计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(9):32−42 [doi: 10.15888/j.cnki.csa.009219]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于双重覆盖信息协同的协议模糊测试①

张颖君[1,2], 周　赓[1,2], 程　亮[1,2], 孙晓山[1,2], 张　阳[1,2]

[1](中国科学院大学, 北京 100049)
[2](中国科学院 软件研究所 可信计算与信息保障实验室, 北京 100190)
通信作者: 张　阳, E-mail: zhangyang@iscas.ac.cn

摘　要: 模糊测试在挖掘协议软件安全漏洞、提高安全性方面发挥着巨大的作用. 近年来将状态引入服务端程序模糊测试受到广泛关注. 本文针对现有方法未充分利用协议模糊测试过程信息、无法持续关注重点状态, 导致模糊测试效率较低的问题, 提出了基于双重覆盖信息协同的协议模糊测试方法. 首先, 本文提出的状态选择算法, 通过建立状态空间到程序空间的映射, 利用启发式的计算方法为每个状态设置权重, 以引导模糊测试持续关注更可能存在缺陷的状态. 其次, 快速探测种子不影响状态但改变程序覆盖的位置, 并限制变异位置以充分测试重点状态对应的代码区域. 本文在基线工具 AFLNet 和 SnapFuzz 上验证了改进算法的有效性, 并最终集成实现了协议模糊测试工具 C2SFuzz. 对 LightFTP、Live555 等协议服务端程序最新版进行了实验后, 发现 5 个未知的漏洞.

关键词: 模糊测试; 漏洞挖掘; 协议; 状态选择; 变异优化

引用格式: 张颖君,周赓,程亮,孙晓山,张阳.基于双重覆盖信息协同的协议模糊测试.计算机系统应用,2023,32(9):32–42. http://www.c-s-a.org.cn/1003-3254/9219.html

Protocol Fuzz Testing Based on Double Coverage Information Coordination

ZHANG Ying-Jun[1,2], ZHOU Geng[1,2], CHENG Liang[1,2], SUN Xiao-Shan[1,2], ZHANG Yang[1,2]

[1](University of Chinese Academy of Sciences, Beijing 100049, China)
[2](Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Fuzzing plays a significant role in discovering security vulnerabilities and improving security in protocol software. In recent years, the introduction of the state into server program fuzzing has received widespread attention. This study addresses the problem of low efficiency of fuzzing due to the insufficient utilization of information in the protocol fuzzing process and the inability to continuously focus on key states. The study also proposes a protocol fuzzing method based on the cooperation of double cover information. Firstly, the state selection algorithm proposed in this study sets weights for each state by mapping the state space to the program space and using heuristic calculation methods to guide the fuzzing to continuously focus on states that are more likely to have defects. Secondly, the study detects a seed position that will not affect the state but can change the program coverage and restricts the mutation position to adequately test the code area corresponding to the focus state. The study also verifies the effectiveness of the improved algorithm on the baseline tools AFLNet and SnapFuzz and integrates them into a protocol fuzzing tool, namely C2SFuzz. Experiments are carried out on the latest version of protocol server programs such as LightFTP and Live555, and five unknown vulnerabilities are detected.

Key words: fuzz testing; vulnerability mining; protocol; state selection; mutation optimization

① 基金项目: 国家自然科学基金 (62072448)
收稿时间: 2023-02-17; 修改时间: 2023-03-14; 采用时间: 2023-03-30; csa 在线出版时间: 2023-07-21
CNKI 网络首发时间: 2023-07-24

1 引言

软件及协议栈作为信息系统的重要基础设施, 与我们的工作和生活联系越来越紧密. 然而近年来网络协议软件漏洞层出不穷, 一些高危漏洞对现实生产和生活造成严重危害. 如存在于 TLS 协议软件 OpenSSL 中的心脏滴血漏洞, 导致全球 2/3 的网站受到影响; WannaCry 勒索病毒利用的 SMB 协议程序漏洞进行传播, 全球 70 余国家十多万计算机遭受攻击. 网络协议服务端软件是 IT 基础设施攻击面的关键部分, 易遭受流量攻击导致拒绝服务, 或由恶意攻击者在服务器计算机上执行恶意代码以进行进一步的攻击. 相比于常规终端软件, 网络协议软件具有更复杂的程序逻辑, 涉及多种协议状态, 覆盖更多漏洞种类, 涵盖更加多样的程序语义信息, 分布于更多的平台架构中.

网络安全攻防领域非常注重时效性, 人工方式挖掘漏洞成本高、耗时长, 难以应对快速挖掘网络协议漏洞的需求. 与此同时, 模糊测试已经成为漏洞自动化挖掘领域的重要研究方向. 模糊测试是一种软件测试技术, 常用于检测软件或计算机系统的安全漏洞, 其核心思想是将自动或半自动生成的随机数据输入到一个程序中, 并监视程序异常, 如崩溃、断言失败, 即可能存在的诸如内存泄漏等程序错误[1]. 自模糊测试技术提出以来, 在真实软件中挖掘出大量安全漏洞, 对软件安全防护、软件质量提高起到了重要作用. 模糊测试工具通常可以被分为两类: 一是变异测试, 通过改变已有的数据样本去生成测试数据; 二是生成测试, 通过对程序输入的建模来生成新的测试数据. 相比符号执行、污点分析等其他漏洞挖掘技术, 模糊测试具备易部署、轻量级等优势, 可快速、高效对目标软件进行测试, 对现代大型软件具有良好适应性.

随着网络协议在各方面的应用, 越来越多的研究开始针对协议中存在的安全性问题进行研究. 目前大多数针对协议进行模糊测试的测试目标都是在协议服务器端 (service under test, SUT), 因为客户端可能没有实现全部的协议功能. 与传统应用程序不同, 针对协议程序的模糊测试过程需要应对额外的挑战. 首先, 网络协议是客户端和服务器之间网络数据通信的一种协议规则. 协议通过指定通信过程中发送的数据结构和规则, 使连接的设备或程序模块进行交互[2]. 因此, 在模糊测试过程中, 所构造的测试用例不能像传统应用的模糊测试一样通过随机突变进行构造, 而需要在一定程度上满足协议的规范, 否则发送给目标程序时容易被丢弃, 使得模糊测试的探索过程难以进入较为深层次的程序区域, 导致模糊测试效率低下. 其次, 许多网络协议都是有状态的, 客户端和服务器需要在有状态的会话中保持通信. 这种状态需要在模糊测试中进行维护, 否则通信双方将无法正常通信, 从而导致模糊测试的失败[3]. 因此在模糊测试的过程中, 需要维护相应的自动状态机来实现状态的控制.

为了应对上述两个关于协议模糊测试的挑战, 目前一部分工作主要是通过理解网络协议的内部实现和网络协议相关的输入格式来生成有效的测试用例, 并将有效的测试用例用于后续的模糊测试过程, 例如 Peach[4] 和 BooFuzz[5]. 但这类工具的测试效率很大程度上取决于对网络协议规范和通信数据格式语法的理解, 由于自动化程度较低, 需要较多人工干预, 阻碍了其大规模应用. 近年来, 随着人工智能技术的发展, 安全人员开始研究如何将机器学习技术应用于协议模糊测试用例生成中, 例如使用 seq2seq 模型的 SeqFuzzer[6]. 这类方法需要构造目标协议的测试用例训练集, 训练模型学习协议的语义和结构特征. 但对这种方法而言, 数据集的质量对训练效果有很大影响, 且针对不同协议需要不同的模型和训练集, 测试代价较高.

另一类基于 AFL 变异思想的工作是当前的主流方法, 它们通过对初始语料库中的消息序列进行突变生成新的测试用例, 并按照消息执行的反馈结果指导下一次的变异, 例如 AFLNet[7] 和 SGFuzz[8]. 这类工具能够自动化的构建状态机, 并且不需要对协议格式具备先验知识. 但当前这部分工作仍然存在两个问题: 1) 虽然这些工具已经将状态作为重要信息, 并与覆盖率引导的思想相结合共同指导模糊测试, 但是两者使用时较为独立, 并没有进行关联分析; 2) 变异策略未根据协议场景做相关调整, 生成无效样本的概率较大. 再者, 对于模糊测试关键的种子调度而言, 协议模糊测试的场景下只关注种子这一级别无法实现提高测试效率的目的. 总体来看种子的调度是分阶段的, 种子的选择过程是在协议状态选择之后. 现有的工作在协议状态选择方面, 使用较为简单或原始的状态选择方法, 会对种子调度和漏洞探索的性能造成影响.

对于上述两个问题, 本文提出了对应的改进方案: 在状态选择阶段充分利用模糊测试的过程信息、建立程序空间与协议状态空间的联系, 设计并实现增强的

协议过程状态机和基于此的状态选择算法; 在变异阶段使用状态选择感知的变异策略充分探索对应的程序空间, 通过上述利用状态覆盖和程序覆盖双重信息协同的改进, 提高协议模糊测试发现漏洞的能力. 本文主要贡献如下.

首先, 本文建立了程序空间到状态空间的映射, 并基于此辅助模糊测试工具选择重点状态以进行优先测试. 本文将状态机进行扩展, 增加多个模糊测试过程间信息, 构建基于增强状态图的模糊测试. 最终的目的是选择有价值的种子分配更多的能量, 对其临近的分支进行充分的探索. 对于协议模糊测试而言, 有价值的种子应能够触发有价值的状态. 于协议状态图层次越深、关联状态路径越多、映射程序空间越大的状态, 模糊测试的价值也越大. 因此, 本文增加了状态层次、关联路径等过程间信息, 以及程序空间对应于状态空间的映射关系信息, 并基于该部分信息和启发式的计算方法为每个状态赋予权重, 以权重为状态的被选择概率, 使得更有价值的状态能够优先被探索.

其次, 本文提出了状态选择感知的变异算法以提升重点状态对应的程序空间的覆盖. 对于选中状态所对应选取的消息序列, 使用状态选择感知的变异策略. 即为减少变异的盲目性, 以一定概率只变异能够触发该状态的单条消息. 首先使用快速的单字节变异算法对该消息进行变异, 将不影响状态覆盖但影响程序空间覆盖的位置识别为可变位置. 然后对该部分位置进行有针对的变异, 尽可能覆盖该状态对应的程序空间.

最后, 在 AFLNet 和 SnapFuzz[9] 上实现并验证了本文提出的算法的有效性: 在两个工具基础上, 添加了基于增强状态图的状态选择算法和状态选择感知的变异算法, 实现了 C2SFuzz 的两个原型版本. 本文在常用的 4 个协议服务端程序上进行了测试, 实验结果表明加入了本文改进算法的模糊测试结果覆盖率有所提升, 并挖掘到 5 个未知的漏洞.

2 研究背景

本节主要介绍协议模糊测试的相关概念、技术和工作流程.

2.1 基本概念

网络协议: 网络协议是指在计算机网络中相互交互的实体之间, 信息交换必须遵守的一组规则. 网络协议规范由 RFC 定义, 包含计算机网络通信的 3 个关键要素, 即语义、语法和时序[10]. 具体来说, 语义指要交换的内容, 语法指定如何在网络中通信, 时序定义通信顺序. 如上所述, 协议模糊测试中的测试用例实际上是一个包含了多条消息的消息序列. 模糊测试的基础是保证测试用例能够被目标程序接受, 从而覆盖更多的代码区域. 为了通过目标程序的语法检查, 每条消息都需要满足协议格式, 发送消息的顺序也要满足协议的状态机模型.

消息结构: 消息结构描述了消息的内容和数据格式, 一般包括消息头和数据部分. 消息头决定了消息的协议类型、控制标志和携带的数据大小, 一个有效的模糊测试应该尽可能保持数据格式, 以便成功触发状态转换并探索目标程序中更多的代码区域.

协议状态机: 协议状态机是一种描述网络协议转移的合适的形式化描述方法. 它描述了状态的数量和他们之间的转换关系. 现有的模糊测试工具可以根据状态机指导模糊测试的进程, 如遍历有限状态机并利用在特定状态下接收到的消息模型生成新消息序列, 或以状态机的覆盖为导向引导模糊测试.

2.2 灰盒协议模糊测试

随着 AFL 的出现, 以覆盖率为导向的灰盒模糊测试逐渐成为学术界的主要研究方向, 许多研究工作将其应用到协议场景. 下面简要介绍一下现在主流的灰盒协议模糊测试的主要流程. 如图 1 所示, 模糊测试器以正确的顺序从捕获的网络流量中提取和解析单个请求, 并根据服务器的响应信息构造初始状态机. 然后模糊测试将进入主循环, 它将依次重复执行消息选择、消息变异、网络交互和监控反馈. 消息选择是从消息集合中选择最具有潜力的消息作为种子. 消息变异对于所选种子执行的单个或多个变异, 以生成由一个或一系列变异消息组成的测试用例. 网络交互通过顺序发送从测试用例中选择的消息并接收相应的响应, 与目标被测协议程序进行交互并实时更新状态机. 状态机将指导种子选择和变异操作, 模糊测试器维护状态和种子执行队列的映射关系. 最后监控反馈将检查目标程序以收集执行信息, 并根据该信息确定测试用例是否有足够的潜力加入消息集并等待下一次选择.

由上述信息可知, 状态机是在模糊测试的运行时构建的. 在协议结构解析完成后, 构造消息向测试对象发送请求消息 req_mes, 然后模糊测试工具接收响应, 从返回信息中解析状态码 stat_code, 如果此 stat_code

为从未发现过的, 则说明协议进入了新的状态, 将其添入协议状态机. 状态机 M 为一个有向图<V, E>, V 为状态节点集合, E 为边集合.

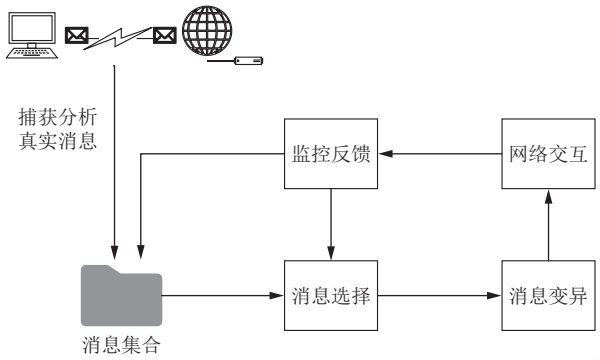


图 1 灰盒协议模糊测试流程

如图 2 所示, 对协议模糊测试可以抽象成 4 层[11], 在上述的灰盒协议模糊测试过程中, 状态机模型是被测协议程序的最高抽象层, 它位于程序调用图、控制流图和二进制文件之上. 选择不同的上层区域将激活对应的下层不同区域, 所以对下层的探索依赖于上层的选取. 有大量研究在程序调用图和控制流图级别评估探索算法以提高模糊测试的有效性, 如研究路径、函数和基本块选择算法对模糊测试性能的影响. 但对于协议模糊测试的实际执行过程, 最先被模糊测试工具选择的是状态, 之后再选择映射到该状态的队列中的种子. 因此, 状态选择算法将会影响最终的种子选择. 如果不考虑状态对应于代码覆盖的关联性, 随机或者低频选择某状态, 可能导致代码缺陷的状态难以被充分测试.

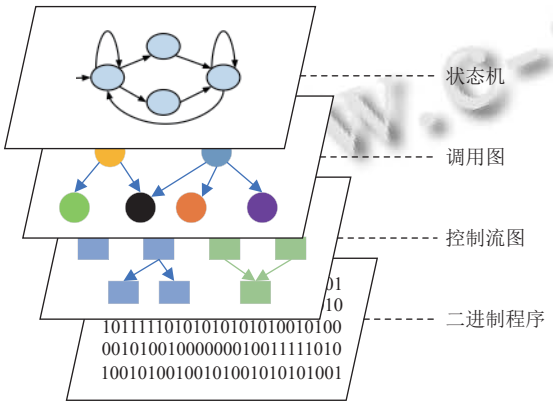


图 2 协议程序的 4 层抽象

3 方案设计

如上所述, 现有协议模糊测试方法存在无法持续关注重点状态、序列变异过于随机等问题. 针对上述问题, 本文通过: 1) 提出增强协议过程状态机的概念, 利用该状态机在状态选择阶段建立起两者之间的关联, 使二者协同引导模糊测试; 2) 在种子变异过程, 提出状态选择感知的变异算法识别变异区域并进行有针对性的变异, 以持续测试重点状态的代码区域, 最终提升协议模糊测试的效率.

具体而言, 首先, 由状态构建器通构建增强的协议过程状态机来增加更多的状态相关信息, 建立起状态空间到程序空间的联系; 然后基于权重的状态选择器根据相关的信息对不同的状态赋予权重, 并引导模糊测试工具优先测试重点状态; 最后修改多级选择之后种子的变异策略, 在种子变异器中新增了状态选择感知的变异方法, 以一定概率确保种子只变异不影响状态覆盖但影响程序空间覆盖的子节, 最大程度确保对被选中状态对应的程序空间的探索. 经由上述改进, 本文最终实现了一个由多级选择算法和多类变异算法组成的高效协议模糊测试系统 C2SFuzz (code to state fuzz). C2SFuzz 的整体工作流程如图 3 所示, 状态机构建器、状态选择器和种子变异器的功能如前文描述, 种子选择器从语料库中选择种子, 消息池存放抓包得到的真实消息.

3.1 增强协议过程状态机 (APISM) 的构建

当前的 AFLNet、SGFuzz 等主流灰盒协议模糊测试工具使用的状态机模型通常只存储了朴素的状态信息. 状态信息由协议的反馈码或状态变量来标识. 在状态机的实现中, 并无与程序空间关联的信息, 这种设计正是影响两种信息无法相互协同的原因. 因此, 本文对状态机 M 的结构进行了扩展, 设计实现了增强的协议过程状态机 (APISM) 以建立程序空间到状态空间的映射, 并增加更多的过程间信息.

首先将 M 扩展为新的有向图结构<V, E, Σ: $V\rightarrow$ {req_mes}>, Σ 是记录状态和对应消息的映射. 若 req_mes 发送前的协议状态为 v, 那么返回 stat_code 后, 将 stat_code 加入 V, $v\rightarrow$state_code 加入 E, 映射关系<state_code, req_mes>加入 Σ. 其次, 将状态节点扩展为<pnode, depth, discovered_paths, covered_bits, selected_times, fuzzed_times>, 其中,

pnode$\in V$, 表示该状态的前向状态节点;

depth$\in N$, 表示该状态所处的深度;

discovered_paths$\in N$, 表示测试该状态时探索到的

状态转移路径数量;

covered_bits∈M, 表示测试该状态能够覆盖到的程序位图, 其中 M 为二维矩阵;

selected_times∈N, 表示该状态被选中的次数;

fuzzed_times∈N, 表示该状态被测试的次数.

在一个协议状态图 M 中, 对于当前的状态节点 $v \in V$, 其深度记为离入口节点 $v0$ 的距离 $d(v, v0)$, 计算方法同图上两点的距离计算. 显然, 深度值越大的状态节点越难被探索到[12]. discovered_paths 是指当选择探索某状态 v 时, 能够发现的不同状态转移路径. 在实际计算时, 对所有的状态转移路径 state_path 计算哈希值, 并统计唯一哈希值∑unique(hash(state_path)), 得到的统计值作为该状态的 discovered_paths. 其中 state_path 为一组状态序列, 函数unique(·)的含义为统计不同 hash 值的数量. discovered_paths 越大, 说明以当前状态为目标状态时, 探索到的状态转移路径数量越多. 则该状态与其他状态的关联度越高, 也意味着该状态处在整个状态图的位置越重要. covered_bits 是探索当前状态时, 执行的测试用例能够覆盖达到的边覆盖的 bitmap, 模糊测试工具对每一个状态维护一个独立的 bitmap, 在模糊测试的过程中负责记录该状态下的种子执行时对程序的覆盖情况, 然后对每个状态机节点计算对应的消息集合执行时位图覆盖 Σbits(req_mes∈v), 函数 bits(·)的含义为计算测试用例执行时的位图覆盖. 随着模糊测试过程的进行, 模糊测试工具将建立起协议状态空间到程序空间的映射关系<V→{covered_bits}>. 同上述过程, 可以得知对于协议状态v_i, 到达该状态时可能触发的程序覆盖情况covered_bits$_i$. 如果状态 v 的 covered_bits 值越大, 说明该状态对应的程序空间越大. 相应的, 增加对这种状态的探索也能有可能发现漏洞. 状态被选择的次数 selected_times 和状态被测试的次数 fuzzed_times 用来维护模糊测试中新状态探索和现有状态利用两个过程的平衡, 防止模糊测试工具只关注对现有状态的测试.

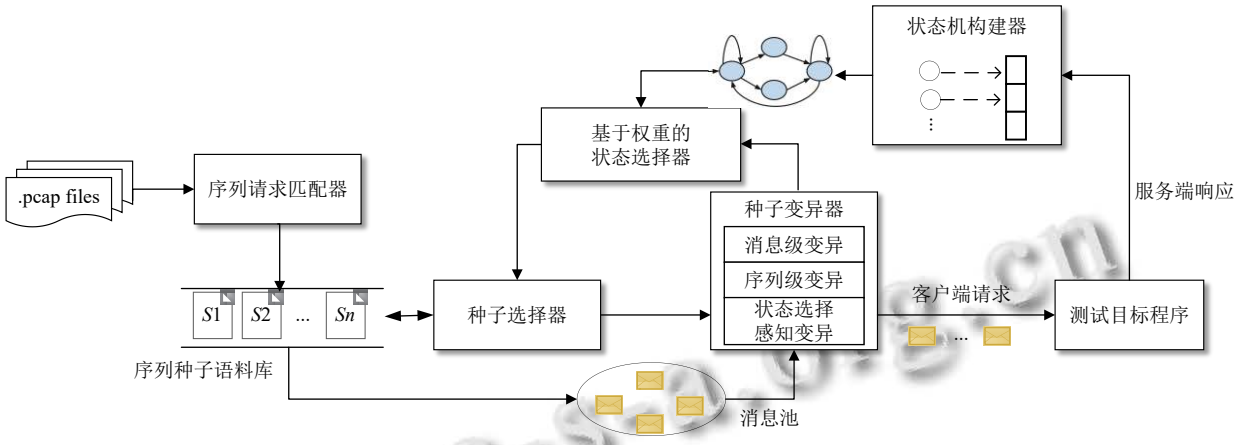


图 3 C2SFuzz 工作流程

3.2 基于 APISM 的状态选择算法

本文对状态的定义和识别延续了 AFLNet 的方法, 即以服务端响应码标识程序所处的状态. 由于并非所有的状态都同等重要, 并且模糊测试活动有时间限制, 因此模糊测试工具需要优先对重点状态进行测试以尽快发现可能的漏洞. 本文利用 APISM 为每个状态计算权重, 并使用启发式的算法来将更多的能量分配给重点状态. 权重计算基于以下几个认识.

(1) 网络通信程度越深的代码区域越难被测试. 由于状态的前向依赖性, 状态的通信程度越深, 构造可以触发它的消息序列越困难, 因此更难对其充分测试, 也越有可能在其中潜藏漏洞.

(2) 对于处在状态机关键位置的状态, 其关联的状态转移路径更多, 选择这类状态进行优先探索能够解锁更多的未知区域.

(3) 状态对应的程序空间越大, 其实现的功能越复杂. 由于该状态的代码复杂度更高, 对其进行全面的测试更困难, 因此也应该被优先测试.

(4) 在探索已有状态时, 应考虑对未知状态的探索, 防止状态机结构不完整.

本文基于上述的原则提出了权重计算的公式, 对于每个状态v_i, 权重计算为:

$$w_i = \frac{\text{depth}_i \times \text{discovered\_paths}_i \times \text{covered\_bits}_i}{\lg \text{selected\_time}_i + \lg \text{fuzzed\_time}_i} \quad (1)$$

计算得到的权重将作为每个状态被选中的概率, 所处状态越深、与状态机路径关联越大、对应的代码区域越大的状态, 权重计算越大, 被选中的可能性越高. 同时, 被选中和测试次数过多的状态, 在上述计算时权重将被限制. 由于 selected_time 和 fuzzed_time 的数量级太大, 为避免分母过大导致权重值区分度太小, 计算过程中对其降低了数量级处理.

以若干 FTP 请求序列构建出的状态机为例 (如图 4), 初步验证上述状态选择权重计算的合理性. 可以看到 250 状态离入口状态距离较深, 且关联多个状态迁移路径, 在实际探测过程中对应的代码区域较大, 则按照本算法将会被优先选中. 经过手动分析, 250 状态是对文件或目录操作成功的响应码, 关联 CWD、DELE、LIST、NLST、STOR 等 10 多个 FTP 命令, 是 FTP 协议中关联命令最多的响应码, 为其分配更多的能量进行测试是合理的策略.

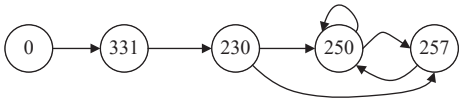


图 4 FTP 请求状态转换示例

需要说明的是, 在模糊测试初始阶段, 本文提出的工具只构造 APISM 但不使用上述的状态选择算法. 因为该阶段的状态机是由初始语料库构构造的, 状态机的完整性取决于初始语料库的多样性. 本文的策略是使用随机状态选择一段时间后, 再开启本文的状态选择算法. 时间可以由用户手动设置, 一般选择为语料队列完成 5 次循环以后, 也可以由用户手动设置. APISM 中的信息随模糊测试不断更新, 由于本算法计算开销较小, 不影响模糊测试整体的效率.

3.3 状态选择感知的变异算法

现有协议模糊测试工作, 对种子的变异可以分为消息级别的变异和序列级别的变异. 消息级别的变异是对所有的消息依次进行变异, 序列级别的变异是将多个序列进行裁剪拼接. 这两种变异操作都涉及对多条消息的处理, 因此无法保证测试用例到达目标状态, 更无法保证对目标状态对应的程序空间进行更细粒度的探索. 为了提高被选中的状态对应的程序空间的覆盖率, 本文提出以一定概率进行状态选择感知的变异过程, 通过算法识别消息序列的可变位置, 只对真正触发目标状态的单条消息进行变异, 从而限制协议模糊测试变异算法的无序性.

具体而言, 首先需要分析状态信息得到对应的消息序列编号, 为此提出状态链分析算法, 细节如算法 1 所示. 对于一个序列中多次触发某个状态的情况, 以 RTSP 协议的服务端程序 Live555 为例, 对部分实验结果进行了观察, 发现由于不依赖协议格式的灰盒测试变异的盲目性, 模糊测试中会生成大量的格式无法通过检查或序列号错误的样本. 在处理这一序列期间, 这些样本可能会触发不同的错误码, 导致服务器会在多个 4** 状态之间循环. 虽然错误处理代码也是需要关注的部分, 但如果大量种子陷入到此类型下, 将导致模糊测试种子集合质量下降, 不利于对实际功能对应的代码区域的测试. 基于上述对这种现象的分析, 本文设计了目标消息的选择依据. 首先, 计算每一个目标状态和其对应的前驱状态序列的独特率 unique_rate, 这代表了该段状态序列中状态类别的复杂度. 由于陷入错误处理循环的状态序列中, 代表错误码的状态会循环出现, unique_rate 的值也会变小. 因此每次优先选择 unique_rate 最高的目标消息, 减少了前置消息中错误消息数量. 这样可以保证该消息和前置消息组成的消息序列多样性最高, 目标消息触发程度越深.

算法 1. 状态链分析算法

输入: 状态机 M, 状态 m, 触发该状态的种子 s.
输出: 被选中变异的消息编号.

```
1 state_set = <>
2 max = 0, index = 0
3 for i in len(s):
4    m′ = get_state(s′)
5    update(state_set)
6    if m′ == m:
7          unique_rate_i = size(state_set)/i
8          if unique_rate_i > max:
9                max = unique_rate_i
10               index = i
11         end if
12   end if
13 end for
14 return index
```

获取完整输入格式可以显著的提高模糊测试的有效性. 然而, 这种方法通常扩展性较差, 并限制了测试的自动化程度. 完整的规范可能面向的是为用户提供

更好的软件服务, 而不是面向模糊测试本身对规范的需求, 即这部分规范对提高程序覆盖和漏洞发现并无帮助. 模糊测试不需要全面的, 特定于某种应用程序和语义丰富输入的规范[13]. 作为普适性和有效性的权衡, 本文使用了轻量级探测的方法. 在状态选择感知变异的第 2 阶段, 变异算法通过单字节快速变异自动发现可变位置.

算法的流程如算法 2 所示. 如果变异该字节能够保持现有的状态触发, 但位图覆盖发生变化, 则将该字节记录到候选变异位置. 得到所有的候选位置后, 对该部分字节进行变异, 以提高该状态对应的程序空间的覆盖率.需要注意的是, 本文的变异算法是为了与状态选择算法协作, 提高重点状态对应的程序空间覆盖率. 而序列级别的变异操作是为了发现新状态和新的状态转移路径, 并且会降低种子格式检查的通过率. 因此, 在本文的状态选择感知的变异中, 不涉及序列及别的变异. 但由于只以一定概率启用状态选择感知变异算法, 所以在整个变异过程中也保留了原有的序列级变异操作.

算法 2. 状态选择感知的变异算法

输入: 状态机 M, 状态 m, 触发该状态的种子 s, 目标消息序号 i.
输出: 保持状态 m 触发的新种子.

```
1 mutate_poses = len(s_i)
2 legal_poses = <>
3 for mutate_pos in mutate_poses:
4    s′ = mutate(s)
5    m′ = get_state(s′)
6    if m′== m:
7          legal_poses.append(mutate_pos)
8    elif m′ not in M:
9          M.add_state(m′)
10         m′.add_seed(s′)
11   end if
12 end for
13 for pos in legal_poses:
14      s′= deter_mutate(s)
15      fuzz_run(s′)
16 end for
```

4 实现与评估

4.1 系统实现

Liu 等人[11] 测试了 AFLNet 的几种状态选择配置, 发现在代码覆盖率方面结果非常相似, 分支覆盖率之间的最大差异为 1.15%. 他们分析存在的原因主要有两个, 一个是协议模糊测试较低的吞吐量, 二是变异策略未针对协议场景做充分改进. 可见若协议模糊测试工具未对上述两个问题做出有效的改进, 不同算法的结果将相差不大, 也无法体现出本文算法的优势. SnapFuzz 使用了异步网络通信, 弥补了 AFLNet 吞吐量较低不足的缺点. 但当前仍没有对协议模糊测试变异过程非常有效的改进工作. 因此, 本文选择 AFLNet 和 SnapFuzz 作为基准测试工具, 分别在这两个工具上实现了 C2SFuzz 的两个版本, 各增加了约 1500 行 C 代码.

4.2 实验评估

本节将评估本文提出的协议模糊测试改进的有效性, 模糊测试通常选取程序覆盖率和发现的崩溃、漏洞数量作为评估指标[14], 因此在本节中将评估以下几个问题.

1) 加入状态选择算法是否能够提高测试过程中发现崩溃的数量?

2) 加入状态选择感知的变异算法是否能够提升协议模糊测试过程中的吞吐量和最终的程序覆盖?

3) C2SFuzz 能否发现真实程序中的漏洞?

本文的实验是在一台型号为 NaviData 5200 G3 的服务器进行的, 该服务器拥有 2 个 Intel(R) Xeon(R) Gold 5218 CPU, CPU 主频 @ 2.30 GHz, 该机器 CPU 一共拥有 32 核心 64 线程. 本文选择 DICOM、DTLS、FTP、RTSP 作为基准测试的网络协议, 这些协议是发展成熟、被企业和用户广泛使用的协议, 并且在以前的模糊测试工作或安全分析中被选作目标. 对于每个选定的协议, 选取适合该协议的开源服务端软件已进行模糊测试. 选取的结果展示在表 1 中, 对于每个协议, 该表都简要描述了该协议的功能和状态信息. 本文选择 AFLNet 和 SnapFuzz 作为基线工具, 分别与对应的 C2SFuzz 版本进行对比实验.

表 1 测试程序信息

协议	开源实现	功能	状态信息	相关文献
DICOM	Dcmqrscp	图像传送	会话进程	[7,15]
DTLS	TinyDTLS	安全数据报通信	用户认证, 会话配置	[7,16]
FTP	LightFTP	文件传输	CWD, 会话标志, 会话进度	[7,17,18]
RTSP	Live555	实时流媒体传输	流式传输进程	[7,19]

4.2.1 验证状态选择算法的有效性

协议模糊测试过程中, 状态选择算法最终会影响

到选择的代码区域. 相较于随机的状态选取, 加入了状态选择后重点状态被优先选中, 所以能提高崩溃触发的概率. 首先在 AFLNet 和 SnapFuzz 上实现了状态选择算法, 在 24 h 内统计了崩溃和挂起的出现情况, 结果如表 2 所示, “-S”代表加入本文相关算法 APISM 的改进版本. 可以看到, 加入了状态选择算法的改进版本在发现崩溃数量上明显优于原版本. 由于状态选择算法能够有偏向性的选择可能存在代码缺陷的状态, 因此触发崩溃的效率更高.

表 2 触发崩溃数量

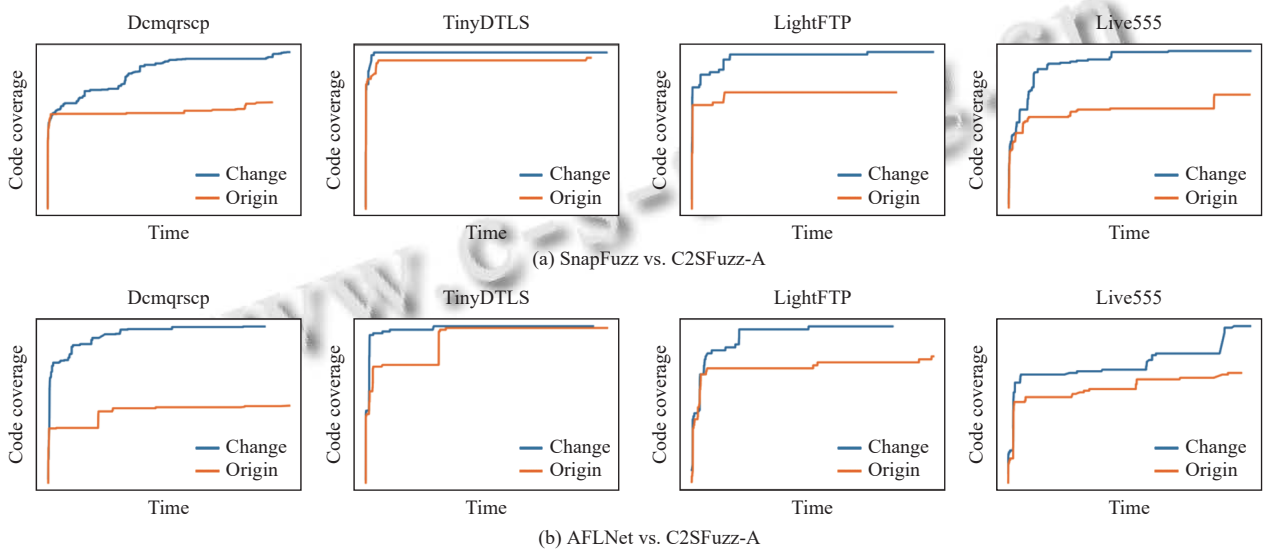
测试程序	SnapFuzz-S	SnapFuzz	AFLNet-S	AFLNet
Dcmqrscp	4	0	0	0
TinyDTLS	47	39	31	22
LightFTP	0	0	0	0
Live555	103	75	44	6

崩溃数量统计结果说明了状态选择算法能够使模糊测试工具持续关注重点状态, 增加了重点状态的触发概率.

4.2.2 验证变异算法的有效性

由分析可知, 限制状态级变异的无序性可以充分探索状态对应的程序空间, 从而提高整体的代码覆盖率; 同时, 种子更容易通过协议格式的检查, 相同时间内执行的种子数量更多, 系统的吞吐量也越高.

在上述的 AFLNet-S 和 SnapFuzz-S 改进的基础上, 加入了状态选择感知的变异算法, 实现了 C2SFuzz 的两个版本 C2SFuzz-A 和 CSFuzz-S. 区别在于, 基于 SnapFuzz 实现的 C2SFuzz-S 版本比基于 AFLNet 实现的 CSFuzz-A 具有更高的吞吐量. 将其与各自的原始版本进行比较, 首先分析了 24 h 后的覆盖率情况, 发现每个测试目标都有不同程度的提升, 覆盖率的变化情况如图 5 所示, 最终的覆盖率增长情况如表 3 所示. 经过分析两者上表现, 存在差别的主要原因是协议测试的配置、模糊测试的吞吐量差距. SnapFuzz 针对协议测试过程中的等待时间进行了修改, 相同时间内能够生成和执行更多的测试用例. 由此可以看出性能越好的模糊测试工具上本算法的表现也越好. 表 4 展示了 24 h 内的吞吐量. 由结果可知在两种基线工具上, 本文的改进都不同程度提升了模糊测试的吞吐量. 说明加入 x 的变异算法对协议模糊测试有增益作用. 由于 TinyDTLS 实现的是二进制协议, 在响应消息中缺少状态码, 而本文并未对状态识别进行改进. 在最终的实验结果中, C2SFuzz 只实现了较弱的覆盖率提升. 实际上, 实验中的 fuzzer 对 TinyDTLS 的测试覆盖率都非常低 (0.9%–1%).



(a) SnapFuzz vs. C2SFuzz-A

(b) AFLNet vs. C2SFuzz-A

图 5 程序覆盖率对比

通过上述对程序覆盖情况和吞吐量分析, 可以说明本文提出的变异算法减少了种子变异的无序性, 更多变异后的种子能够成功触发目标状态, 并对该状态对应的代码进行更细粒度的测试.

4.2.3 验证 C2SFuzz 的漏洞挖掘能力

本节选用 SnapFuzz 和 C2SFuzz-S 版本, 在被测程

序的最新版本上进行了 24 h 实验, 并发现了大量的崩溃样本. 对这些样本逐个分析后, 整理的漏洞信息如表 5 所示. C2SFuzz-S 最终发现了 5 个漏洞, 而 SnapFuzz 只发现了列表中 Live555 的两个 Heap-UAF 漏洞. 经过对表 5 中 Live555 漏洞触发状态的差异分析, 两个 Heap-UAF 触发的状态分别为 0-201-405-200-404-404 和 0-200-201, 状态转移关系相对简单; Stack buffer overflow 触发的状态为 0-200-201-400-405-404-400-205-400-204-454, 状态结构较为复杂, 且触发漏洞时的状态在整个状态图中属于较难充分测试位置. 相对于 SnapFuzz, 本文的方法更容易选择和触发到该状态. 其他两个漏洞都有相似的情况. 我们将相应的漏洞信息反馈给作者, 并等待 CVE 审核. 通过在实际程序的最新版本上进行漏洞挖掘并发现了 5 个未知漏洞, 证明 C2SFuzz 具有实际发现程序漏洞的能力.

表 3 覆盖率提升 (%)

测试程序	SnapFuzz提升比例	AFLNet提升比例
Dcmqrscp	5	9
TinyDTLS	2	0.7
LightFTP	17	10.4
Live555	8	3.2

表 4 模糊测试吞吐量

测试程序	C2SFuzz-S	SnapFuzz	C2SFuzz-A	AFLNet
Dcmqrscp	897	381	301	225
TinyDTLS	241	154	230	187
LightFTP	1 790	1 684	330	254
Live555	1 201	937	441	374

表 5 漏洞挖掘情况统计

协议程序	文件	函数	漏洞类型
TinyDTLS	sha2/sha2.c	dtls_sha256_transform()	Global-buffer-overflow
Live555	liveMedia/include/Media.hh	envir()	Heap-use-after-free
Live555	liveMedia/MPEG1or2Demux.cpp	newElementaryStream()	Heap-use-after-free
Live555	liveMedia/RTSPServer.cpp	lookForHeader()	Stack-buffer-overflow
Dcmqrscp	Dcmnet/libsrc/dulparse.cc	parseUserInfo()	Nullptr-dereference

5 相关工作

按照对被测实体的了解情况, 协议模糊测试方法可以分为黑盒、白盒和灰盒方法, 目前大多数的工作都是黑盒或灰盒方法.

黑盒模糊测试通常使用一些预定义的规则, 构造结构良好的测试用例. 代表性的黑盒模糊测试器如 Peach、Boofuzz 通过手工编写协议规范的描述文件, 然后基于这类文件自动生成种子. 2015 年, de Ruiter 等人开创了分析通过模型学习所获得的协议状态机, 以进行系统化的状态模糊测试的方法. 他们分析了常见的 TLS 协议实现[20], 从协议实现中推断出状态机, 并人工分析了这些状态机的差异, 找出可能暴露出程序逻辑缺陷的行为. 这类方法被研究人员应用到了其他的一些安全攸关的协议上, 如基于模型目视检查方法对 OpenVPN[21] 协议分析、基于模型学习和检查的方法对 TCP[22]、SSH[23]、IPSec[24] 协议进行分析. 除传统的黑盒测试方法之外, 有研究人员提出了基于神经网络的黑盒测试方法. 如 2019 年 Zhao 等人提出的基于 seq2seq 模型的 SeqFuzzer[6], 可以从工控协议流量中学习协议格式以生成测试用例; 2022 年 Yu 等人提出的 CGFuzzer[25], 使用了基于覆盖的生成对抗网络 (CovGAN) 生成具有高通过率的测试用例.

尽管黑盒方法简单且易于部署, 但由于缺乏协议实体内部的逻辑, 实际效率低下. 与模型学习和检测相结合的改进方法虽然能提升效率, 但需要增加对协议的先验了解, 并且自动化程度较低. 虽然结合深度学习技术的方法能够提高测试用例的质量, 但是模型的训练依赖于观测和收集到的流量. 并且针对不同协议需要使用对应的流量数据集训练的模型, 测试代价较高. 这也是近年来白盒方法和灰盒方法受到极大关注的原因. 白盒协议模糊测试不同于黑盒方法, 它可以获得协议实体的详细信息, 包括源代码、协议规范和运行时信息, 并使用这些信息来指导测试用例的生成. MACE[26] 结合了混合符号执行和主动状态机学习以进行模糊测试, 它使用具体执行和符号执行来迭代推断和提炼协议的抽象模型. 理论上白盒方法可以覆盖协议实体所有的路径, 但在实际使用时, 由于真实协议实体中的执行路径较多和符号执行中约束解的不确定性, 白盒方法无法到达 100% 的代码覆盖率, 并且由于符号执行的繁重机制, 也会引发性能的可扩展性问题[27].

灰盒协议模糊测试是白盒的一种变体, 这些模糊器在执行过程中观察程序状态, 并使用覆盖率反馈和状态反馈来指导测试用例的生成. IJON[28] 使用人工代码注释来标记状态. INVSCOV[29] 使用 kernel 的不变量

来划分程序状态. AFLNet 不需要协议规范, 而是使用变异的方法生成测试用例. 它使用响应码识别服务器的状态, 对于每次运行, AFLNet 选择一个状态并选取能够到达该状态的消息序列. AFLNet 内置了几种简单的状态选择算法, Liu 等人[11] 也评估了概率模型的状态选择算法, 但最终结果并未优于其中的简单启发式算法. 但上述的诸多方法只考虑状态频率和连接关系, 并未与程序覆盖进行协同. 后续灰盒协议模糊测试工具对协议模糊测试不同方向进行了优化, 如 StateAFL[30]、NSFuzz[31]、SGFuzz[8] 等使用关键变量来标识协议程序的状态, SNPSFuzzer[32] 使用快照技术、SnapFuzz[9] 使用异步网络通信加快模糊测试. 但上述工作均没有改进状态选择过程, 与本文的改进方向是正交的. 本文的方法使用了两种覆盖信息协同的方法, 保证了有价值的状态优先被测试. AFLNet 虽然扩展了序列级的变异, 但相对于协议格式而言变异仍过于随机. 本文提供了一种状态选择感知的变异策略, 能够保证该状态被选中后, 更多的种子能够到达该状态并充分探索对应的程序空间.

6 结束语

模糊测试是一种高效的漏洞挖掘方法, 能够发现运行在真实世界程序的漏洞. 本文针对协议场景下模糊测试工具未充分利用过程信息、难以持续关注重点状态导致效率低下的问题, 设计了一种基于双重覆盖信息协同的状态选择算法和对应的变异算法. 首先通过建立状态空间到程序空间的映射, 并使用启发式的方式为每个状态设置权重, 引导模糊测试持续关注更可能存在缺陷的状态. 然后一定概率进行状态选择感知变异, 通过消息链分析算法和快速探测算法确定变异候选位置, 候选位置由不改变原状态覆盖但影响程序空间覆盖的字节组成. 限制只变异该部分字节, 保证种子能够触发重点状态并充分探索该状态的程序空间. 本文实现了基于上述改进的协议模糊测试工具 C2SFuzz, 在测试目标上达到了不同程度的覆盖率提升, 并发现了 5 个未知的漏洞. 总体来说, 本文的方法取得了比较好的改进效果.

未来工作中, 我们会研究如何将本文的方法与现有的其他方向的改进工作进行协同. 在实验过程中发现状态的表征对程序测试结果具有重要意义, 在未来也将研究如何以较低的开销快速识别程序状态, 进一步提高协议模糊测试发现漏洞的能力.

参考文献

1 任泽众, 郑晗, 张嘉元, 等. 模糊测试技术综述. 计算机研究与发展, 2021, 58(5): 944–963. [doi: 10.7544/issn1000-1239.2021.20201018]

2 Pfrang S, Meier D, Friedrich M, *et al*. Advancing protocol fuzzing for industrial automation and control systems. Proceedings of the 4th International Conference on Information Systems Security and Privacy. Funchal: SciTePress, 2018. 570–580.

3 李伟明, 张爱芳, 刘建财, 等. 网络协议的自动化模糊测试漏洞挖掘方法. 计算机学报, 2011, 34(2): 242–255.

4 Zhang H, Zhang Z, Tang W. Improve peach: Making network protocol fuzz testing more precisely. Applied Mechanics and Materials, 2014, 551: 642–647. [doi: 10.4028/www.scientific.net/AMM.551.642]

5 Pereyda J. BooFuzz: A fork and successor of the Sulley fuzzing framework. https://github.com/jtpereyda/boofuzz. (2020-04-29)[2023-01-26].

6 Zhao H, Li ZH, Wei HS, *et al*. SeqFuzzer: An industrial protocol fuzzing framework from a deep learning perspective. Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST). Xi'an: IEEE, 2019. 59–67.

7 Pham VT, Böhme M, Roychoudhury A. AFLNET: A greybox fuzzer for network protocols. Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification (ICST). Porto: IEEE, 2020. 460–465.

8 Ba JS, Böhme M, Mirzamomen Z, *et al*. Stateful greybox fuzzing. Proceedings of the 31st USENIX Security Symposium. Boston: USENIX Association, 2022. 3255–3272.

9 Andronidis A, Cadar C. SnapFuzz: High-throughput fuzzing of network applications. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2022. 340–351.

10 Hu ZH, Pan ZL. A systematic review of network protocol fuzzing techniques. Proceedings of the 4th IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC). Chongqing: IEEE, 2021. 1000–1005.

11 Liu DG, Pham VT, Ernst G, *et al*. State selection algorithms and their impact on the performance of stateful network protocol fuzzing. Proceedings of the 2022 IEEE International

Conference on Software Analysis, Evolution and Reengineering (SANER). Honolulu: IEEE, 2022. 720–730.
12 Fang DL, Song ZW, Guan L, *et al*. ICS3Fuzzer: A framework for discovering protocol implementation bugs in ICS supervisory software by fuzzing. Proceedings of the 2021 Annual Computer Security Applications Conference. ACM, 2021. 849–860.
13 You W, Wang XQ, Ma SQ, *et al*. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP). San Francisco: IEEE, 2019. 769–786.
14 Klees G, Ruef A, Cooper B, *et al*. Evaluating fuzz testing. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto: ACM, 2018. 2123–2138.
15 Wang ZQ, Li QQ, Wang YZ, *et al*. Medical protocol security: DICOM vulnerability mining based on fuzzing technology. Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. London: ACM, 2019. 2549–2551.
16 Fiterau-Brostean P, Jonsson B, Merget R, *et al*. Analysis of DTLS implementations using protocol state fuzzing. Proceedings of the 29th USENIX Security Symposium. USENIX Association, 2020. 2523–2540.
17 Gascon H, Wressnegger C, Yamaguchi F, *et al*. PULSAR: Stateful black-box fuzzing of proprietary network protocols. Proceedings of the 11th International Conference on Security and Privacy in Communication Systems. Dallas: Springer, 2015. 330–347.
18 Takanen A, DeMott JD, Miller C, *et al*. Fuzzing for Software Security Testing and Quality Assurance. Boston: Artech House, 2018.
19 Gao ZC, Dong WY, Chang R, *et al*. Fw-fuzz: A code coverage-guided fuzzing framework for network protocols on firmware. Concurrency and Computation: Practice and Experience, 2022, 34(16): e5756.
20 de Ruiter J, Poll E. Protocol state fuzzing of TLS implementations. Proceedings of the 24th USENIX Security Symposium. Washington: USENIX Association, 2015. 193–206.
21 Daniel LA, Poll E, de Ruiter J. Inferring OpenVPN state machines using protocol state fuzzing. Proceedings of the 2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). London: IEEE, 2018. 11–19.
22 Fiterău-Broştean P, Janssen R, Vaandrager F. Combining model learning and model checking to analyze TCP implementations. Proceedings of the 28th International Conference on Computer Aided Verification. Toronto: Springer, 2016. 454–471.
23 Fiterău-Broştean P, Lenaerts T, Poll E, *et al*. Model learning and model checking of SSH implementations. Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software. Santa Barbara: ACM, 2017. 142–151.
24 Guo JX, Gu CX, Chen X, *et al*. Model learning and model checking of IPSec implementations for Internet of Things. IEEE Access, 2019, 7: 171322–171332. [doi: 10.1109/ACCESS.2019.2956062]
25 Yu ZH, Wang HL, Wang D, *et al*. CGFuzzer: A fuzzing approach based on coverage-guided generative adversarial networks for industrial IoT protocols. IEEE Internet of Things Journal, 2022, 9(21): 21607–21619. [doi: 10.1109/JIOT.2022.3183952]
26 Cho CY, Babić D, Poosankam P, *et al*. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. Proceedings of the 20th USENIX Conference on Security. San Francisco: USENIX Association, 2011. 10.
27 Zhang T, Jiang Y, Guo RS, *et al*. A survey of hybrid fuzzing based on symbolic execution. Proceedings of the 2020 International Conference on Cyberspace Innovation of Advanced Technologies. Guangzhou: ACM, 2020. 192–196.
28 Aschermann C, Schumilo S, Abbasi A, *et al*. Ijon: Exploring deep state spaces via fuzzing. Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP). San Francisco: IEEE, 2020. 1597–1612.
29 Fioraldi A, D'Elia DC, Balzarotti D. The use of likely invariants as feedback for fuzzers. Proceedings of the 30th USENIX Security Symposium. USENIX Association, 2021. 2829–2846.
30 Natella R. STATEAFL: Greybox fuzzing for stateful network servers. Empirical Software Engineering, 2022, 27(7): 191. [doi: 10.1007/s10664-022-10233-3]
31 Qin S, Hu F, Zhao B, *et al*. NSFuzz: Towards efficient and state-aware network service fuzzing. ACM Transactions on Software Engineering and Methodology, 2023. [doi: 10.1145/3580598]
32 Li JQ, Li SY, Sun G, *et al*. SNPSFuzzer: A fast greybox fuzzer for stateful network protocols using snapshots. IEEE Transactions on Information Forensics and Security, 2022, 17: 2673–2687. [doi: 10.1109/TIFS.2022.3192991]

(校对责编: 孙君艳)