuzzing for Stateful Network Servers[J]. *Empirical Software Engineering*, 2022, 27(7): 191.

[6] Mallissery S, Wu Y S. Demystify the Fuzzing Methods: A Comprehensive Survey[J]. *ACM Computing Surveys*, 2023, 56(3): 1-38.

[7] Lyu C, Ji S, Zhang C, et al. {MOPT}: Optimized mutation scheduling for fuzzers[C]. *28th USENIX Security Symposium,* 2019: 1949-1966.

[8] Ding Z H, Huang Y H, Yuan H, et al. Introduction to Reinforcement Learning[M]. Deep Reinforcement Learning: Fundamentals, Research and Applications. Singapore: Springer Singapore, 2020: : 47-123.

[9] Kuleshov V, Precup D. Algorithms for Multi-Armed Bandit Problems[EB/OL]. 2014: arXiv: 1402.6028. https://arxiv.org/abs/1402.6028.

[10] Clifton J, Laber E. Q-Learning: Theory and Applications[J]. *Annual Review of Statistics and Its Application*, 2020, 7: 279-301.

[11] Honggfuzz. Google. https://github.com/google/honggfuzz. Aug. 2024.

[12] Serebryany K. Continuous Fuzzing with libFuzzer and AddressSanitizer[C]. *2016 IEEE Cybersecurity Development*, 2017: 157.

[13] AFLNW: network wrapper for AFL. LyleMi. https://github.com/lylemi/aflnw. Aug. 2024.

[14] Hong X Q, Jia P, Liu J Y. AFLNe Trans: Fuzzing of Protocols with State Relationship Awareness[J]. *Netinfo Security*, 2024, 24(1): 121-132.
(洪玄泉, 贾鹏, 刘嘉勇. AFLNeTrans: 状态间关系感知的网络协议模糊测试[J]. *信息网络安全*, 2024, 24(1): 121-132.)

[15] Zhao X Q, Qu H P, Xu J L, et al. AMSFuzz: An Adaptive Mutation Schedule for Fuzzing[J]. *Expert Systems with Applications*, 2022, 208: 118162.

[16] Zou Y Y, Zou W, Yin J W, et al. Research on Mutator Strategy-Aware Parallel Fuzzing[J]. *Journal of Cyber Security*, 2020, 5(5): 1-16.
(邹燕燕, 邹维, 尹嘉伟, 等. 变异策略感知的并行模糊测试研究[J]. *信息安全学报*, 2020, 5(5): 1-16.)

[17] Drozd W, Wagner M D. FuzzerGym: A Competitive Framework for Fuzzing and Learning[EB/OL]. 2018: arXiv: 1807.07490. https://arxiv.org/abs/1807.07490.

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

[19] Aschermann C, Schumilo S, Blazytko T, et al. REDQUEEN: Fuzzing with Input-to-State Correspondence[C]. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019: 1-15.

[20] Gan S T, Zhang C, Chen P, et al. GREYONE: Data Flow Sensitive Fuzzing[C]. *USENIX Security Symposium*, 2020

[21] Karayiannis C. The Lighttpd Web Server[M]. Web-Based Projects that Rock the Class. Berkeley, CA: Apress, 2019: 165-217.

[22] Patil V U, Kapur A R. Water Management System Using Dynamic IP Based Embedded Webserver in Real Time[C]. *2015 International Conference on Nascent Technologies in the Engineering Field*, 2015: 1-5.

[23] Reese W. Nginx: The High-Performance Web Server and Reverse Proxy[J]. *Linux Journal*, 2008, 2008(173): 2.

[24] CVE-2019-11072. CVE. https://cve.mitre.org/cgi-bin/cvename.cgi name=CVE-2019-11072. Aug. 2024.

[25] CVE-2024-5294. https://www.cve.org/CVERecord id=CVE-2024-5294. Dec. 2024.

[26] CVE-2024-5293. https://www.cve.org/CVERecord id=CVE-2024-5293. Dec. 2024.

[27] CVE-2017-17663. CVE. https://cve.mitre.org/cgi-bin/cvename.cgi name=CVE-2017-17663. Aug. 2024.

[28] CVE-2019-8387. CVE. https://cve.mitre.org/cgi-bin/cvename.cgi name=CVE-2019-8387. Aug. 2024.

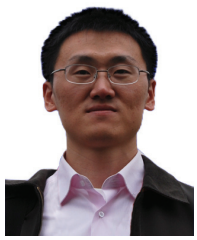
陈乾 于 2021 年在南京理工大学紫金学院软件工程专业获得学士学位。现在陆军工程大学网络空间安全专业攻读硕士学位。研究领域为协议安全、软件安全。研究兴趣包括: 模糊测试、协议逆向。Email: CHENQianLGD@163.com

江川 于 2024 年在陆军工程大学信息安全专业获得学士学位。现在陆军工程大学网络空间安全专业攻读硕士学位。研究领域为软件安全、协议安全。研究兴趣包括: 模糊测试、污点分析。Email: 331657393@qq.com

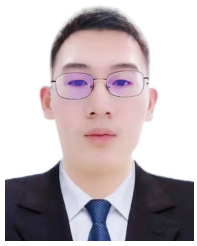
秦素娟 于 2009 年在哈尔滨工业大学计算机科学与技术专业获得硕士学位。现任陆军工程大学指挥控制工程学院助教。研究领域为网络安全、软件安全。研究兴趣包括: 软件定义网络、模糊测试。Email: 86538835@qq.com

崔帅 于 2021 年在泰山学院计算机科学与技术专业获得学士学位。现在陆军工程大学网络空间安全专业攻读硕士学位。研究领域为系统安全。研究兴趣包括高级持续性威胁检测。Email: 18454838830@163.com

洪征 于 2007 年在解放军理工大学计算机应用专业获得博士学位。现任陆军工程大学指挥控制工程学院副教授。研究领域为协议安全、软件安全。研究兴趣包括: 协议逆向、模糊测试。Email: hz5215@163.com



张国敏 于 2009 年在解放军理工大学网络工程专业获得博士学位。现任陆军工程大学指挥控制工程学院副教授。研究领域为网络安全。研究兴趣包括: 软件定义网络、分布式系统。Email: 40519667@qq.com



古津榜 于 2023 年在陆军工程大学网络工程专业获得学士学位。现在陆军工程大学网络空间安全专业攻读硕士学位。研究领域为软件安全、AI 安全。研究兴趣包括: 模糊测试、AI 后门检测。Email: jinbang66666@126.com